TU WIEN Informatics

Schema Discovery in Property Graphs Using Formal Concept Analysis

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Jakob Hitzelhammer, BSc
Matrikelnummer 11734084

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Assistant Prof. Dr.techn. Mantas Simkus, MSc

Wien, 24. März 2025

Jakob Hitzelhammer	Mantas Simkus

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

TU WIEN Informatics

Schema Discovery in Property Graphs Using Formal Concept Analysis

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Jakob Hitzelhammer, BSc
Registration Number 11734084

to the Faculty of Informatics

at the TU Wien

Advisor: Assistant Prof. Dr.techn. Mantas Simkus, MSc

Vienna, March 24, 2025

Jakob Hitzelhammer	Mantas Simkus

Technische Universität Wien
A-1040 Wien ▪ Karlsplatz 13 ▪ Tel. +43-1-58801-0 ▪ www.tuwien.at

Erklärung zur Verfassung der Arbeit

Jakob Hitzelhammer, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 24. März 2025

Jakob Hitzelhammer

Acknowledgements

First and foremost, I would like to express my gratitude to my thesis supervisor, Mantas Simkus, for his support and guidance. His insightful ideas and thoughtful feedback were invaluable in shaping this work. I truly appreciate his constant availability and willingness to accommodate meetings, always providing constructive input.

I would also like to extend my sincere thanks to my colleagues who have been by my side throughout my time at university. A special mention goes to Laurenz Stampfl and Lukas Briem, as this journey would not have been nearly as enriching and memorable without them.

Lastly, I am profoundly grateful to my family for their support and encouragement throughout my studies. Their constant belief in me and support in every endeavor have been the foundation of both my academic and personal growth.

Kurzfassung

In den letzten Jahren hat das Property-Graph-Datenmodell zunehmend an Popularität gewonnen, da es die Darstellung komplexer und stark vernetzter Daten ermöglicht. Ein wesentlicher Grund für die weitverbreitete Nutzung von Graphdatenbanken ist ihre inhärente Flexibilität. Sie erfordern in der Regel kein vordefiniertes Schema und unterliegen keiner starren Struktur. Diese Flexibilität macht sie besonders geeignet für sich schnell entwickelnde Umgebungen und unterstützt gleichzeitig eine skalierbare Datenverarbeitung. Allerdings bringt dieser schemafreie Ansatz auch verschiedene Herausforderungen mit sich. Er kann zu Inkonsistenzen führen, die Optimierung von Anfragen erschweren oder die Datenintegration komplizieren. Die manuelle Definition eines Schemas ist oft unpraktikabel, da sie umfangreiches Domänenwissen erfordert und Graphdatenbanken oft sehr groß sind. Dies führt zum sogenannten Schema-Discovery-Problem. In dieser Arbeit adressieren wir das Schema-Discovery-Problem in Property-Graphs und präsentieren eine neuartige Methode zur automatischen Schema-Extraktion auf Basis der "Formal Concept Analysis". Formal Concept Analysis ist ein mathematisches Framework, das primär für die Datenanalyse genutzt wird, dessen Werkzeuge sich jedoch auch für die Schema-Extraktion als besonders geeignet erweisen. Wir schlagen eine vollständig automatisierte Methode zur Schema-Extraktion für Property-Graphs vor und implementieren diese. Dabei nutzen wir Formal Concept Analysis, um Knoten- und Kantentypen anhand der Ähnlichkeit ihrer "Labelsünd "Propertiesßu identifizieren. Unsere Methode wird experimentell mit dem aktuellen Stand der Technik verglichen, und wir entwickeln einen Prozess zur synthetischen Graphgenerierung, um eine detaillierte Evaluierung zu ermöglichen. Darüber hinaus geht diese Arbeit auf die Herausforderung ein, Schemata für mehrere Property-Graphs zu extrahieren. Dazu wird ein Schema-Merging-Verfahren vorgestellt, das iterativ auf die extrahierten Schemata angewendet werden kann