

# Ensuring Schema Compliance in Neo4j: Post-Import Validation with Cypher Queries

---

*Schemakompatibilitet i Neo4j: Post-importvalidering med  
Cypher Queries*

**Kent Thang**

Supervisor : Adriana Concha  
Examiner : Olaf Hartig

## Upphovsrätt

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

## Copyright

The publishers will keep this document online on the Internet - or its possible replacement - for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

## Abstract

Schema validation is a critical component of graph data pipelines, ensuring data quality and consistency as graphs evolve and scale. In property graph databases, validation is commonly performed as a pre-validation step prior to data ingestion, which can limit scalability and increase pipeline complexity. This thesis investigates whether in-database, query-driven validation can improve scalability and maintainability while ensuring that production graphs remain schema-compliant.

The work first presents a systematic catalog of schema constraints commonly found in property graph schemas, including property-level, structural and contextual constraints. These schema constraints are then mapped to reusable Cypher query templates that identify violations directly within Neo4j, demonstrating that Cypher is sufficiently expressive to serve as a general-purpose validation language for property graphs beyond the scope of native database constraints.

To evaluate the practical implications of this approach, a prototype query-driven validation tool is implemented and integrated into an industrial Neo4j data pipeline. An empirical evaluation compares query-driven validation with a traditional pre-validation strategy across multiple dataset scales. The results show that query-driven validation achieves significantly higher throughput and lower end-to-end processing time for large datasets, with performance gains increasing as dataset size grows. While pre-validation prevents invalid data from being ingested, query-driven validation introduces a trade-off by allowing temporary violations, shifting responsibility for remediation to the application layer.

Overall, the findings indicate that query-driven validation can substantially improve the scalability of graph data ingestion pipelines while maintaining expressive and flexible schema enforcement. The thesis concludes by discussing limitations related to data model dependency, update handling and invalid data management while outlining directions for future work such as hybrid validation strategies and automated remediation of detected violations.

# Acknowledgments

I would like to thank my examiner, Olaf Hartig, for his integral role in bringing this thesis to completion. His supervision and feedback were invaluable and greatly contributed to the quality and coherence of the work.

Furthermore, I would like to thank my company supervisor, Pia Løtvedt, for her guidance on experimental design and implementation as well as for her work in enabling the experiments involving pre-validation.

Lastly, I would like to thank the love of my life, Alyssa Baker, for her unwavering support and for keeping me focused and grounded throughout the thesis work and the thesis defence.

# Contents

<b>Abstract</b>	<b>iii</b>
<b>Acknowledgments</b>	<b>iv</b>
<b>Contents</b>	<b>v</b>
<b>List of Figures</b>	<b>vii</b>
<b>List of Tables</b>	<b>viii</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Motivation . . . . .	1
1.2 Aim . . . . .	2
1.3 Research Questions . . . . .	2
1.4 Approach . . . . .	2
1.5 Contributions . . . . .	2
1.6 Delimitations . . . . .	3
<b>2 Background</b>	<b>4</b>
2.1 Property Graph Models . . . . .	4
2.2 Graph Database Management Systems (GDBMSs) . . . . .	5
2.3 Schema and Constraints in Databases . . . . .	6
2.4 Cypher Query Language . . . . .	9
2.5 Validation Strategies . . . . .	9
<b>3 Related Work</b>	<b>11</b>
3.1 Schema Languages for Property Graphs . . . . .	11
3.2 Techniques for Schema Validation in Graph Databases . . . . .	12
3.3 Validation and Performance Trade-offs . . . . .	13
<b>4 Methodology</b>	<b>14</b>
4.1 Research Design . . . . .	14
4.2 Identifying Schema Constraints . . . . .	14
4.3 Representing Schema Constraints as Cypher Queries . . . . .	15
4.4 Experimental Design . . . . .	16
4.5 Evaluation Setup . . . . .	22
<b>5 Results</b>	<b>24</b>
5.1 Identified Schema Constraints . . . . .	24
5.2 Mapping Schema Constraints to Cypher Queries . . . . .	27
5.3 Experiment Results . . . . .	28
<b>6 Discussion</b>	<b>37</b>
6.1 Discussion of Results . . . . .	37

6.2	Discussion of Method . . . . .	40
6.3	Limitations of the Query-Driven Validation Strategy . . . . .	42
6.4	The Work in a Wider Context . . . . .	44
<b>7</b>	<b>Conclusion</b>	<b>46</b>
7.1	Answers to the Research Questions . . . . .	46
7.2	Implications and Contributions . . . . .	47
7.3	Future Work . . . . .	47
	<b>Bibliography</b>	<b>49</b>
<b>A</b>	<b>Constraint Examples</b>	<b>51</b>

# List of Figures

2.1	Example property graph with two nodes with label <code>Person</code> (Alice and Bob) connected by an edge with label <code>KNOWS</code> . . . . .	5
4.1	Comparison of workflows for pre-validation and query-driven validation. . . . .	20
5.1	Total execution times for pre-validation and query-driven validation. . . . .	34
5.2	Stacked chart of query-driven validation execution time by stage. . . . .	35
5.3	Ingestion time throughput for query-driven validation per dataset. . . . .	36

# List of Tables

4.1	Data model for the dataset. . . . .	16
4.2	Node and outgoing edge label distribution for the mock domain graph. . . . .	17
4.3	In-degree and out-degree ranges for each node label in the mock domain graph. . . . .	17
4.4	Labels and counts of the nodes and edges introduced by valid and invalid instances of the constraints. . . . .	18
4.5	Synthetic dataset configurations used in the evaluation. Node and edge counts exclude those introduced by valid and invalid constraint instances. . . . .	19
4.6	Violation distribution across scale factors (25 baseline constraint violations + 1% random). Total graph elements counts nodes and edges combined, excluding those added by valid and invalid constraint instances. . . . .	19
5.1	Schema catalog of property graph constraint types. . . . .	26
5.2	Mapping of graph schema constraint types with corresponding violation conditions and Cypher queries for detection. . . . .	33
5.3	Overall execution time comparison between strategies. . . . .	34
5.4	Breakdown of query-driven validation execution time by stage. . . . .	35



# 1 Introduction

This chapter introduces the research topic, outlining the motivation for studying schema validation in property graph databases, the importance of the problem in practice, how it is studied and the scientific novelty of approaching validation as an in-database task.

## 1.1 Motivation

*Graph database management systems (GDBMSs)* based on the *property graph model* have gained popularity due to their ability to model complex and interconnected data. However, unlike relational database management systems, which enforce schemas by design, many GDBMSs such as *Neo4j* allow property graphs to be managed without a predefined schema. This *schema-optional behaviour* offers flexibility when integrating heterogeneous data but also shifts the responsibility for ensuring *schema compliance* (i.e., adherence to expected graph structures) onto the user. In this context, a schema describes the expected structure of the graph, including which node and edge types are allowed and how they may be connected. In contrast, systems such as *TigerGraph* enforce schemas, illustrating that schema support is a design choice rather than an inherent property of graph databases.

Neo4j has recently introduced basic schema constraints (e.g., *uniqueness*, *existence*), but these cover only a limited subset of what a property graph schema typically specifies. For example, they cannot express rules about allowed *label combinations*, *mandatory relationship types* or *edge cardinalities* (i.e., limits on how many edges a node can have). Ensuring compliance with richer schema definitions therefore requires additional mechanisms.

This raises the problem of how to efficiently ensure schema-compliant data in large-scale GDBMSs such as Neo4j. Current practice often validates data before insertion, but this approach can become a bottleneck when handling heterogeneous sources and high volumes. An alternative - importing data first and validating it inside the graph - remains underexplored both in research and practice.

This limitation is also evident in real-world data integration pipelines. For example, Entiros, an enterprise integration platform provider, manages large volumes of heterogeneous data from multiple sources. In their current pipeline, data is validated against expected structural assumptions prior to ingestion into a Neo4j-based graph store. As data volume and heterogeneity increased, this pre-validation step became both a performance bottleneck and

a source of operational complexity, directly motivating the investigation of alternative validation strategies explored in this thesis.

The correctness of graph data is critical for downstream applications such as recommendation systems, knowledge graphs and enterprise integration platforms. At the same time, validation must balance reliability with performance: strict *pre-validation* can slow down pipelines, while naive *post-validation* risks allowing invalid data to propagate. The scientific novelty of this thesis lies in reframing validation as an *in-database* problem rather than a preprocessing step. Specifically, it explores representing schema constraints as *Cypher queries* - Cypher being Neo4j's SQL-like query language - that identify violations directly within the graph, analogous to how SQL queries can detect constraint violations in relational databases. This query-driven approach, in combination with existing mechanisms such as native constraints and schema-aware import strategies, will be evaluated for its scalability and effectiveness.

## 1.2 Aim

The aim of this thesis is to investigate whether in-database validation can improve scalability and maintainability of graph data pipelines while ensuring that production graphs remain schema-compliant.

## 1.3 Research Questions

1. What are the typical elements (e.g., types of constraints) that schemas for property graphs may consist of?
2. How can these schema constraints be represented as Cypher queries that identify violations in Neo4j?
3. How do different validation strategies - query-driven and pre-validation - compare in terms of:
  - a) *Throughput*: how many data items can be imported and validated per second?
  - b) *Processing time*: how long it takes from data arrival until guaranteed schema compliance is achieved?

## 1.4 Approach

For the purpose of answering the research questions, multiple research methods are used.

A *prototype tool* is brought forward which has the purpose of validating the data after it has been imported and is integrated in *Entiros' pipeline*. This type of system is expected to provide higher accuracy and improved performance compared to the legacy ETL (extract, transform, load) system.

## 1.5 Contributions

The main contributions of this thesis are:

- A *schema catalog* that identifies and classifies typical constraints for property graphs.
- A *reusable library of Cypher templates* that implement these constraints as in-database validation rules.
- An empirical evaluation comparing query-driven validation with pre-validation approaches in terms of scalability and effectiveness.

## 1.6 Delimitations

This work focuses exclusively on Neo4j as the target graph database management system and on Cypher as the query language for expressing validation logic. While the proposed query-driven validation approach is conceptually applicable to other property graph systems, no attempt is made to generalize or empirically evaluate the approach across multiple GDBMSs or query languages. Consequently, conclusions regarding performance and applicability should be interpreted within the context of Neo4j.

Furthermore, this thesis does not aim to implement or support full formal schema languages such as PG-Schema, nor to provide complete schema management functionality. Instead, the focus is on evaluating practical mechanisms for detecting schema violations using executable queries within an existing graph database.

The evaluation emphasizes the correctness and performance of schema violation detection rather than end-to-end operational guarantees. In particular, the prototype identifies violating nodes and edges but does not enforce corrective actions or prevent downstream consumers from observing intermediate invalid states. Designing and evaluating automated remediation strategies, transactional isolation guarantees or operational safeguards for production pipelines is considered outside the scope of this work.



## 2 Background

This section provides the theoretical foundation for the thesis. It introduces key concepts from graph data management that are essential for understanding the challenges of schema enforcement in *property graphs*. The discussion covers how schemas and constraints function in different database paradigms, how property graphs and systems such as Neo4j handle them in practice and what strategies exist for validating schema compliance. These foundations will later serve as the basis for the experimental design and analysis.

### 2.1 Property Graph Models

#### 2.1.1 Definition

A property graph is a directed graph that can be enriched with labels and properties [1]. It consists of a set of *nodes* and a set of *edges*<sup>1</sup>, where each edge connects exactly two nodes: one as its source and one as its target. Both nodes and edges can carry one or more *labels*, drawn from a set of label names (typically strings). Labels are optional and may be combined to classify an element into multiple categories.

In addition, nodes and edges may have *properties*, expressed as key–value pairs. Property keys are strings and must be unique within a given node or edge. Property values are typed, usually drawn from a set of atomic types such as integers, floating-point numbers, strings, booleans, date and time values or even arrays in some systems. These constraints ensure that each property name occurs at most once per node or edge and that values have a well-defined type.

Figure 2.1 shows a simple property graph with two nodes, each carrying the label `Person` and a property `name` with values “Alice” and “Bob”. The nodes are connected by a directed edge labeled `KNOWS`, which also has a property `since` with value 2020.

---

<sup>1</sup>Neo4j uses the term “relationship” for edges. In this thesis, “edge” is used interchangeably with “relationship” for clarity.

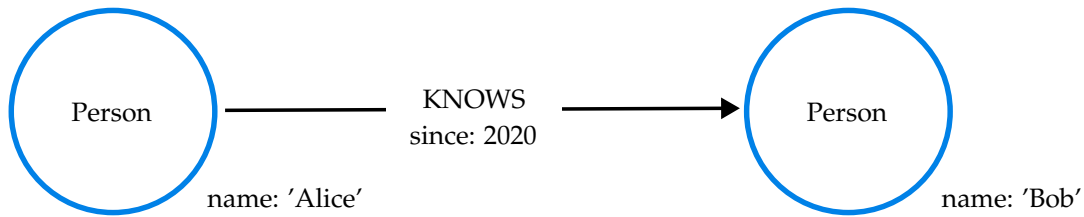


Figure 2.1: Example property graph with two nodes with label `Person` (Alice and Bob) connected by an edge with label `KNOWS`.

### 2.1.2 Difference from Relational Databases

Unlike relational databases, where schemas are enforced by design, property graphs are schema-optional, meaning nodes and edges can be added without predefined structure<sup>2</sup>. This flexibility is useful in exploratory or rapidly changing domains, but it raises challenges for maintaining consistency at scale. This design difference highlights why property graph databases require custom validation strategies, a focus of this thesis, as schema compliance cannot be assumed by default.

### 2.1.3 Difference from RDF Graphs

Resource Description Framework (RDF) is another popular graph data model [2], but it differs fundamentally from property graphs. In RDF, data is represented as triples (`subject`, `predicate`, `object`), where each triple encodes a single fact. Formally, a triple consists of a node for the subject, an edge (arc) representing the predicate and a node for the object, with each element identified by a Uniform Resource Identifier (URI). Unlike property graphs, RDF does not attach arbitrary properties directly to nodes or edges and instead express additional information through more triples.

While property graphs rely on labels and properties with optional constraints, RDF uses ontologies (formal specifications of classes, edges and allowed values), such as the RDF Schema (RDFS) or Web Ontology Language (OWL). RDF emphasizes semantic rigor and standardized validation, highlighting the expressive flexibility of property graphs and the need for custom validation strategies, which this thesis addresses through query-driven approaches.

### 2.1.4 Why Property Graphs

The combination of flexible structure and lack of built-in schema enforcement makes property graphs attractive for applications such as recommendation systems, knowledge graphs and enterprise integration platforms, but also creates the central challenge of this thesis: how to ensure schema compliance efficiently in large-scale Neo4j databases.

## 2.2 Graph Database Management Systems (GDBMSs)

### 2.2.1 Overview of GDBMSs

As opposed to property graph databases, which are collections of related data, a graph database management system (GDBMS) is the software system that implements and provides tools to create, manage and query this graph data model [3]. Examples of popular GDBMSs are Neo4j, TigerGraph and JanusGraph.

<sup>2</sup><https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/graphdb-vs-rdbms/>, accessed: 2025-09-09

### 2.2.2 APOC Library

*Awesome Procedures On Cypher (APOC)* is a community-maintained library that has become a de facto extension of Neo4j<sup>3</sup>. It provides hundreds of additional procedures and functions, many of which are highly relevant for validation. For instance, `apoc.validate()` allows rule-based checks with custom error handling, `apoc.trigger` can enforce business rules on data modifications and `apoc.periodic.iterate` supports scalable batch validation on large datasets<sup>4</sup>. In this thesis, APOC plays a complementary role: where Cypher alone is insufficient for expressing complex or performance-sensitive checks, APOC provides the procedural mechanisms to implement them.

## 2.3 Schema and Constraints in Databases

### 2.3.1 Schemas in Relational Databases

To understand schema enforcement in property graphs, it is useful to first contrast it with relational databases. In a relational database, a schema defines the structure of the database by specifying the relations (tables), their attributes (columns) and the types of values each attribute can hold. It also specifies constraints, such as keys or domain restrictions, which act as a blueprint to ensure data integrity and consistency when data is inserted, updated or deleted.

Constraints in relational databases are generally grouped into three categories, all of which depend on the relational data model<sup>5</sup>:

- **Implicit constraints**, which are derived directly from the data model. For example, in a relational database, every tuple in a relation must have exactly one value for each attribute of that relation. Additionally, tuples cannot contain values for attributes that are not part of the relation's schema.
- **Explicit constraints**, which are declared at the schema level. Common examples include:
  - *Uniqueness constraints*, such as primary keys ensuring no two tuples share the same identifier.
  - *Referential integrity constraints*, such as foreign keys that ensure referenced tuples exist. A foreign key is an attribute in one table that references a primary key in another table, ensuring that relationships between tables remain consistent.
  - *Domain constraints*, specifying allowable values or types for attributes.
- **Application-based constraints**, which are enforced by application logic rather than the schema. For example, ensuring that an employee's salary is never decreased may not be enforced by the schema but by business logic in the application layer.

These three categories illustrate the different ways in which constraints can be enforced in a relational database, highlighting the complementary roles of the schema and application logic in maintaining correctness and consistency.

<sup>3</sup><https://neo4j.com/labs/apoc/>, accessed: 2025-09-10

<sup>4</sup><https://neo4j.com/docs/apoc/current/overview/>, accessed: 2025-09-10

<sup>5</sup><https://www.geeksforgeeks.org/dbms/constraints-on-relational-database-model/>, accessed: 2025-09-10

### 2.3.2 Schemas in GDBMSs

Property graphs are typically schema-optional, though the degree of enforcement varies across GDBMSs<sup>6,7</sup>. For instance, Neo4j allows property graphs to be used without a pre-defined schema, whereas TigerGraph requires a schema to be defined and enforces it strictly.

However, even in flexible GDBMSs like Neo4j, lightweight schemas offer benefits similar to those in relational databases, including faster query traversal, improved data integrity and better support for collaboration and maintenance among developers and administrators [4]<sup>8</sup>. These benefits are typically enforced through constraints which can also be categorized similarly to relational databases:

- **Implicit constraints**, which arise from the graph data model itself, such as the requirement that edges must connect two nodes.
- **Explicit constraints**, which are declared in the graph schema where supported. Examples include uniqueness of node identifiers and existence of required properties.
- **Application-based constraints**, which are enforced outside the schema through application logic, for instance, ensuring a node's property value satisfies a business rule.

**Neo4j** Many property graph systems, however, support only a subset of explicit constraints natively. Neo4j, one of the most widely adopted property graph database management systems, uses the declarative query language Cypher to manipulate and query property graphs<sup>9</sup>. Historically schema-less, Neo4j gave developers modeling flexibility but required application logic to enforce data quality. In recent times, Neo4j introduced limited schema support in the form of constraints, thereby moving toward schema-awareness without abandoning its core principle of flexibility.

Currently, Neo4j's Enterprise Edition supports several constraint types<sup>10</sup>:

1. Property uniqueness constraints, which require that the value of a single property or a specified combination of properties is unique for all nodes or edges with a specific label.

For example: `CREATE CONSTRAINT FOR (p:Person) REQUIRE p.ssn IS UNIQUE` ensures that no two `Person` nodes share the same `ssn` value.

2. Property existence constraints, which require that a specific property exists for all nodes or edges with a specific label.

For example: `CREATE CONSTRAINT FOR (p:Person) REQUIRE p.name IS NOT NULL` ensures that every `Person` node has a `name` property.

3. Property type constraints, which require that a specific property has the declared type for all nodes or edges with a specific label.

For example: `CREATE CONSTRAINT FOR (p:Person) REQUIRE p.age IS INTEGER`

4. Key constraints enforce two conditions. First, a specified combination of properties, called a key, must exist on every node or edge with a given label. Second, the combination of property values must be unique across those nodes or edges.

For example: `CREATE CONSTRAINT FOR (p:Person) REQUIRE (p.firstName, p.lastName) IS NODE KEY`

<sup>6</sup><https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/#graphdb-schema>, accessed: 2025-09-09

<sup>7</sup><https://docs.tigergraph.com/gsql-ref/4.2/ddl-and-loading/defining-a-graph-schema>, accessed: 2025-09-09

<sup>8</sup><https://www.byteplus.com/en/topic/403100?title=common-mistakes-with-graph-databases-a-comprehensive-guide>, accessed: 2025-09-10

<sup>9</sup><https://neo4j.com/product/>, accessed: 2025-09-10

<sup>10</sup><https://neo4j.com/docs/cypher-manual/current/constraints/>, accessed: 2025-09-09

While these constraints provide some level of schema enforcement, they are limited in scope. For example, Neo4j cannot natively enforce edge cardinalities, conditional rules or dependencies across different node types. This limitation motivates the use of query-driven strategies, where Cypher queries explicitly check for violations that built-in constraints cannot capture. This gap forms the basis for the schema constraint catalog defined in RQ1 and explored further in RQ2.

### 2.3.3 Schema Languages

A schema language provides a formal way to specify the structure and constraints of a database. For relational databases, this role is fulfilled by SQL Data Definition Language (DDL), which can be used to define tables, attributes, keys and integrity rules.

For RDF graphs, schema specification and validation is typically done using ontologies, such as RDFS or OWL, together with the Shapes Constraint Language (SHACL)<sup>11</sup>. SHACL allows the declarative definition of *shapes*, which specify constraints on target nodes, including required properties, data types, cardinality and edges. Validation produces a report of any violations, ensuring that RDF data conforms to its intended structure.

For instance, consider a SHACL shape that defines constraints for `Person` nodes:

```
ex:PersonShape
  a sh:NodeShape ;
  sh:targetClass ex:Person ;
  sh:property [
    sh:path ex:name ;
    sh:datatype xsd:string ;
    sh:minCount 1 ;
  ] ;
  sh:property [
    sh:path ex:email ;
    sh:datatype xsd:string ;
    sh:pattern "^.+@.+\\.+.$" ;
  ] .
```

The shape declares that it targets all resources of class `ex:Person` (`sh:targetClass ex:Person`). It then specifies two property constraints:

- The `name` property must be present at least once (`sh:minCount 1`) and must be a string (`sh:datatype xsd:string`).
- The `email` property, if present, must also be a string and additionally conform to a regular-expression pattern that approximates the structure of an email address (`sh:pattern "^.+@.+\\.+.$"`).

For property graphs, no single standardized schema language exists [5][6][7]. Different proposals introduce schema languages that define which node labels, edge labels and properties are permitted, along with constraints such as uniqueness, existence, cardinality and type restrictions.

Schema languages serve as formal mechanisms to express the structure and constraints of a database, enabling the benefits of schemas to be realized. Specifically, they allow for improved data quality through explicit validation rules and better communication of the intended data model across developers, tools and systems. These ideas are particularly relevant for this thesis, since the schema constraint catalog in RQ1 builds on the types of constraints commonly defined in such schema languages.

<sup>11</sup><https://www.w3.org/TR/shacl/>, accessed: 2025-09-30

## 2.4 Cypher Query Language

Cypher is Neo4j's declarative query language for property graphs. Similar in spirit to SQL for relational databases, Cypher enables expressive querying and manipulation of graph data through pattern matching rather than imperative traversal instructions<sup>12</sup>. Queries describe the shape of nodes, edges and properties to be retrieved or filtered and the database engine determines how to execute them efficiently.

It is designed to resemble ASCII art, making patterns in the graph readable and intuitive. Nodes are written inside parentheses and edges inside square brackets, with arrows indicating their direction. Labels and properties can be specified inline. To illustrate how nodes, edges, labels and properties are represented in queries, the property graph from Figure 2.1 can be expressed in Cypher as: `(a:Person {name: "Alice"})-[:KNOWS {since: 2020}]->(b:Person {name: "Bob"})`. Here, `a` and `b` are identifiers for the nodes which are used to identify them later in queries. `Person` is a label assigned to both nodes, `name` is a property stored on each node, `KNOWS` is the label assigned to the edge connecting Alice to Bob and `since` is a property on the edge, indicating the year it was established.

Beyond this basic example, Cypher supports a range of features for querying and constraining graphs. For the purposes of this thesis, only the core features of Cypher relevant to schema validation are introduced:

1. **Pattern matching:** In Cypher, the `MATCH` keyword is used to specify graph patterns to be searched for in the database. Graph patterns are written using parentheses for nodes and arrows for edges, optionally including labels and properties. For example, `MATCH (p:Person)-[:KNOWS]->(q:Person)` matches all subgraphs where a node with label `Person` is connected via an edge with label `KNOWS` to another node with label `Person`. Here, `(p:Person)` and `(q:Person)` are *node patterns* that match nodes with the `Person` label and `[:KNOWS]` is an *edge pattern* that matches edges with the `KNOWS` label.
2. **Filtering:** The `WHERE` clause restricts matches to conditions, e.g., `MATCH (p:Person) WHERE NOT exists(p.name) RETURN p` finds all nodes with label `Person` without a `name` property.
3. **Aggregation:** Aggregating functions compute a single value from a set of records. For example, `COUNT` returns the number of matches and `DISTINCT` filters out duplicates. These functions help validate constraints such as uniqueness (e.g., no two nodes share the same identifier) or cardinality (e.g., a node has at most one outgoing edge of a given type).
4. **Constraints:** Neo4j supports a subset of schema constraints through Cypher's `CREATE CONSTRAINT` statement. For details and examples of the supported constraint types, see Section 2.3.2.
5. **Return values:** Queries return bindings of matched nodes, edges or computed values.

Cypher's declarative nature makes it well suited for validation: constraints can be formalized as query patterns whose results indicate non-compliant states. Where native constraints are insufficient, Cypher queries provide a flexible and uniform mechanism for expressing validation rules.

## 2.5 Validation Strategies

When integrating external data into a database, one of the core challenges is ensuring that new records do not violate the intended schema. Different validation strategies exist, each

<sup>12</sup><https://neo4j.com/docs/getting-started/cypher/>, accessed: 2025-09-19

with trade-offs in terms of performance, coverage and maintainability. These strategies can be broadly grouped as pre-validation and post-validation strategies.

### 2.5.1 Pre-validation

In pre-validation, data is validated before being inserted into the database. This is the approach typically used in Extract–Transform–Load (ETL) pipelines and traditional relational databases, where schema-based constraints (such as domain or key constraints) can be enforced automatically by the system [8]. Pre-validation prevents invalid data from ever entering the database, thereby ensuring that all persisted data is compliant by construction. The drawback is that it may slow down data ingestion, since each data item must be checked before it can be stored. Moreover, in highly heterogeneous or rapidly evolving settings, the need to predefine and enforce a rigid schema can reduce flexibility and hinder adaptability. This strategy reflects Entiros’ current pipeline and serves as the baseline for RQ3, where it will be compared to query-driven strategies.

### 2.5.2 Post-validation

Post-validation (also referred to as “insert-first-then-check”) takes the opposite approach: data is first loaded into the database, after which compliance with the schema is verified [7]. This strategy is common in scenarios where fast ingestion is prioritized or where the schema is flexible or evolving, such as in data lakes and graph databases. Violations are detected after the fact, often through query-based checks or batch validation processes. While this approach supports high-throughput ingestion, it requires additional mechanisms for isolating, repairing or rejecting invalid data once violations are discovered. Post-validation is particularly well-suited for property graph database management systems such as Neo4j, since their schema-optional nature and expressive query languages make it easier to check complex rules after ingestion than to enforce them upfront. This aligns with the query-driven strategy investigated in RQ2 and will be directly evaluated against Entiros’ pre-validation method in RQ3.

### 2.5.3 Implications for Property Graph Databases

Most property graph database management systems, including Neo4j, provide only limited built-in support for schema-based constraints. While uniqueness and existence constraints can be specified, many important schema constraints (such as cardinality rules or mandatory edges) are not natively enforceable. As a result, validation in property graphs may rely on query-driven post-validation strategies, where schema rules are expressed as Cypher queries that identify violations after data has been loaded [9]. In this thesis, these three strategies are not only discussed conceptually but also compared experimentally in the context of Neo4j. Specifically, RQ3 investigates how query-driven post-validation and Entiros’ existing pre-validation pipeline differ in terms of schema coverage, throughput and processing time.



## 3 Related Work

Research on schema validation in property graphs has followed three main directions. First, several schema languages have been proposed to formally capture constraints such as uniqueness, existence and cardinality. Second, practical techniques have been developed to validate schema compliance, ranging from theoretical formulations to query-driven implementations. Third, a line of work has examined the performance implications of validation, highlighting the trade-offs between native enforcement and external pre-validation. Together, these studies form the background against which this thesis positions its contributions.

### 3.1 Schema Languages for Property Graphs

Previous research within the field has gone into defining schema languages for property graphs. These works converge on identifying key schema constraints such as uniqueness, existence and cardinality, which informs the schema catalog developed to address RQ1.

Hartig and Hidders [6] repurpose GraphQL Schema Definition Language (SDL) to describe property graph schemas. SDL was originally designed to specify queryable object types in GraphQL APIs, but here it captures node types, edge types and constraints such as mandatory fields and type restrictions. GraphQL SDL supports constraints such as mandatory fields, type restrictions and demonstrates how schema constraints like existence and type constraints can be captured declaratively - elements that also appear in this thesis' schema catalog.

Angles et al. [5] introduce a dedicated, expressive schema language for property graphs. Unlike the GraphQL mapping, PG-Schema is designed from the ground up for property graphs and supports advanced features like type hierarchies, mandatory properties, cardinality constraints and integrity rules. For example, the following schema defines two node types (Person and Customer) and a friendship relation, together with key and foreign key constraints:

```
CREATE GRAPH TYPE socialGraphType STRICT {
  (p:Person {id INT, name STRING}),
  (c:Customer {id INT}),
  (p)-[friend:Knows]->(p),
  FOR (x:Person) EXCLUSIVE MANDATORY SINGLETON x.id,
```

```
FOR (x:Customer) MANDATORY y.id WITHIN (y:Person) WHERE y.id = x.id
}
```

Here, the first constraint (`FOR (x:Person) EXCLUSIVE MANDATORY SINGLETON x.id`) specifies that `id` uniquely identifies each `Person` node. It acts like a key in the sense that no two `Person` nodes can share the same `id`. The second constraint (`FOR (x:Customer) MANDATORY y.id WITHIN (y:Person) WHERE y.id = x.id`) ensures that every `Customer` node is associated with an existing `Person` node with the same `id`. In other words, the `Customer.id` property must match a valid `Person.id`, similar to a foreign key in relational databases. This example illustrates how PG-Schema can capture constraints beyond the limited uniqueness checks available in current graph databases.

Angles et al. formalize the semantics of schema compliance and provide both the theoretical foundation and practical constructs for rigorous schema validation which makes it highly relevant for this thesis. The practical contribution consists of concrete definitions of constraint types, formal validity rules that can be checked against a graph instance and translation mechanisms that show how such constraints can be expressed and verified in practice.

While GraphQL SDL and PG-Schema define expressive languages for specifying property graph constraints, they do not map these schema constraints to executable Cypher queries or provide reusable implementations. They also lack performance benchmarks on large graphs and integration with real-world ETL pipelines. This thesis addresses these gaps by implementing a query-driven validation strategy, evaluating its efficiency and coverage on synthetic and production datasets and integrating it into an industrial Neo4j pipeline.

## 3.2 Techniques for Schema Validation in Graph Databases

Beyond defining schema languages, prior research has investigated how schema rules can be checked in practice within property graph DBMSs. These techniques differ in how validation is operationalized, ranging from theoretical formulations to query-based implementations.

Building on PG-Schema’s expressive formalism (see Section 3.1), Angles et al. also define algorithms for validating compliance. Validation is defined in terms of schema compliance and the paper formalizes how the aforementioned constraints can be verified. This work demonstrates that schema validation can be rigorously grounded in theory, though its practical implementation is still limited to research prototypes.

Bonifati et al. [7] further formalize schema validation through graph homomorphisms, framing validation as a structural correspondence between schema and data graphs. This perspective captures both structural constraints (nodes, edges) and semantic constraints (properties, multiplicities) and lays the foundation for automated reasoning about schema compliance and evolution. However, their approach emphasizes theoretical soundness over operational tools for practitioners.

In the RDF ecosystem, SHACL is a W3C standard for constraint validation. Recent work, such as Figuera et al. [9], explore optimization strategies for large-scale validation, while Corman et al. [10] formalize recursive constraints and highlight the trade-off between expressiveness and computational cost. These studies underscore the practical relevance and complexity of constraint validation, offering insights applicable to property graph settings.

Taken together, these works show that schema validation in graph databases can be achieved through formal compliance definitions and homomorphism-based reasoning. However, none of them provide a practical, reusable library of query templates or performance evaluations on large datasets. Furthermore, few implementations exist beyond research prototypes. This thesis addresses this gap by implementing Cypher-based validation strategies and empirically evaluating their effectiveness in industrial pipelines.

### 3.3 Validation and Performance Trade-offs

Schema validation in graph data management is not only a matter of expressivity but also of performance. Different strategies distribute the cost of validation in different stages of the data pipeline and the literature highlights distinct trade-offs between native constraints, external pre-validation and benchmarking studies.

#### 3.3.1 Native Database Constraints

While Section 2.3.2 introduce the available constraints, Šestak et al. [11] examine the expressiveness of such mechanisms and show that enforcing more complex constraints, such as  $k$ -vertex cardinalities, requires custom query-based encodings with non-negligible overhead. Skavantzios et al. [12] similarly analyze the cost of supporting advanced uniqueness constraints in property graphs. Together, these works suggest that, while native constraints can be applied efficiently at insertion time, they are inherently limited in scope, necessitating complementary validation strategies to achieve full schema compliance.

#### 3.3.2 External Pre-validation

A different approach is to validate data before it enters the graph store, analogous to ETL workflows in relational systems. The SHACL community has developed some of the most comprehensive work in this space. For example, Figuera et al. [9] benchmark different SHACL validator implementations and introduces optimization techniques such as constraint reordering and incremental validation, showing measurable runtime reductions. Although these studies are situated in the RDF ecosystem, they illustrate how systematic benchmarking can reveal the trade-offs of pre-validation: correctness is ensured at import time, but throughput can suffer significantly for large datasets.

#### 3.3.3 Benchmarking and Performance Studies

The evaluation of validation strategies draws on broader benchmarking efforts in the graph systems community. Erling et al. [13] propose the LDBC Social Network Benchmark, which defines metrics such as throughput, latency and scalability for complex graph workloads. While not targeting schema validation, LDBC establishes an experimental methodology based on controlled workload definitions. Similarly, Rusu et al. [14] show that Neo4j's performance in bulk loading and query execution degraded under certain workloads when compared with other graph systems. Together, these studies illustrate how systematic workload design and controlled experiments can reveal performance trade-offs, a methodological perspective directly relevant when assessing the cost of schema validation. To our knowledge, however, no prior benchmark has systematically measured the overhead of enforcing full schema compliance (all relevant constraint types) in property graphs across multiple validation strategies.



## 4 Methodology

This section describes the methodology used to investigate schema validation strategies in property graph databases. The approach is structured around the three research questions: (RQ1) defining a catalog of schema constraints relevant for property graphs, (RQ2) implementing validation mechanisms in Neo4j and (RQ3) evaluating their effectiveness and performance against Entiros' existing pipeline.

### 4.1 Research Design

The study follows a design–implement–evaluate approach. First, a catalog of schema constraints was constructed (RQ1). Second, validation strategies were implemented in Neo4j (RQ2). Finally, the strategies were benchmarked against an industrial pre-validation pipeline provided by Entiros (RQ3).

### 4.2 Identifying Schema Constraints

To address RQ1, a systematic process was used to identify the set of schema constraints relevant to property graphs.

1. **Literature synthesis:** A review of existing schema languages for property graphs was conducted, including PG-Schema, GraphQL SDL and prior work on schema validation and evolution. From these sources, recurring schema constraints such as uniqueness, existence, type and cardinality were extracted. This literature-driven synthesis approach is inspired by prior work on property graph schema design and validation [7][5][6].
2. **Pipeline validation:** The extracted schema constraints were compared against the validation requirements of Entiros' industrial pipeline. In a dedicated meeting with Entiros representatives, the catalog was reviewed and constraints that were not relevant to the pipeline (e.g., edge property uniqueness, since edges in their data model do not have properties) were marked as not required (-) in the catalog, while those required for validation were marked with a checkmark (✓).

3. **Catalog consolidation:** The resulting schema constraints were consolidated into a schema catalog, documenting for each constraint type its definition, references from prior work, relevance for the Entiros pipeline and how the constraint is currently implemented by Entiros. In addition, a small number of constraints were added based directly on Entiros' validation requirements, as these were not covered in the surveyed literature but are critical in their industrial setting.

This catalog provides the foundation for the subsequent implementation and evaluation steps (RQ2 and RQ3). The chosen dimensions ensure both theoretical grounding (literature references) and industrial relevance (Entiros requirements).

### 4.3 Representing Schema Constraints as Cypher Queries

To address RQ2, schema constraints from the catalog (Section 4.2) were systematically translated into Cypher query patterns for validation in Neo4j. This translation followed a structured procedure inspired by prior work on property graph schema validation and constraint enforcement [7, 5, 6], as well as established practices in relational databases, where constraints can be expressed as SQL queries (e.g., uniqueness checks or referential integrity constraints). In the graph context, Bonifati et al. [7] and Angles et al. [5] formalize schema compliance in terms of query-based verification and Neo4j documentation recommends Cypher queries or APOC procedures for constraints not natively supported.

The mapping procedure consisted of five conceptual steps:

1. **Violation condition:** For each applicable schema constraint<sup>1</sup> in the schema catalog (see Table 5.1), a natural-language description was created, specifying when the schema rule is violated (e.g., "a `Person` node with no `name` property").
2. **Test graph construction:** Two small graph instances were created in Neo4j: one representing a valid structure and one representing an invalid structure according to the schema constraint. These instances explicitly encode the presence or absence of required nodes, edges, labels or properties.
3. **Violation detection query:** A concrete Cypher `MATCH` query was written for each schema constraint to retrieve only the invalid graph elements (nodes or edges), while excluding the corresponding valid graph elements. These queries were evaluated against the constructed test graphs to ensure that they correctly discriminated between compliant and non-compliant data.
4. **Query generalisation:** Each violation detection query was then generalised into a reusable, parameterized Cypher query template by replacing schema-specific labels and property names with placeholders (e.g., `%label%`, `%prop%`). Some templates include a brief inline comment (e.g., `// m=2`) to specify the particular configuration shown in that example.
5. **Mapping consolidation:** The results of the mapping were consolidated into a query mapping table, documenting for each constraint type its violation condition, the corresponding parameterised Cypher query template and, where illustrative, an example instantiation with concrete labels and properties.

This procedure ensures a systematic and reproducible translation from abstract schema constraints to executable validation templates, forming the foundation for a reusable library of Cypher patterns applicable across different datasets and pipelines. The complete query mapping is presented in Table 5.2, together with selected example instantiations.

<sup>1</sup>The open-world property set constraint is excluded, as no well-defined violation condition can be specified for it.

## 4.4 Experimental Design

To evaluate the proposed schema validation mechanisms and to address RQ3, a controlled experimental design is required. The objective is to generate datasets and workloads that systematically cover the identified schema constraints (see Section 4.2), while also allowing performance and correctness to be assessed under varying levels of data scale. This section outlines the dataset construction, scaling methodology, query workloads and measurement procedure.

### 4.4.1 Dataset Construction

#### Data Model and Construction

A synthetic dataset will be created to encode all identified schema constraints. The use of a synthetic rather than real dataset enables precise control over structural and semantic properties, ensuring that each type of constraint is represented. This approach follows prior work in graph data management where synthetic datasets are preferred to guarantee coverage of desired patterns and constraints [7].

The dataset construction begins with the creation of a *mock domain graph*, which is instantiated programmatically in Neo4j using parameterised Cypher `CREATE` and batched `UNWIND` operations that deterministically generate nodes, edges and properties according to the data model in Table 4.1. This data model is a simplified representation of the data model of a product developed by Entiros. The term “mock domain graph” originates from that context, where a `Domain` represents the top-level data object within which users operate. Each `Domain` serves as a logical workspace or container for related network components such as systems, services and endpoints.

Type	Label	From	To	Properties
Node	Domain	–	–	id, name, created, updated, externalId
Node	Network	–	–	id, name, created, updated, externalId
Node	System	–	–	id, name, created, updated, externalId
Node	Subsystem	–	–	id, name, created, updated, externalId
Node	Service	–	–	id, name, created, updated, externalId
Node	Endpoint	–	–	id, name, created, updated, externalId
Node	Reference	–	–	id, name, created, updated, externalId
Edge	CONTAINS	Domain	Network	created, updated
Edge	MODELS	Network	System	created, updated
Edge	PROVIDES_SERVICE	System	Service	created, updated
Edge	PROVIDES_ENDPOINT	Service	Endpoint	created, updated
Edge	INTERACTS_WITH	Service	Endpoint	created, updated
Edge	TO_SERVICE	Reference	Service	created, updated
Edge	TO_ENDPOINT	Reference	Endpoint	created, updated
Edge	CONSUMES	System	Reference	created, updated

Table 4.1: Data model for the dataset.

The mock domain graph is hierarchical: a single node with the `Domain` label acts as the root (parent) and connects to multiple nodes with label `Network`, which in turn connect to nodes with labels `System`, `Service`, `Endpoint` and `Reference`. This structure forms the core of the experimental datasets. Table 4.2 summarizes the intended distribution of node and outgoing edge labels within the mock domain graph. Table 4.3 shows the in-degree and out-degree ranges for each node label, providing an overview of node connectivity and edge density.

Node Label	Count	Outgoing Edge Labels & Count
Domain	1	CONTAINS (10)
Network	10	MODELS (35)
System	35	PROVIDES_SERVICE (30), CONSUMES (5)
Service	30	INTERACTS_WITH (20), PROVIDES_ENDPOINT (20)
Endpoint	20	-
Reference	5	TO_SERVICE (3), TO_ENDPOINT (2)
<b>Total</b>	<b>101</b>	<b>125</b>

Table 4.2: Node and outgoing edge label distribution for the mock domain graph.

Node Label	In-degree (min/max)	Out-degree (min/max)
Domain	0	10
Network	1	3-4
System	1	0-2
Service	1-2	0, 2
Endpoint	2-3	0
Reference	1	1

Table 4.3: In-degree and out-degree ranges for each node label in the mock domain graph.

Generation proceeds in two phases. First, node sets for each label (*Network*, *System*, *Service*, *Endpoint*, *Reference*) are created according to a pre-defined distribution (see Table 4.2). The *Domain* node is pre-created and excluded from this phase because Entiros’ backend ingestion process requires it to exist beforehand; all other nodes are appended as children of the *Domain* node to preserve the hierarchical structure expected by the pipeline.

Second, edges are generated according to the data model (see Table 4.1) and inserted using chunked UNWIND batches to avoid large transactions and reduce driver overhead. Edges of type *CONTAINS* from the pre-existing *Domain* node to *Network* nodes are excluded from this stage, as they are created separately during ingestion. Batch sizes were determined empirically by evaluating several candidate values and selecting the configuration that yielded the shortest ingestion time.

To construct the full experimental dataset, referred to as the *base dataset*, additional nodes and edges are added to represent constraint instances used for validation. Specifically, one valid and one invalid instance are generated for each constraint type defined in the schema catalog (see Table 5.1). Exceptions include compound constraints formed by combinations of simpler constraints, as well as constraints that inherently do not admit invalid instances<sup>2</sup>. The labels of these additional nodes and edges are selected from the previously described data model (Table 4.1) rather than introducing any new labels, with the exception of a new label *Division*. Examples of these valid and invalid instances are listed in Appendix A.

Each invalid instance corresponds to a deliberate constraint violation (for example, a service missing a required property or an outdated timestamp). To better approximate how such violations would manifest in realistic settings, these invalid elements are not created as isolated structures. Instead, they are attached to the mock domain graph by randomly selecting an appropriate parent node (e.g., a random *Network* or *System* node) and establishing a relevant edge. This ensures that all violations remain embedded within the overall graph topology, accurately reflecting how schema inconsistencies would appear in production-like data rather than as detached, synthetic fragments. Table 4.4 presents the labels and counts of the nodes and edges introduced by both the valid and invalid instances of the constraints.

<sup>2</sup>The excluded constraints are: *Node key constraint*, *edge key constraint* and *open-world property set*.

In addition to the invalid instances, valid instances are generated for each applicable constraint. These valid instances may introduce properties or structural patterns that are not otherwise present in the mock domain graph. Their purpose is to ensure that the constraint validation queries precisely distinguish violations from compliant structures. In particular, the presence of valid instances prevents the `MATCH` patterns used for validation from trivially matching all occurrences of a given structure, thereby verifying that the queries correctly identify only genuine constraint violations and do not produce false positives.

Type	Label	Count
Node	Network	5
Node	System	34
Node	Service	31
Node	Endpoint	19
Node	Reference	2
Node	Division	1
Edge	CONTAINS	5
Edge	MODELS	34
Edge	PROVIDES_SERVICE	31
Edge	PROVIDES_ENDPOINT	19
Edge	INTERACTS_WITH	27
Edge	TO_SERVICE	2
Edge	CONSUMES	2
Edge	IS_MEMBER_OF	1

Table 4.4: Labels and counts of the nodes and edges introduced by valid and invalid instances of the constraints.

To ensure reproducibility, all randomized aspects of dataset generation (e.g., assignment of property values or edge targets) will be controlled through a fixed pseudo-random seed. This ensures that each scale factor produces structurally identical datasets across repetitions, eliminating unintended variance while preserving statistical distributions.

### Scaling Methodology

To study scalability, datasets are generated at increasing sizes by varying the number of nodes and edges under a single `Domain` node, rather than creating multiple mock domain graphs. This approach preserves a single top-level `Domain`, while allowing the dataset to grow by adding more `Network`, `System`, `Service`, `Endpoint` and `Reference` nodes beneath it. This methodology is adapted from benchmarking practice in the Linked Data Benchmark Council (LDBC) [15, 13], where a base dataset is repeatedly expanded while preserving the same schema structure and statistical distributions.

In this thesis, the scale factor controls the size of a single mock domain graph by proportionally increasing the number of nodes and edges at each level of the hierarchy. A scale factor of 1 (SF-1) corresponds to the base mock domain graph shown in Table [tab:nodedistribution]. Increasing the scale factor multiplies the number of nodes accordingly, while preserving the same schema structure and relative distributions. For example, a scale factor of 10 (SF-10) results in 100 `Network` nodes instead of 10.

This allows evaluation of both functional correctness (at small scales) and performance behaviour (at larger scales). This thesis evaluates four scale factors (SF-1, SF-10, SF-100 and SF-1000), where SF-1 corresponds to the base dataset with a single `Domain` and its initial set of child nodes. The choice of these scale increments is inspired by the methodology of Rusu and Huang [14]. Table 4.5 summarizes the dataset configurations planned for the experiments.

Scale Factor	# Nodes	# Edges
SF-1	101	125
SF-10	1,001	1,250
SF-100	10,001	12,500
SF-1000	100,001	125,000

Table 4.5: Synthetic dataset configurations used in the evaluation. Node and edge counts exclude those introduced by valid and invalid constraint instances.

In addition to the baseline individual constraint violations that guarantee coverage of all constraint types, a small fixed violation ratio ( $r = 1\%$ ) is applied relative to the total number of graph elements (nodes and edges). As the scale factor increases (i.e., amount of nodes and edges in the dataset increases), the number of random violations grows proportionally, ensuring that larger datasets maintain realistic proportions of invalid data while preserving coverage across all constraint types.

It should be noted that certain constraints<sup>3</sup> were excluded from the set of eligible random violations. This exclusion was necessary because introducing violations for these constraints could lead to conflicts or errors in other constraints, for example, creating duplicates in uniqueness-constrained properties or violating assumptions required by compound constraints.

Table 4.6 summarizes the total number of graph elements (nodes and edges), the number of additional violations introduced by the fixed violation ratio and the total number of constraint violations across the different scale factors.

Scale Factor	Total Graph Elements	1% Violations	# Constraint Violations
SF-1	226	2	27
SF-10	2,251	22	47
SF-100	22,501	225	250
SF-1000	225,001	2250	2,275

Table 4.6: Violation distribution across scale factors (25 baseline constraint violations + 1% random). Total graph elements counts nodes and edges combined, excluding those added by valid and invalid constraint instances.

### Dataset Export

Finally, the generated dataset is exported from Neo4j into a JSON format that matches Entiros' import specification. This export preserves the hierarchical edges between *Network*, *System*, *Service*, *Endpoint* and *Reference* nodes, allowing the dataset to be ingested by the system under test. Only the relevant structure and fields defined by the data model (Table 4.1) are included. Implementation details of the export process are omitted for brevity.

### 4.4.2 Experiment

This section describes the experimental workflow used to evaluate two ingestion and validation strategies: (1) a *query-driven validation* approach and (2) a *pre-validation* approach. The goal of the experiment is to compare the performance and behaviour of the query-driven validation approach with the pre-validation strategy currently used in the reference implementation. The exact workflows, the validation points and the metrics recorded during experiments are stated for reproducibility. Figure 4.1 provides a high-level comparison of the two workflows, highlighting the different validation points and execution order.

<sup>3</sup>The excluded constraint types are: *node property uniqueness*, *edge property uniqueness*, *min incoming edges* and *node path restriction constraint*.

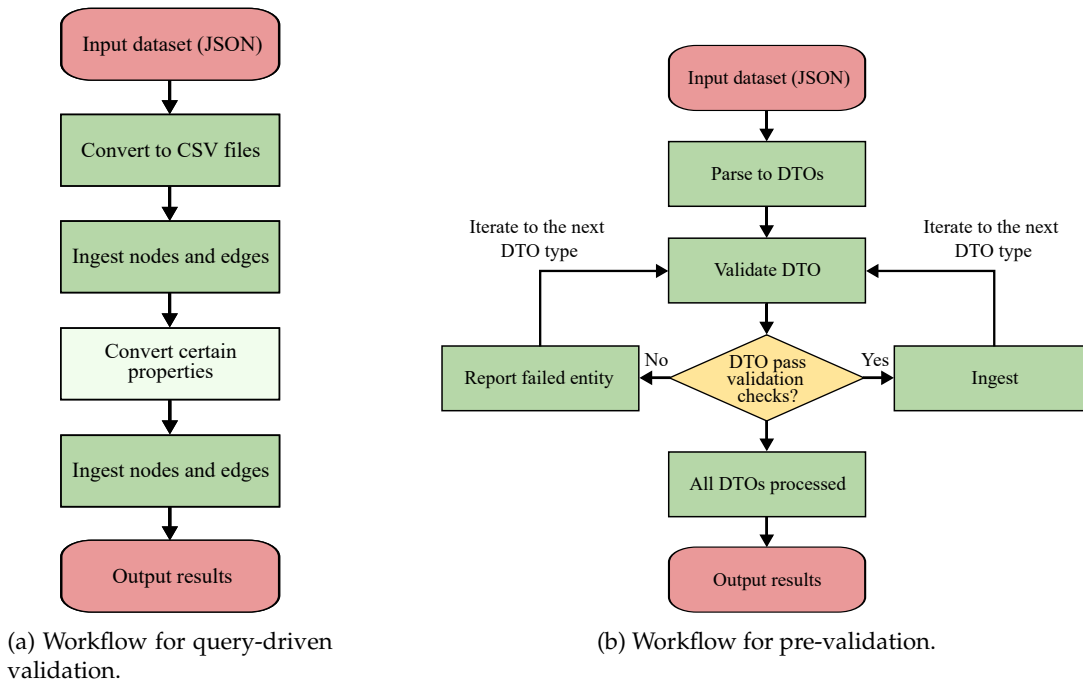


Figure 4.1: Comparison of workflows for pre-validation and query-driven validation.

### Query-driven Validation Workflow

The query-driven approach separates data preparation from validation: input JSON is converted to CSV files, the CSV files are imported into Neo4j and *after* the import completes, a set of Cypher `MATCH` queries are executed to detect violations. This design emphasizes throughput of the ingest path while keeping validation as a post-processing analysis step.

#### Detailed flow

1. **JSON → CSV conversion.** The input JSON graph is transformed to multiple CSV files (each node and edge label getting outputted to one CSV file), using a deterministic naming and column scheme. This conversion is required because Neo4j's bulk import tool (`LOAD CSV`) expects graph data to be provided in CSV format. The generated CSV files therefore follow exactly the structure expected by Neo4j's bulk loader, ensuring compatibility and efficient ingestion.
2. **Bulk ingestion.** CSV files are loaded into Neo4j (via `LOAD CSV`). The main focus of this step is to create/upsert nodes and edges but for certain nodes and edges there is additional logic in the import query. This is necessary to conditionally add certain labels which will be used for the violations.
3. **Property data type conversion.** The violations for the node property type constraint and edge property type constraint require that a property is not a string and so a boolean was chosen. However, `LOAD CSV` always parses field as strings so even if a boolean value is sent, it will end up as a string in Neo4j. To circumvent this, a set of Cypher queries are executed to find the relevant nodes and edges and to convert them to booleans.
4. **Post-ingest validation queries.** After ingestion and the property data type conversion finishes, a structured set of Cypher queries is executed against the dataset to identify constraint violations. Each query is instantiated from the schema-to-query mapping

(see Table 5.2), ensuring that all constraint types are covered by corresponding Cypher queries.

5. **Reporting.** Query results are aggregated into a structured violation report (JSON) that indicates which constraint was violated and includes relevant information about the affected nodes or edges, such as their labels, `externalId` and other key properties.

### Pre-validation Workflow

The pre-validation strategy performs validation prior to or during data insertion, following an incremental and node type-aware approach. The input JSON file is first parsed into a hierarchy of *Data Transfer Objects* (DTOs), which serve as lightweight, structured representations of the graph elements. Each DTO encapsulates the relevant properties for its corresponding node type, allowing validation and transformation to occur independently of the database layer. Validation functions and Cypher existence checks are executed in a controlled sequence, ensuring that nodes and edges are upserted only after all relevant validations succeed.

### Detailed flow

1. **JSON → DTO parsing.** Input JSON is parsed into domain DTOs (one DTO per node/edge).
2. **DTO type-by-DTO type validation and upsert.** For each DTO type, the backend:
  - a) Runs property-level validations (lengths, required fields, formatting).
  - b) Executes targeted Cypher checks when necessary (e.g., verify that a referenced parent node already exists).
  - c) Only if checks pass, performs the upsert (`MERGE/CREATE`) for that DTO.
3. **Iterate to next DTO type.** The process repeats DTO type-by-DTO type; because some validations depend on previously inserted DTOs, the order matters.

### Experimental Protocol and Metrics

Because the two ingestion workflows differ significantly in structure, their metrics are not directly comparable in all aspects. In the query-driven validation strategy, ingestion and validation occur in two distinct phases, while in the pre-validation workflow, validation and insertion are tightly interleaved throughout the process.

**Query-driven validation.** For this strategy, the following metrics are recorded separately:

- **JSON-to-CSV conversion time** - the time taken to transform the input JSON into CSV files.
- **Ingestion time** - the time taken to import all nodes and edges into Neo4j from CSV.
- **Boolean conversion time** - the time required to execute the Cypher queries that convert selected node and edge properties from strings to booleans, as described in the property data type conversion step.
- **Validation time** - the time taken to run all post-ingestion Cypher validation queries.
- **Total time** - the overall wall-clock time from when the input JSON file is submitted to when all validation queries have finished running.

- **Throughput** - calculated as the total number of graph elements (nodes and edges) ingested divided by the ingestion time.
- **Failure counts** - the number of entities flagged as violating any constraint.

**Pre-validation.** For the pre-validation workflow described in Section 4.4.2, the tightly coupled nature of validation and ingestion makes it impractical to measure validation and insertion times separately. Therefore, only the **total time** - from when the input JSON file is submitted until all nodes and edges have been validated and ingested - is recorded. This total duration serves as the single directly comparable metric between the two workflows.

**Experimental procedure.** Each experiment run follows the same overall procedure:

1. Prepare the dataset (synthetic JSON) and record its size (number of nodes and edges).
2. Run the chosen ingestion strategy.
3. Automatically capture execution times and derived performance metrics at predefined stages of the ingestion and validation pipeline.
4. Collect and when applicable, export the results of validation queries (for the query-driven approach).
5. Repeat each experiment at least three times per dataset size to ensure reliability and reproducibility.

## 4.5 Evaluation Setup

To ensure reproducibility and consistency in the performance measurements, all experiments were conducted on a dedicated machine. The system configuration, including operating system, processor, memory, storage and relevant software versions, is summarized below. This section also details the configuration settings to disable Neo4j's query plan cache and why that was necessary for the evaluation.

### 4.5.1 Experimental Environment

The evaluation aims to compare the performance and scalability of the two ingestion and validation strategies described in Section 4.4.2. Each strategy is evaluated independently using the same synthetic datasets of increasing size and structural complexity to emulate realistic enterprise data ingestion scenarios.

All experiments are executed on a dedicated machine to ensure consistent and reproducible performance results. The environment specifications are as follows:

- **Operating system:** Windows 11 Pro, version 24H2 (build 26100.7171)
- **Processor:** 11th Gen Intel(R) Core(TM) i7-11800H @ 2.30GHz, 8 cores / 16 threads
- **Memory:** 64 GB RAM
- **System type:** 64-bit operating system, x64-based processor
- **Storage:** 475 GB NVMe SSD
- **Neo4j Browser version:** 4.4.2
- **Neo4j Server version:** 4.4.4 (Community Edition)
- **Backend runtime:** OpenJDK 21 with Spring Boot 3.4.1

### 4.5.2 Neo4j Configuration for Query-driven Validation

Neo4j caches query plans to improve performance when executing repeated Cypher statements<sup>4</sup>. By default, the database maintains up to 1000 cached query plans, split between two caches: a *string cache* and an *AST (abstract syntax tree) cache*.

The string cache stores plans keyed by a hash of the query string, meaning that semantically equivalent queries with differences in casing or whitespace may result in separate cached plans. The AST cache stores normalized queries, allowing queries with the same structure but different literal values to reuse a cached plan. While caching improves performance in general, it causes the first execution of a query to be slower than subsequent executions, particularly in query-driven validation where each statement is executed repeatedly against large datasets. Subsequent runs benefit from cached plans, leading to inconsistent timing results that do not reflect the true performance of the ingestion and validation strategy.

To avoid this problem, the following Neo4j configuration setting were applied for the query-driven validation experiments:

```
dbms.query_cache_size=0
```

This setting was applied *only* for the query-driven validation experiments. For the pre-validation strategy, which executes fewer queries per import, the query plan caching was left enabled, as the performance discrepancy between the first and subsequent runs was not observed in that strategy.

---

<sup>4</sup><https://neo4j.com/developer/kb/understanding-the-query-plan-cache/>, accessed: 2025-12-12



## 5 Results

This chapter presents the results of the thesis work, organized according to the three research questions introduced in Chapter 1. First, the set of schema constraints relevant to property graphs is identified through a literature-driven synthesis and cross-checking against Entiros' requirements (RQ1). Second, these schema constraints are mapped to Cypher query patterns for in-database validation (RQ2). Finally, the performance and effectiveness of different validation strategies are evaluated in the context of an industrial Neo4j pipeline (RQ3). Together, these results provide both a conceptual catalog of property graph constraints and empirical evidence for the feasibility and scalability of query-driven schema validation.

### 5.1 Identified Schema Constraints

This section presents the set of schema constraints relevant to property graphs, addressing RQ1. The schema constraints were identified through a combination of literature review and consultation with Entiros. The resulting schema catalog captures both widely recognized constraint types in property graph literature and domain-specific requirements arising from real-world usage in Entiros' platform.

A complete list of schema constraints and their definitions is presented in Table 5.1. The following subsections highlight key categories and examples, particularly for constraints arising in Entiros' industrial platform.

#### 5.1.1 Summary of Literature-derived Schema Constraints

Existing schema languages for property graphs provide a relatively consistent core set of constraint types. Across PG-Schema [5], GraphQL SDL [6] and prior work on schema validation and evolution [7], three recurring categories can be identified:

**Property-level constraints.** These include uniqueness, mandatory properties, key constraints and type constraints. They form the most widely supported group, appearing in all surveyed sources. Property-level constraints define the expected structure of node and edge attributes and are fundamental to ensuring data consistency.

**Structural constraints.** Several schema languages define constraints on graph topology, such as edge typing rules (restricting the permitted combinations of source and target labels) and cardinality constraints for incoming and outgoing edges. These constraints reflect

the structural semantics of property graphs and are especially prominent in PG-Schema and GraphQL SDL.

**Label constraints.** Some schema languages provide mechanisms to restrict or refine the label sets of nodes. This includes label hierarchies, co-occurrence rules and closed-world label sets. These constraints help capture schema-level semantics around classification and inheritance.

Overall, the literature shows substantial agreement on a core schema vocabulary centered around properties, structure and labels. However, constraints relating to temporal consistency, scoped uniqueness and conditional or path-based rules are not represented in existing schema languages.

### 5.1.2 Industry-derived Schema Constraints

In addition to the literature-derived constraints, several further constraint types were identified through collaboration with Entiros. These constraints reflect requirements that arise in their industrial integration platform but are largely absent from existing schema languages.

**Temporal constraints.** These cover rules on the validity and ordering of timestamp properties, ensuring that time-related information remains consistent.

**Property value and length restrictions.** Certain node types require property values to be drawn from predefined enumerations or to satisfy maximum length limitations. Such domain-specific property restrictions are generally unsupported in existing schema languages.

**Scoped and conditional constraints.** Some constraints apply only within specific sub-graphs or under certain conditions, including scoped uniqueness, conditional exclusion of edges and dependency rules that activate only when particular property or structural conditions hold.

**Path and domain-scoping rules.** Because Entiros' platform isolates data into domains, constraints are needed that restrict connectivity across domain boundaries, preventing disallowed paths between nodes.

Together, these industry-derived schema constraints extend the literature-based catalog by capturing operational and platform-specific requirements observed in real-world property graph systems.

### 5.1.3 Consolidated Schema Catalog

The resulting schema constraints were consolidated into a schema catalog, presented in Table 5.1. For each entry, the table documents the constraint type, its definition, if it is from Entiros, which sources reference it, if it is required in the Entiros pipeline and how Entiros currently implements it.

### 5.1.4 Glossary of Constraint Types

To aid interpretation, the following text briefly elaborates on the constraints in Table 5.1, illustrating how they typically manifest in property graph settings.

**Node property uniqueness** ensures that nodes of a given label do not share the same value for a specified property (e.g., two `Person` nodes cannot share an `email`). **Edge property uniqueness** applies the same principle to edges. **Node key** and **edge key constraints** generalize uniqueness to combinations of properties. **Mandatory properties** require that certain attributes must always be present, while **property type constraints** ensure that values conform to expected data types. Property-set constraints may further restrict properties to a predefined set (closed world) or require at least a base set of properties (open world).

Connectivity-related constraints include **minimum** and **maximum edge cardinalities**, which specify lower and upper bounds on the number of allowed connections between

Constraint Type	Definition / Purpose	From Entries	PG-Schema	GraphQL SDL	Bonifati et al.	Required in Entries Pipeline	Currently Implemented With
<b>Property-level constraints</b>							
Node property uniqueness	No two nodes with label X share property P value.	-	✓R4	✓DS7	✓(identity)	✓	None; Constraint, Program Code
Edge property uniqueness	No two edges of label Y share property P value (for same src/dst).	-	✓R4	✓DS7	✓(identity)	-	-
Node key constraint	Nodes with label X must have properties $P_1, \dots, P_n$ and combination is unique.	-	✓R4	✓DS7	✓(identity)	✓	Program Code
Edge key constraint	Edges with label Y must have properties $P_1, \dots, P_n$ and combination is unique.	-	✓R4	✓DS7	✓(identity)	-	-
Mandatory node property	Every node with label X must have property P.	-	✓R5	✓DS5	✓(mandatory property)	✓	Program Code
Mandatory edge property	Every edge of label Y must have property P.	-	✓R5	✓DS6	✓(mandatory property)	-	-
Closed-world property set	Node with label X may only have properties $P_1, \dots, P_n$ .	-	✓R3	-	-	-	-
Open-world property set	Node/edge may have arbitrary additional properties.	-	✓(R7, R8)	-	-	-	-
<b>Property value constraints</b>							
Node property value constraint	Property P of a node with label X must take its value from a predefined set of values $v_1, \dots, v_n$ .	✓	-	-	-	✓	Program Code
Node property length constraint	String property value Y of node with label X should not exceed N length.	✓	-	-	-	✓	Program Code
Temporal validity constraint	No node may have temporal property values set to a future time.	✓	-	-	-	✓	Program Code
Temporal ordering constraint	For any node, the value of temporal property $P_1$ must not precede the value of temporal property $P_2$ .	✓	-	-	-	✓	Program Code
Node property type constraint	Node property P must have type T (Integer, String, etc.).	-	✓R3	✓(WS1, WS2)	✓(type)	✓	Program Code
Edge property type constraint	Edge property P must have type T (Integer, String, etc.).	-	✓R3	✓(WS1, WS2)	✓(type)	-	-
<b>Structural constraints</b>							
Min outgoing edges	Node with label X must have $\geq N$ outgoing edges of label Y.	-	-	✓WS4	✓(cardinality)	✓	Queries, Program Code
Max outgoing edges	Node with label X must have $\leq N$ outgoing edges of label Y.	-	-	✓WS4	✓(cardinality)	✓	Queries, Program Code
Min incoming edges	Node with label X must have $\geq N$ incoming edges of label Y.	-	-	-	✓(cardinality)	✓	Queries, Program Code
Max incoming edges	Node with label X must have $\leq N$ incoming edges of label Y.	-	-	-	✓(cardinality)	✓	Queries, Program Code
Edge typing constraint	Edge of label Y must connect nodes of labels $A \rightarrow B$ .	-	✓R2	✓WS3	✓(type/domain)	✓	Queries
No loops	Prohibit self-edges of label X, i.e., $(v) \rightarrow (v)$ .	-	-	✓DS2	-	✓	Program Code
<b>Path / pattern constraints</b>							
Node path restriction constraint	From a given node with label X, traversing the graph over edges of labels A, B, C, ..., N, may only connect it to a single node of label Y.	✓	-	-	-	✓	Queries, Program Code
Conditional relationship exclusion constraint	If a node with label X has an incoming edge from a node with label Y such that Yproperty = v, then the node X must not have any outgoing edge to a node with label Z.	✓	-	-	-	✓	Program Code
Scoped uniqueness constraint	For nodes with label X connected by an edge of label T to the same node, property P must be unique within that scope.	✓	-	-	-	✓	Program Code
Conditional relationship dependency	An edge of label E from node A to node B is only valid if there also exists an edge of label F from A to B.	✓	-	-	-	✓	Program Code
<b>Label constraints</b>							
Node label constraint	Node with label X must also have label Y (hierarchy).	-	✓R6	-	-	✓	Queries
Closed-world label set constraint	Nodes/edges may only have labels from a predefined set.	-	-	✓Example 3.1	-	✓	Queries
Label co-occurrence constraint	Nodes with label X must always also have labels Y and Z.	✓	✓R6	-	-	✓	Queries
Label exclusive-choice constraint	Nodes with label X must always have exactly one of Y, Z.	✓	-	-	-	✓	Queries, Program Code

Table 5.1: Schema catalog of property graph constraint types.

nodes. The **no loops constraint** prevents edges from connecting a node to itself. **Node label constraints** enforce label hierarchies, while **edge typing constraints** restrict valid source–target label combinations for a given edge type.

### Entiros Constraint Use Cases

The following examples illustrate how the constraint categories above manifest in Entiros’ platform.

**Temporal constraints** appear as rules ensuring that the values of timestamp properties such as `created` and `updated` never lie in the future and that `updated` does not precede `created`.

**Property restrictions** include enumerated subscription types and maximum string lengths for selected attributes, enforcing domain-specific value validity.

**Domain-scoped path rules** ensure that nodes remain within their designated workspace or domain. For example, edges that connect nodes across domains are disallowed, maintaining strict data isolation.

**Conditional relationship exclusion** occurs when certain subscription types or configuration states forbid the presence of specific edges.

**Scoped uniqueness** is used to ensure that names or identifiers are unique within a given workspace or network, without requiring global uniqueness.

**Conditional relationship dependency** ensures that certain edges may only exist if supporting edges are also present, preserving logical consistency in the graph’s structure.

### 5.1.5 Observations from the Catalog

Several patterns emerge from the consolidated schema catalog:

- **Property-level constraints** are the most consistently supported across both literature and industry. Uniqueness, mandatory properties and type constraints appear in all surveyed sources and form the backbone of graph schema validation.
- **Structural and edge constraints** such as edge typing and cardinality are frequently defined in literature and required in the industrial setting, but some edge-level constraints (e.g., edge property uniqueness) are not relevant for Entiros’ data model.
- **Label constraints** are widely recognized in literature and are also applied in the industrial context to enforce hierarchy and co-occurrence rules.
- **Domain-specific constraints** such as temporal validity, scoped uniqueness, conditional relationship rules and path restrictions appear exclusively in the Entiros pipeline. These highlight gaps in existing schema languages with respect to more specific, real-world operational requirements.
- Overall, the catalog reflects a combination of standard, widely supported constraints and specialized, industrially motivated constraints, providing a comprehensive view of schema constraints relevant to property graphs.

## 5.2 Mapping Schema Constraints to Cypher Queries

This section answers RQ2 by translating each identified schema constraint into a Cypher query pattern that detects violations in Neo4j.

### 5.2.1 Table of Cypher Query Templates

Table 5.2 presents the resulting catalog of query templates, each parameterized by labels, properties or edge types. These templates show how Cypher can operationalize the schema semantics introduced in RQ1 as concrete graph-validation rules.

Placeholders such as `%label%` and `%prop%` are used in the Cypher Query Template column to denote labels and properties that are instantiated when a specific constraint is applied.

### 5.2.2 Observations from the Mapping

Across the constraint types, several patterns emerge in how schema constraints translate into Cypher. Simple property- or label-based constraints map directly to single-pattern queries, while edge- and scope-based constraints typically require more complex constructs such as aggregation, path exploration or multi-hop pattern matching. Some constraint types rely on APOC procedures due to the absence of native Cypher support for type introspection. Cardinality, scoped uniqueness and conditional constraints illustrate that Cypher can express higher-level schema semantics, but at the cost of more complex and potentially expensive queries. These observations highlight both the expressive power and the practical limitations of using Cypher as a schema-validation language.

## 5.3 Experiment Results

This section presents the results obtained from the experimental evaluation of the two validation strategies. The results are organized into three parts. First, the overall execution time of pre-validation is compared against query-driven validation, across all dataset scale factors. Second, the execution time of query-driven validation is broken down into its constituent processing stages to identify their relative contributions. Finally, the throughput of the query-driven approach is reported in terms of processed graph elements per second for each dataset.

### 5.3.1 Comparison Between Pre-validation and Query-driven Validation

Figures 5.1a–5.1d show the total execution times for pre-validation and query-driven validation across the different dataset scale factors. Each figure presents the mean execution time with error bars representing the standard deviation over the three experimental runs.

Table 5.3 summarizes all execution time measurements across the different scale factors in a single view. It reports the mean and standard deviation for both pre-validation and query-driven validation, using seconds for all mean values and milliseconds for the very small standard deviations in the query-driven case. Additionally, the table provides the speedup, calculated as the ratio between pre-validation and query-driven validation mean execution times. This consolidated overview makes it easy for the reader to compare the two strategies directly and to see how the performance gap increases with dataset size.

Constraint Type	Violation Condition	Cypher Query Template	Example Instantiation
Property-level constraints			
Node property uniqueness	Two nodes with the same value for a property that should be unique.	<pre> MATCH (n1:%label%), (n2:%label%) WHERE n1.%prop% = n2.%prop% AND id(n1) &lt; id(n2) RETURN n1, n2 </pre>	-
Edge property uniqueness	Two edges with the same label between the same nodes sharing the same property value.	<pre> MATCH (n1)-[e1:%label%]-&gt;(n2), (n1)- [e2:%label%]-&gt;(n2) WHERE e1.%prop% = e2.%prop% AND id(e1) &lt; id(e2) RETURN DISTINCT e1, e2, n1, n2 </pre>	-
Node key constraint	Given a set $\{p_1, \dots, p_m\}$ of $m \geq 1$ properties, the constraint is violated if two nodes share the same values for all these properties.	<pre> // m=2 MATCH (n1:%label%), (n2:%label%) WHERE n1.%prop1% = n2.%prop1% AND n1.%prop2% = n2.%prop2% AND id(n1) &lt; id(n2) RETURN n1, n2 </pre>	<pre> MATCH (n1:System), (n2:System) WHERE n1.id = n2.id AND n1.name = n2.name AND id(n1) &lt; id(n2) RETURN n1, n2 </pre>
Edge key constraint	Given a set $\{p_1, \dots, p_m\}$ of $m \geq 1$ properties, the constraint is violated if two edges share the same values for all these properties.	<pre> // m=2 MATCH (n1)-[e1:%label%]-&gt;(n2), (n1)- [e2:%label%]-&gt;(n2) WHERE e1.%prop1% = e2.%prop1% AND e1.%prop2% = e2.%prop2% AND id(e1) &lt; id(e2) RETURN e1, e2, n1, n2 </pre>	-
Mandatory property	A node of a given label is missing a required property.	<pre> MATCH (n1:%label%) WHERE n1.%prop% IS NULL RETURN n1 </pre>	<pre> MATCH (n1:System) WHERE n1.id IS NULL RETURN n1 </pre>
Mandatory property	An edge of a given label is missing a required property.	<pre> MATCH ()-[e:%label%]-&gt;() WHERE e.%prop% IS NULL RETURN e </pre>	-

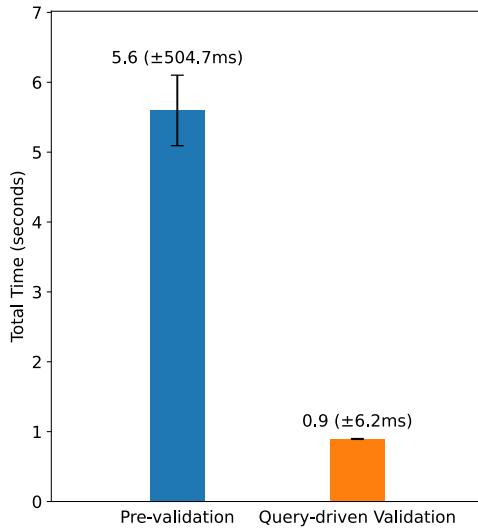
Constraint Type	Violation Condition	Cypher Query Template	Example Instantiation
Closed-world property set	Node has a property outside the allowed set.	<pre>MATCH (n:%label%) WHERE any(p IN keys(n) WHERE NOT p IN %allowed_property_set%) RETURN n</pre>	-
<b>Property value constraints</b>			
Node property value constraint	Node has a property with a value outside the allowed set.	<pre>MATCH (n:%label%) WHERE NOT n.%prop% IN %allowed_property_values% RETURN n</pre>	-
Node property length constraint	Node has string property with length greater than N.	<pre>MATCH (n:%label%) WHERE size(n.%prop%) &gt; %max_length% RETURN n</pre>	-
Temporal validity constraint	Node has a temporal property set to a future datetime.	<pre>MATCH (n) WHERE n.%temporal_prop% &gt; datetime() RETURN n</pre>	-
Temporal ordering constraint	Node has temporal property $P_1$ with a value earlier than temporal property $P_2$ .	<pre>MATCH (n) WHERE n.%temporal_prop1% &lt; n.%temporal_prop2% RETURN n</pre>	-
Node property type constraint	A node property has a value of the wrong type.	<pre>MATCH (n:%label%) WHERE n.%prop% IS NOT NULL AND apoc.meta.type(n.%prop%) &lt;&gt; %data_type% RETURN n</pre>	-
Edge property type constraint	An edge property has a value of the wrong type.	<pre>MATCH (n1)-[:%label%]-&gt;(n2) WHERE e.%prop% IS NOT NULL AND apoc.meta.type(e.%prop%) &lt;&gt; %data_type% RETURN e</pre>	<pre>MATCH (n1:Network)- [:MODEL\$]-&gt;(n2:System) WHERE e.name IS NOT NULL AND apoc.meta.type(e.name) &lt;&gt; "STRING" RETURN n1,n2, e;</pre>

Constraint Type	Violation Condition	Cypher Query Template	Example Instantiation
Structural constraints			
Min outgoing edges	Node has fewer outgoing edges of label T than required.	<pre> MATCH (n:%label%) OPTIONAL MATCH (n)-[:%label%]-&gt;(m) WITH n, count(m) AS outgoing_count WHERE outgoing_count &lt; %min_outgoing_edges% RETURN n </pre>	<pre> // min_outgoing_edges = 1 MATCH (n:Thesis:Network) OPTIONAL MATCH (n)-[:MODELS]-&gt;(m) WITH count(m) AS outgoing_count WHERE outgoing_count &lt; 1 RETURN n; </pre>
Max outgoing edges	Node has more outgoing edges of label T than allowed.	<pre> MATCH (n:%label%) OPTIONAL MATCH (n)-[:%label%]-&gt;(m) WITH n, count(m) AS outgoing_count WHERE outgoing_count &gt; %max_outgoing_edges% RETURN n </pre>	-
Min incoming edges	Node has fewer incoming edges of label T than required.	<pre> MATCH (n:%label%) OPTIONAL MATCH (m)-[:%label%]-&gt;(n) WITH n, count(m) AS incoming_count WHERE incoming_count &lt; %min_incoming_edges% RETURN n </pre>	-
Max incoming edges	Node has more incoming edges of label T than allowed.	<pre> MATCH (n:%label%) OPTIONAL MATCH (m)-[:%label%]-&gt;(n) WITH n, count(m) AS incoming_count WHERE incoming_count &gt; %max_incoming_edges% RETURN n </pre>	-
Edge typing constraint	An edge connects nodes with disallowed labels.	<pre> MATCH (n1)-[:%label%]-&gt;(n2) WHERE NOT (n1:%label% AND n2:%label%) RETURN n1, e, n2 </pre>	-
No loops	An edge connects a node to itself.	<pre> MATCH (n)-[:%label%]-&gt;(n) RETURN n, e </pre>	-

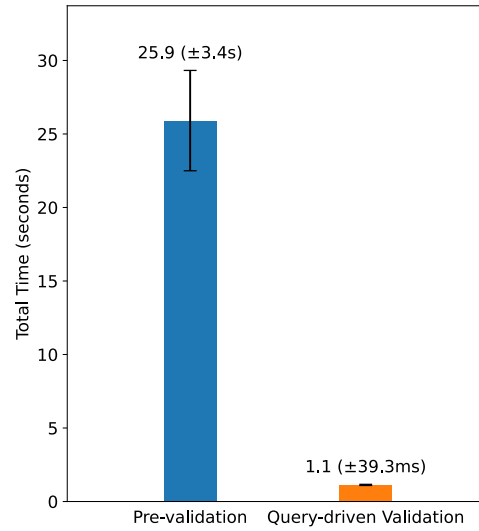
Constraint Type	Violation Condition	Cypher Query Template	Example Instantiation
Path / pattern constraints			
Node path restriction constraint	A node with label X can reach more than one distinct node with label Y via allowed edge labels $\{A, B, \dots, N\}$ .	<pre> MATCH (n:%source_label)-[:%allowed_edge_labels]-&gt;(m:%target_label) WITH n, collect(DISTINCT m) AS targets WHERE size(targets) &gt; 1 RETURN n, targets </pre>	-
Conditional relationship exclusion constraint	A node X connected to a node Y with $Y.prop = v$ also has an edge to a node Z.	<pre> MATCH (n1:%source_label)-[:%edge_label1%]-(n2:%target_label){%prop%:%prop_value%} MATCH (n1)-[:%edge_label2%]-&gt;(n3:%second_target_label%) RETURN n1, n2, n3 </pre>	-
Scoped uniqueness constraint	Two nodes with label X connected to the same node by an edge of label T share the same property P.	<pre> MATCH (n1:%label1%)-[:%label2%]-&gt;(m)&lt;[:%label2%]-(n2:%label1%) WHERE n1 &lt;&gt; n2 AND n1.%prop% = n2.%prop% RETURN m, n1, n2 </pre>	-
Conditional relationship dependency	An edge of label E exists from nodes A to B, but no edge of label F exists from A to B.	<pre> MATCH (n1:%label1%)-[:%label2%]-&gt;(n2:%label3%) WHERE NOT (n1)-[:%label4%]-&gt;(n2) RETURN n1, n2, e </pre>	-
<b>Label constraints</b>			
Node label constraint	Given a set $\{L_1, \dots, L_m\}$ of $m \geq 1$ expected labels, the constraint is violated if a node does not have any of these labels.	<pre> // m=2 MATCH (n:%label1%) WHERE NOT (n:%label2% OR n:%label3%) RETURN n </pre>	-
Closed-world label set constraint	Node's full label set $\notin$ allowed sets.	<pre> MATCH (n:%label%) WHERE any(1 IN labels(n) WHERE NOT 1 IN %allowed_labels%) RETURN n </pre>	-

Constraint Type	Violation Condition	Cypher Query Template	Example Instantiation
Label co-occurrence constraint	Node has label X but is missing required label(s) Y, Z.	<pre>MATCH (n:%label1%) WHERE NOT (n:%label2% AND n:%label3%) RETURN n</pre>	-
Label exclusive-choice constraint	A node must have exactly one label from the allowed set $\{L_1, \dots, L_m\}$ . A violation occurs if the node has none or more than one of these labels.	<pre>MATCH (n:%label%) WHERE size([l IN %allowed_labels% WHERE l IN labels(n)]) &lt;&gt; 1 RETURN n</pre>	<pre>// allowed_labels = [System, Service] MATCH (n) WHERE size([l IN ['System', 'Service'] WHERE l IN labels(n)]) &lt;&gt; 1 RETURN n</pre>

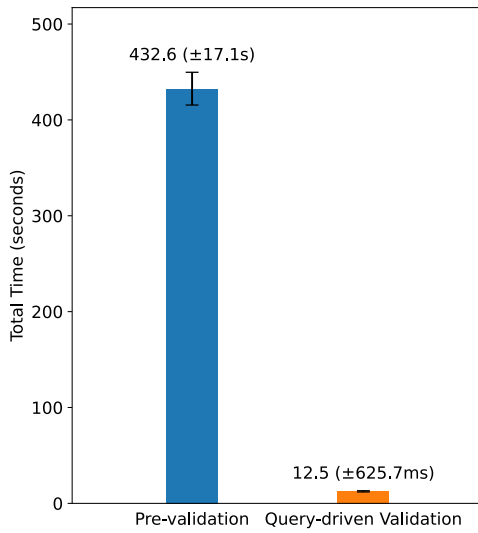
Table 5.2: Mapping of graph schema constraint types with corresponding violation conditions and Cypher queries for detection.



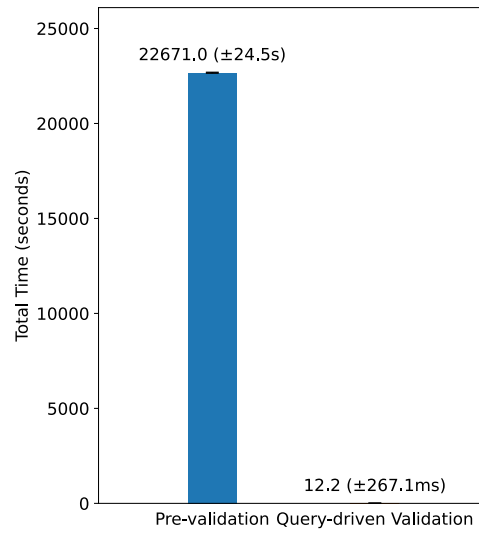
(a) Comparison for dataset SF-1.



(b) Comparison for dataset SF-10.



(c) Comparison for dataset SF-100.



(d) Comparison for dataset SF-1000.

Figure 5.1: Total execution times for pre-validation and query-driven validation.

From the charts and the table, it is evident that query-driven validation consistently outperforms pre-validation across all dataset sizes. The performance gain becomes increasingly significant for larger datasets: while the speedup for SF-1 is moderate, for SF-1000 the execution time of pre-validation is three orders of magnitude higher than that of query-driven validation. The error bars remain relatively small for query-driven validation, indicating stable and predictable performance.

Dataset (Scale Factor)	Pre-validation Mean (s)	Pre-validation SD (s)	Query-driven Validation Mean (s)	Query-driven Validation SD (ms)	Speedup
SF-1	5.6	0.5	0.9	6.2	6.3x
SF-10	25.9	3.4	1.1	39.3	22.9x
SF-100	432.6	17.1	12.7	559.4	34.2x
SF-1000	22671.0	24.5	12.2	267.0	1863.2x

Table 5.3: Overall execution time comparison between strategies.

### 5.3.2 Breakdown of Query-driven Validation

Table 5.4 provides a detailed breakdown of the query-driven validation strategy. The table lists the mean execution time of each processing stage - JSON to CSV conversion, ingestion, boolean conversion and validation - as well as the total execution time for each dataset scale factor.

Figure 5.2 visualizes these values in a stacked chart, where each segment corresponds to one stage of the processing pipeline. This representation makes it easier to compare the relative contributions of each stage across different dataset sizes.

From both the table and the chart, several observations can be made:

- **JSON to CSV conversion** contributes only a small fraction of the total execution time for all datasets, even for the largest scale factor, indicating that this stage is not a bottleneck.
- **Ingestion** dominates the total time for the medium-sized dataset (SF-100), whereas for the largest dataset (SF-1000), both ingestion and validation stages contribute substantially, highlighting the scaling impact.
- **Boolean conversion** consistently contributes the least time across all datasets, confirming it is a minor processing step.
- **Validation** grows in absolute time with dataset size, reflecting the increased number of graph elements processed. Its contribution becomes more significant for SF-1000.
- Overall, the stacked chart clearly shows that the majority of execution time is concentrated in the ingestion and validation stages, while conversion-related operations are negligible.

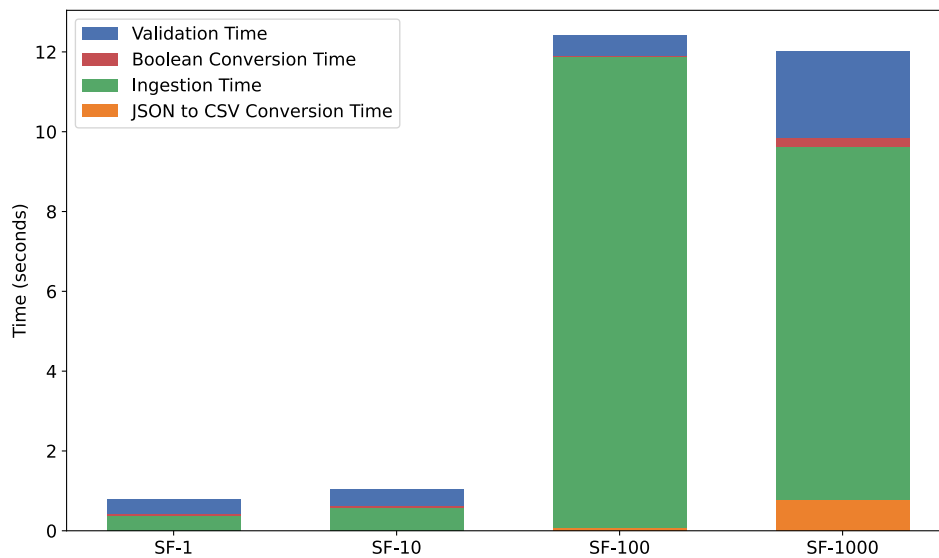


Figure 5.2: Stacked chart of query-driven validation execution time by stage.

Dataset (Scale Factor)	JSON → CSV Conv. (ms)	Ingestion (s)	Boolean Conv. (ms)	Validation (s)	Total (s)
SF-1	5.6	0.4	27.9	0.4	0.9
SF-10	11.1	0.6	28.8	0.4	1.1
SF-100	70.3	11.9	36.8	0.5	12.7
SF-1000	763.2	8.9	222.7	2.2	12.2

Table 5.4: Breakdown of query-driven validation execution time by stage.

### 5.3.3 Throughput Results

Figure 5.3 summarizes the throughput achieved by the query-driven validation strategy during ingestion. Throughput is expressed as the number of graph elements ingested per second and the chart reports the mean and standard deviation for each dataset scale factor.

Across the datasets, throughput varies noticeably with scale. For SF-1 and SF-10, the measured throughput is approximately 1,150 elements/s and 4,460 elements/s, respectively, indicating higher per-element processing rates for these smaller datasets. For SF-100, throughput is lower, at around 2,000 elements/s. For the largest dataset, SF-1000, throughput increases substantially, reaching about 26,600 elements/s. In all cases, the standard deviation is small relative to the mean, indicating stable performance across repeated runs.

Overall, the chart shows that throughput does not follow a monotonic trend with dataset size and instead, the measured rates vary across scales, with the highest throughput observed for SF-10 and SF-1000.

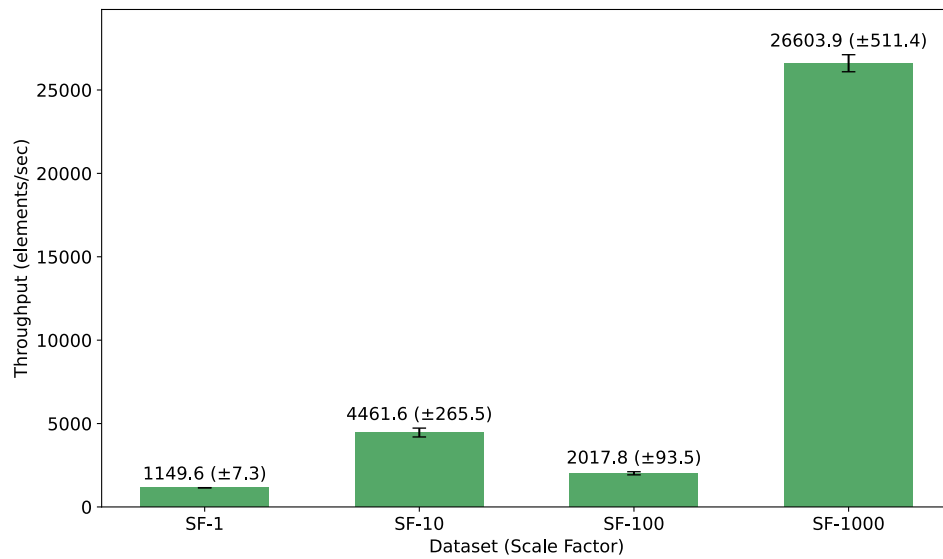


Figure 5.3: Ingestion time throughput for query-driven validation per dataset.



## 6 Discussion

This chapter discusses and reflects on the results, methods and broader implications of the work presented in this thesis. The discussion interprets the experimental findings, critically examines the methodological choices and their limitations and situates the proposed validation approach within a wider industrial, societal and ethical context.

### 6.1 Discussion of Results

The following section discusses the results from the literature review, the schema constraint to query mapping and experiment results.

#### 6.1.1 Schema Catalog

##### Supported Constraint Types

The schema catalog highlights a clear distinction between constraint types that are well supported in existing property graph schema languages and those that emerge primarily from industrial practice. Core structural and property-level constraints such as mandatory properties, type constraints and global uniqueness are consistently represented across the surveyed literature. These constraints describe local node or edge properties and align closely with the data modeling capabilities traditionally emphasized in schema design.

##### Contextual and Industrial Constraints

In contrast, a substantial portion of the constraints identified through collaboration with Entiros are either weakly supported or entirely absent from existing schema languages. In particular, contextual constraints such as scoped uniqueness, conditional relationship dependencies and path-based restrictions are not addressed by the surveyed approaches. These constraints depend on the surrounding subgraph, workspace boundaries or conditional combinations of properties and edges, reflecting operational and business-level requirements rather than purely structural ones. Their absence suggests that current schema languages primarily target static structure, while industrial systems require schemas that also encode dynamic and contextual validation logic.

### **Structural vs. Validation Semantics**

The results further indicate that practical schemas often blur the boundary between structural modeling and validation rules. Several constraints identified in the catalog - especially those related to domain isolation, conditional exclusions and temporal consistency - function as integrity rules that enforce application-level semantics. This observation supports the view that property graph schemas in real-world systems act not only as documentation or typing mechanisms, but also as executable validation specifications that ensure consistency during data ingestion and evolution.

### **Redundancy and Composability**

Finally, the catalog reveals a degree of redundancy and composability among constraint types. For example, key constraints can be expressed as combinations of mandatory and uniqueness constraints. While some schema languages elevate such composite constraints to first-class constructs, their underlying semantics remain reducible to simpler primitives. This raises questions about the appropriate level of abstraction in schema languages and whether greater emphasis should be placed on composability and reuse rather than expanding the set of primitive constraint types.

## **6.1.2 Query Mapping**

### **Expressibility of Constraints**

The query mapping demonstrates that all identified validatable schema constraints, including complex contextual and conditional ones, can be expressed as Cypher queries that detect violations in Neo4j. This finding confirms that Cypher is sufficiently expressive to function as a general-purpose validation language for property graphs, even in the absence of native support for many constraint types. In particular, constraints involving aggregation, negated patterns and multi-node conditions can be implemented using standard Cypher constructs.

The open-world property set rule falls outside this category. By design, it does not admit meaningful violations, as it explicitly allows nodes or edges to have arbitrary additional properties beyond those specified. Consequently, it represents a permissive schema rule rather than a constraint intended for violation-based validation.

### **Recurring Query Patterns**

Across the catalog, recurring query patterns emerge. Uniqueness-related constraints are typically expressed using self-joins combined with identity checks, while cardinality constraints rely on aggregation and counting. Conditional and scoped constraints frequently require multiple pattern matches and explicit negation to identify invalid structures. These patterns suggest that, although Cypher provides the necessary expressive power, implementing schema validation in this manner requires careful query design and a deep understanding of both the data model and query semantics.

### **Complexity and Maintainability**

The mapping also highlights an increase in query complexity as constraints move from local to contextual. Simple property-level constraints can often be expressed concisely, whereas scoped uniqueness, path restrictions and conditional dependencies require longer queries with multiple matches and intermediate collections. This has implications for maintainability, as complex validation queries may become difficult to reason about, debug and evolve alongside the data model. Without additional abstraction or tooling support, managing a large set of such queries can introduce cognitive and operational overhead.

Overall, the query mapping illustrates a trade-off between expressiveness and complexity. While query-driven validation enables a broad range of constraints to be enforced without extending the database system, it shifts responsibility for correctness, performance and maintainability to the application layer. This observation motivates a closer examination of when query-based validation is appropriate and where alternative mechanisms may be preferable, a topic further discussed in the section on limitations of the query-driven validation strategy.

### **Caveat on Template Queries**

It should also be noted that the queries presented in the catalog are not intended to represent optimal or definitive implementations of their respective constraints. Rather, they serve as generic templates that demonstrate how different classes of schema constraints can be expressed in Cypher. In practice, individual queries may be further refined, optimized or specialized based on characteristics of the data, indexing strategies or execution plans. Consequently, the catalog should be understood as a proof of expressibility and a foundation for further refinement, rather than a collection of finalized validation queries.

## **6.1.3 Interpretation of Experiment Results**

### **Comparison between strategies**

The comparison between pre-validation and query-driven validation reveals a fundamental difference in how the two strategies scale with dataset size. While pre-validation performs adequately for small datasets, its execution time increases dramatically as the number of elements grows. This behaviour can be attributed to its incremental nature, where validation logic is interleaved with data insertion and often requires repeated database interactions and label-dependent ordering. In contrast, query-driven validation defers all checks until after ingestion, allowing validation to be expressed as set-oriented queries that operate over the complete dataset.

The observed speedups increase by three orders of magnitude for the largest dataset, indicating that the performance advantage of query-driven validation is not merely constant but asymptotic. Furthermore, the low variance observed for query-driven validation across all runs indicates stable and predictable performance, which is an important property for large-scale ingestion pipelines.

### **Breakdown of Query-driven Validation**

The breakdown of the query-driven validation pipeline shows that conversion-related steps, such as JSON-to-CSV transformation and boolean property conversion, contribute negligibly to the total execution time. This confirms that these preparatory steps do not represent performance bottlenecks and validates their inclusion in the workflow. Instead, the majority of execution time is concentrated in ingestion and validation, with their relative contributions varying across dataset sizes.

For the medium-sized dataset (SF-100), ingestion accounts for most of the total execution time, whereas for the largest dataset (SF-1000), both ingestion and validation contribute noticeably. The SF-100 ingestion time is higher than expected, even exceeding that of SF-1000. Follow-up runs to verify measurement reliability showed that SF-100 ingestion times occasionally fluctuated between 2s and 13s. These variations were uncommon but not rare and the exact cause remains unclear; possible contributors include Neo4j's internal query planning, caching, transaction handling, memory allocation, JVM warm-up or background database activity. Intermediate dataset sizes like SF-100 may be more sensitive to such factors, where caching or batching mechanisms interact less efficiently than for very small or very large datasets. Importantly, these fluctuations and anomalous throughput do not affect

the overall comparison between pre-validation and query-driven validation, as the relative scaling trends and observed performance advantages remain consistent.

#### 6.1.4 Practical Implications of the Results

The results have several practical implications for the design and operation of large-scale property graph ingestion pipelines. First, the experimental comparison shows that query-driven validation enables substantially faster validation at scale than pre-validation, particularly for large datasets. This implies that, in environments where large volumes of graph data are imported in batch-oriented workflows, deferring validation until after ingestion can significantly reduce end-to-end execution time without compromising the ability to detect constraint violations.

Second, the schema catalog and query mapping together demonstrate that industrially relevant constraints can be enforced using query-based validation even when they are not natively supported by the database system. This implies that organizations are not strictly limited by the expressiveness of built-in schema mechanisms when validating complex domain rules, but can instead leverage the query language to implement application-specific integrity checks.

The breakdown of query-driven validation further indicates that preparatory steps such as data format conversion contribute negligibly to total execution time. This suggests that integrating additional preprocessing or normalization steps into the ingestion workflow is unlikely to dominate performance, provided that the core ingestion and validation stages remain efficient. As a result, system designers can focus optimization efforts on ingestion and query execution rather than on auxiliary transformation stages.

Finally, the observed low variance in execution time for query-driven validation across repeated runs indicates stable performance behaviour. For industrial use cases, this implies that query-driven validation offers predictable execution characteristics, which is important for scheduling, capacity planning and integration into automated data pipelines. Taken together, the results suggest that query-driven validation is a viable and scalable approach for enforcing complex schema constraints in large property graph datasets.

## 6.2 Discussion of Method

This section critically discusses the methodological choices made in this thesis, focusing on their strengths, limitations and implications for the interpretation of the results. The discussion addresses the construction of the schema catalog and query mapping, the experimental design and execution and evaluates the method in terms of validity, reliability and replicability.

### 6.2.1 Methodological Limitations

A first limitation concerns the construction of the schema catalog. While the catalog is grounded in a systematic literature review and supplemented with constraints derived from Entiros' industrial requirements, it does not claim to be exhaustive. The identified schema constraints reflect a combination of commonly supported constraints and those that were deemed particularly relevant in the studied industrial context. Other application domains may rely on additional constraint types or emphasize different aspects of schema validation, which are not captured in this catalog. Consequently, the results should be interpreted as representative rather than universally applicable to all property graph use cases.

A second limitation relates to the schema-to-query mapping. Although all constraint types in the catalog were successfully translated into Cypher query templates, the mapping relies on manual design and expert knowledge of Cypher semantics. Query complexity increases substantially for contextual and conditional constraints, which may affect maintain-

ability and introduce a risk of implementation errors. While each query template was tested against valid and invalid graph instances, the approach does not eliminate the possibility that alternative formulations could yield different performance characteristics or edge-case behaviours.

The experimental evaluation also introduces several methodological constraints. The use of a synthetic dataset enables precise control over structure, scale and violation coverage, but it may not fully capture the heterogeneity, skewed distributions or irregular structures found in real-world production graphs. Although the synthetic data model is inspired by an actual Entiros product, performance characteristics observed in this study may differ when applied to real datasets with different access patterns or property distributions.

In addition, each experiment was repeated three times per dataset size. While this repetition allows basic assessment of performance stability, it limits the ability to characterize variance in greater detail, particularly for the larger datasets (SF-100 and SF-1000). Initially, a higher number of repetitions was planned, but the long execution times for large-scale imports, reaching several hours per run, necessitated a reduction to ensure feasibility within the available time and computational resources.

A further limitation concerns the configuration of query plan caching in Neo4j during the experimental evaluation. For the query-driven validation strategy, query plan caching was disabled to avoid inconsistent timing results caused by query compilation and warm-up effects. In contrast, query plan caching was left enabled for the pre-validation strategy for two reasons. First, the pre-validation strategy did not exhibit the same sensitivity to plan caching effects. Second, disabling caching resulted in a significant and unrealistic performance degradation relative to Entiros' production configuration, where query caching is enabled by default.

As a result, the two strategies were evaluated under different query planning configurations. While this asymmetry may affect strict cross-strategy comparability under identical database settings, it reflects a deliberate trade-off between measurement stability and ecological validity. The reported results should therefore be interpreted as indicative of performance under realistic and strategy-appropriate operating conditions, rather than as absolute benchmarks under a uniform database configuration.

Finally, the evaluation focuses on batch ingestion and validation under controlled conditions. Incremental updates, concurrent workloads and mixed read-write scenarios were not considered. As a result, the findings primarily reflect performance characteristics in offline or batch-oriented ingestion pipelines rather than continuously evolving graph systems.

### 6.2.2 Attribution Analysis

Both the ingestion approach and the timing of validation differ between the query-driven and pre-validation strategies, making it difficult to quantify the contribution of each factor to the observed speedup. Consequently, reported performance improvements should be interpreted as indicative of the overall effectiveness of the proposed strategy in a practical setting, rather than as absolute measurements of the intrinsic efficiency of query-driven validation in isolation.

Future work could further isolate and quantify the effects of individual factors to provide a more granular understanding of performance differences. One possible approach is to introduce an additional baseline that combines bulk ingestion with batch-level pre-validation, where data is imported in bulk but validated either during or immediately after each batch using the same constraints. This baseline is expected to outperform incremental pre-validation while still underperforming relative to query-driven post-validation, because the primary performance cost arises from interleaving validation logic with data insertion rather than from ingestion mechanics alone.

### 6.2.3 Validity, Reliability and Replicability

With respect to validity, the experimental setup aligns well with the research questions. The schema catalog and query mapping address RQ1 and RQ2 by identifying constraints and demonstrating their expressibility in Cypher, while RQ3 is examined through a controlled comparison of two validation strategies across increasing dataset sizes. External validity is limited by the use of synthetic data and a single database system (Neo4j), which may affect generalization.

Reliability is ensured through deterministic dataset generation, controlled conditions and repeated runs, with a fixed pseudo-random seed guaranteeing structurally identical datasets. Low variance in query-driven validation measurements indicates stable performance.

Replicability is supported by detailed documentation of dataset construction, workflows and system configuration. Disabling Neo4j’s query plan cache ensures that execution times are not affected by prior runs.

Overall, despite limitations in scope and scale, the method provides a systematic, transparent and reproducible approach for evaluating schema validation strategies in property graph databases.

### 6.2.4 Source Criticism

The sources consulted in this work include peer-reviewed articles, technical reports and standards documentation, covering both theoretical models and practical implementations of graph database schemas and validation strategies. They provide a solid foundation for understanding property graph modeling, constraint enforcement and performance implications of large-scale data ingestion, while highlighting gaps in practical enforcement mechanisms that motivated the experimental evaluation.

Three core sources guided this thesis: Hartig and Hidders [6], Angles et al. (PG-Schema) [5] and Bonifati et al. [7]. Together, they cover schema specification, validation semantics and schema evolution, serving primarily as conceptual and methodological anchors rather than implementation blueprints. Hartig and Hidders informed the abstraction of schema constraints independent of Neo4j-specific features, PG-Schema provided a structured basis for classifying constraints and mapping them to Cypher and Bonifati et al. helped identify additional schema constraints, though their focus on schema evolution was not fully leveraged.

Other sources, particularly from RDF and SHACL, offer mature validation tooling and benchmarks, but their assumptions differ from property graph databases and are therefore of limited direct applicability. Overall, the primary sources were appropriate for framing the research questions and designing the schema catalog, but required interpretation and extension to support empirical evaluation and industrial integration.

## 6.3 Limitations of the Query-Driven Validation Strategy

While the query-driven validation strategy demonstrates substantial performance advantages for large-scale data ingestion, these gains come with a set of potential trade-offs. In particular, deferring validation until after ingestion affects how tightly validation logic is coupled to the data model, how updates and incremental changes can be handled and how invalid data is managed within the system. This section discusses the functional limitations introduced by the query-driven approach and their practical implications.

### 6.3.1 Dependency on the Underlying Data Model

The query-driven validation strategy is inherently dependent on the structure and semantics of the underlying data model. Validation queries are written against specific node labels, edge

labels and property conventions and therefore assume that the ingested graph conforms to an expected structural baseline. Changes to the data model, such as the introduction of new node and edge labels or property names, require corresponding updates to the validation query set.

In contrast to declarative schema mechanisms that are enforced automatically by the database, query-driven validation externalizes schema semantics into application-level logic. This increases flexibility, as virtually any constraint can be expressed, but it also shifts responsibility for schema consistency to the validation layer. As a result, the approach is less resilient to schema evolution unless explicit effort is made to maintain alignment between the data model and the validation queries. In industrial settings where the data model evolves frequently, this dependency may introduce additional maintenance overhead.

### 6.3.2 Implications for Data Updates and Incremental Changes

The query-driven ingestion workflow supports updates to existing nodes by relying on `MERGE` semantics in node creation queries, allowing properties to be updated without duplicating nodes. However, the handling of edges introduces limitations for incremental updates. Due to Cypher's restriction that edge labels must be compile-time literals, general edge creation cannot be expressed using parameterized `MERGE` statements. Instead, the prototype relies on `apoc.create.relationship` for dynamic edge creation.

While this approach enables a generic and high-throughput ingestion pipeline, it weakens guarantees about idempotency for edges. In particular, edges created through APOC procedures may be duplicated if the same dataset is ingested multiple times or if partial updates are applied. Although this limitation could be mitigated by defining separate `MERGE` queries for each edge label, doing so would increase query complexity and reduce the generality of the ingestion process.

As a consequence, the current implementation is better suited for batch-oriented ingestion scenarios rather than fine-grained incremental updates. Supporting robust update semantics would require additional mechanisms, such as explicit edge identifiers, deduplication queries or a more label-specific ingestion strategy.

### 6.3.3 Implications for Inserting Invalid Data

A fundamental difference between pre-validation and query-driven validation is how invalid data is handled. In the pre-validation strategy, invalid entities are rejected before insertion and therefore never become part of the persistent graph. In contrast, the query-driven approach allows all data, including invalid elements, to be ingested before validation is performed.

This design choice has several implications. On the one hand, it enables significantly higher ingestion throughput by eliminating conditional checks during insertion. On the other hand, it temporarily exposes the database to inconsistent states. If the graph is queried or consumed by downstream systems before validation completes, invalid data may influence query results or application behaviour.

In practice, this limitation can be mitigated by treating query-driven validation as part of a controlled ingestion pipeline, where the database is not exposed for general use until validation has completed. Furthermore, instead of merely reporting violations, validation queries could be extended to automatically remove or quarantine invalid graph elements once detected. In such a workflow, invalid data exists only transiently and does not affect long-term system consistency.

Overall, the insertion of invalid data represents a deliberate trade-off between strict correctness at ingestion time and scalable performance. Whether this trade-off is acceptable depends on system requirements, particularly with respect to data visibility, failure handling and consistency guarantees.

## 6.4 The Work in a Wider Context

This section situates the presented work in a broader industrial, societal and ethical context. It discusses the potential impact of efficient data validation on real-world data processing systems, as well as ethical considerations related to automated and performance-oriented validation strategies.

### 6.4.1 Industrial and Societal Impact

Efficient data validation is a key component of modern data-driven systems, particularly in industrial environments where large volumes of structured data are continuously ingested and processed. By reducing validation overhead and improving throughput, the query-driven validation strategy evaluated in this work may contribute to more efficient data pipelines, shorter ingestion times and reduced system bottlenecks.

Such improvements are especially relevant in large-scale data ingestion scenarios, including graph-based data management, monitoring systems and integration platforms where data correctness must be ensured prior to analysis or deployment. Improved validation performance may also benefit decision-support systems by enabling more timely access to validated data. In automated pipelines that rely on validated data as input, more efficient validation can reduce delays and lower the risk of propagating invalid data through subsequent processing stages.

At a societal level, improvements in data validation infrastructure can indirectly support the reliability of data-driven services, including public-sector information systems. Although the validation strategy does not directly affect end users, its contribution to more robust and scalable data processing may help sustain dependable services built on accurate data.

### 6.4.2 Ethical Considerations

While performance optimizations in data validation offer practical benefits, they also raise ethical considerations. One risk is that faster validation may be applied to incomplete or partially inconsistent data without sufficient scrutiny, potentially leading to misplaced confidence in data quality. If validation is treated primarily as a technical optimization, invalid or misleading data may still be accepted and propagated.

Another concern is the potential overemphasis on throughput at the expense of correctness, coverage or explainability. In contexts where validated data informs automated or decision-support systems, insufficient awareness of validation limitations could lead to undesirable outcomes.

A concrete risk scenario arises when invalid structural dependencies enter a production graph used for decision support. For example, if a graph represents dependencies between systems, services or organizational units, missing or incorrectly connected edges may cause downstream analyses to underestimate risk, overlook critical dependencies or produce incorrect recommendations. In enterprise integration or operational monitoring contexts, such errors could influence planning decisions, fault diagnosis or resource allocation, potentially leading to financial loss or reduced system reliability.

To mitigate such risks, automated validation strategies should be complemented by safeguards. These include clearly defined validation scopes, explicit reporting of detected violations and mechanisms to prevent unvalidated or partially validated data from being treated as authoritative. In addition, validation results should be auditable and traceable, allowing stakeholders to determine which constraints were applied and which assumptions remain unenforced.

Transparency and traceability are therefore important ethical aspects of automated validation strategies. Validation processes should be documented so that stakeholders involved in the development, operation and use of the data pipeline can understand which constraints

are enforced and which are excluded. Such transparency supports correct interpretation of validation results and helps ensure accountability for data quality.

Overall, while the proposed validation approach aims to improve efficiency, it should be applied with an awareness of its limitations and within a framework that prioritizes data integrity and responsible use of automated systems.



# 7

## Conclusion

This thesis set out to investigate whether in-database validation can improve the scalability and maintainability of graph data pipelines while ensuring schema compliance of production graphs. To this end, the work combined a systematic analysis of schema constraints for property graphs, a practical mapping of these constraints to executable Cypher queries and an empirical evaluation of alternative validation strategies integrated into an industrial Neo4j pipeline. The results demonstrate that query-driven, in-database validation is a viable and scalable approach for enforcing complex schema constraints, particularly in large-scale, batch-oriented ingestion scenarios.

### 7.1 Answers to the Research Questions

**RQ1: What are the typical elements (e.g., types of constraints) that schemas for property graphs may consist of?**

This thesis identified and classified a set of typical schema constraints for property graphs through a combination of literature review and industrial requirements. Core constraint types such as mandatory properties, property type constraints, uniqueness and cardinality are consistently supported across existing schema languages. In addition, the schema catalog highlights a range of contextual and conditional constraints - such as scoped uniqueness, path-based restrictions and conditional dependencies - that are largely absent from existing formal schema languages but are common in industrial practice. The resulting schema catalog provides a structured overview of both theoretically established and practically relevant constraint types.

**RQ2: How can these schema constraints be represented as Cypher queries that identify violations in Neo4j?**

All schema constraints identified in the catalog were shown to be representable as Cypher queries that detect violations in Neo4j. Simple, local constraints can be expressed using concise query patterns, while more complex contextual constraints require multi-pattern matches, aggregation and negation. The resulting query mapping demonstrates that Cypher is sufficiently expressive to serve as a general-purpose validation language for property graphs, even when native database constraints are insufficient. However, the mapping also reveals increasing query complexity for contextual constraints, which has implications for maintainability and tooling support.

In cases where Cypher alone is not sufficient (most notably for fine-grained data type validation), extended functionality can be provided through Neo4j’s APOC library. Procedures such as `apoc.meta.type` enable explicit type checking and metadata inspection, allowing certain validation rules to be expressed that would otherwise be cumbersome or impossible in pure Cypher. This suggests that a practical in-database validation approach may benefit from combining declarative Cypher queries with well-scoped procedural extensions.

**RQ3: How do query-driven and pre-validation strategies compare in terms of throughput and processing time?**

The experimental evaluation shows a clear performance difference between the two validation strategies. Pre-validation performs adequately for small datasets but scales poorly as dataset size increases, resulting in rapidly growing execution times. In contrast, query-driven validation achieves substantially higher throughput and lower total processing time for large datasets. The performance advantage grows with larger datasets, showing that query-driven validation is more efficient when handling many graph elements at once. Furthermore, query-driven validation exhibits low variance across repeated runs, suggesting stable and predictable performance characteristics. These results confirm that query-driven validation is better suited for large-scale batch ingestion workflows.

Quantitatively, query-driven validation outperformed pre-validation across all dataset sizes. For small to medium datasets (SF-1 to SF-100), speedups ranged from roughly 6x to 34x, while for the largest dataset (SF-1000), the speedup reached an extreme 1863x.

## 7.2 Implications and Contributions

For practitioners and system designers, the results indicate that in-database, query-driven validation can significantly reduce validation overhead without sacrificing expressiveness. The schema catalog and query mapping together demonstrate that complex, industrially relevant constraints can be enforced without extending the database system or relying on external pre-validation tools. For researchers, the work provides an empirical complement to existing formal schema models by quantifying the performance implications of different validation strategies in a realistic setting.

Overall, the aim of the thesis has been achieved: query-driven in-database validation was shown to improve scalability while maintaining schema compliance and the approach was successfully integrated and evaluated within an industrial Neo4j pipeline.

## 7.3 Future Work

Several directions for future work emerge from the limitations of this study. First, a hybrid validation strategy combining native Neo4j constraints with query-driven validation was not implemented due to time constraints. Exploring such a hybrid approach could provide further insights into how native constraints and post-ingestion validation can complement each other in terms of performance and coverage.

Second, the experimental evaluation focused on scaling dataset size, but did not systematically examine the impact of violation density. A complementary sensitivity analysis could investigate how validation performance changes as the proportion of schema violations increases while dataset size remains fixed. Such an experiment would provide a more nuanced understanding of how different validation strategies behave under varying data quality conditions.

Third, the current prototype identifies violating nodes and edges but does not enforce corrective actions. Future work could extend the validation framework to automatically delete, quarantine or annotate invalid graph elements, thereby ensuring that production graphs become schema-compliant rather than merely reporting violations. Additionally, evaluating the cost of such remediation would be necessary to better understand the operational burden

it introduces and to assess the practical viability of post-validation strategies in production settings.

Finally, the evaluation was limited to Neo4j and batch-oriented ingestion scenarios. Extending the approach to other property graph database systems or to incremental and concurrent update workloads, would improve the generalizability of the results and further clarify the applicability of query-driven validation in continuously evolving graph systems.



## Bibliography

- [1] Renzo Angles. “A Comparison of Current Graph Database Models”. In: *2012 IEEE 28th International Conference on Data Engineering Workshops*. 2012, pp. 171–177. DOI: 10.1109/ICDEW.2012.31.
- [2] Richard Cyganiak, David Wood, Markus Lanthaler, Graham Klyne, Jeremy J. Carroll, and Brian McBride. *RDF 1.1 Concepts and Abstract Syntax*. W3C Recommendation, <https://www.w3.org/TR/rdf11-concepts/>. Feb. 2014.
- [3] Thomas M. Connolly and Carolyn E. Begg. *Database Systems: A Practical Approach to Design, Implementation and Management*. 6th. Pearson, 2014. ISBN: 978-1292061184.
- [4] Chuan Lei, Rana Alotaibi, Abdul Quamar, Vasilis Efthymiou, and Fatma Özcan. *Property Graph Schema Optimization for Domain-Specific Knowledge Graphs*. 2020. arXiv: 2003.11580 [cs.DB]. URL: <https://arxiv.org/abs/2003.11580>.
- [5] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, Filip Murlak, Stefan Plantikow, Ognjen Savkovic, Michael Schmidt, Juan Sequeda, Slawek Staworko, Dominik Tomaszuk, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Dusan Zivkovic. “PG-Schema: Schemas for Property Graphs”. In: *Proceedings of the ACM on Management of Data* 1.2 (June 2023), pp. 1–25. ISSN: 2836-6573. DOI: 10.1145/3589778. URL: <http://dx.doi.org/10.1145/3589778>.
- [6] Olaf Hartig and Jan Hidders. “Defining Schemas for Property Graphs by using the GraphQL Schema Definition Language”. In: *Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA)*. GRADES-NDA’19. Amsterdam, Netherlands: Association for Computing Machinery, 2019. ISBN: 9781450367899. DOI: 10.1145/3327964.3328495. URL: <https://doi.org/10.1145/3327964.3328495>.
- [7] Angela Bonifati, Peter Furniss, Alastair Green, Russ Harmer, Eugenia Oshurko, and Hannes Voigt. *Schema Validation and Evolution for Graph Databases*. 2019. arXiv: 1902.06427 [cs.DB]. URL: <https://arxiv.org/abs/1902.06427>.
- [8] Syed Muhammad Fawad Ali and Robert Wrembel. “From conceptual design to performance optimization of ETL workflows: current state of research and open problems”. In: *The VLDB Journal* 26.6 (Sept. 2017), pp. 777–801. ISSN: 0949-877X. DOI: 10.1007/s00778-017-0477-2. URL: <http://dx.doi.org/10.1007/s00778-017-0477-2>.

- 
- [9] Mónica Figuera, Philipp D. Rohde, and Maria-Esther Vidal. *Trav-SHACL: Efficiently Validating Networks of SHACL Constraints*. 2021. arXiv: 2101.07136 [cs.DB]. URL: <https://arxiv.org/abs/2101.07136>.
- [10] Julien Corman, Juan L. Reutter, and Ognjen Savković. “Semantics and Validation of Recursive SHACL”. In: *The Semantic Web – ISWC 2018*. Ed. by Denny Vrandečić, Kalina Bontcheva, Mari Carmen Suárez-Figueroa, Valentina Presutti, Irene Celino, Marta Sabou, Lucie-Aimée Kaffee, and Elena Simperl. Cham: Springer International Publishing, 2018, pp. 318–336. ISBN: 978-3-030-00671-6.
- [11] Martina Šestak, Marjan Heričko, Tatjana Welzer Družovec, and Muhamed Turkanović. “Applying k-vertex cardinality constraints on a Neo4j graph database”. In: *Future Generation Computer Systems* 115 (2021), pp. 459–474. ISSN: 0167-739X. DOI: <https://doi.org/10.1016/j.future.2020.09.036>. URL: <https://www.sciencedirect.com/science/article/pii/S0167739X19324094>.
- [12] Philipp Skavantzios, Kaiqi Zhao, and Sebastian Link. “Uniqueness Constraints on Property Graphs”. In: *Advanced Information Systems Engineering: 33rd International Conference, CAiSE 2021, Melbourne, VIC, Australia, June 28 – July 2, 2021, Proceedings*. Melbourne, VIC, Australia: Springer-Verlag, 2021, pp. 280–295. ISBN: 978-3-030-79381-4. DOI: 10.1007/978-3-030-79382-1\_17. URL: [https://doi.org/10.1007/978-3-030-79382-1\\_17](https://doi.org/10.1007/978-3-030-79382-1_17).
- [13] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. “The LDBC Social Network Benchmark: Interactive Workload”. In: *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. SIGMOD ’15. Melbourne, Victoria, Australia: Association for Computing Machinery, 2015, pp. 619–630. ISBN: 9781450327589. DOI: 10.1145/2723372.2742786. URL: <https://doi.org/10.1145/2723372.2742786>.
- [14] Florin Rusu and Zhiyi Huang. *In-Depth Benchmarking of Graph Database Systems with the Linked Data Benchmark Council (LDBC) Social Network Benchmark (SNB)*. 2019. arXiv: 1907.07405 [cs.DB]. URL: <https://arxiv.org/abs/1907.07405>.
- [15] Renzo Angles, János Benjamin Antal, Alex Averbuch, Altan Birler, Peter Boncz, Márton Búr, Orri Erling, Andrey Gubichev, Vlad Haprian, Moritz Kaufmann, Josep Lluís Larriba Pey, Norbert Martínez, József Marton, Marcus Paradies, Minh-Duc Pham, Arnau Prat-Pérez, David Püroja, Mirko Spasić, Benjamin A. Steer, Dávid Szakállas, Gábor Szárnyas, Jack Waudby, Mingxi Wu, and Yuchen Zhang. *The LDBC Social Network Benchmark*. 2024. arXiv: 2001.02299 [cs.DB]. URL: <https://arxiv.org/abs/2001.02299>.



## Constraint Examples

This appendix provides illustrative examples for each type of schema constraint used in the experiments. Each example includes a valid and an invalid instance.

### Node Key / Uniqueness

- **Valid (no violation):** `(:System {id: "sys-01", name: "System 1"})`
- **Invalid:** Two nodes `(:System {id: "sys-01", name: "System 1"})` and `(:System {id: "sys-01", name: "System 2"})` violate uniqueness of `id`.

### Mandatory Property

- **Valid:** `(:Service {id: "svc-01", name: "Service 1"})`
- **Invalid:** `(:Service {id: "svc-02"})`, where `name` is missing (if `name` is mandatory).

### Property Type

- **Valid:** `created: datetime("2024-01-01T00:00:00")`
- **Invalid:** `created: "2024-01-01"` (string instead of a datetime)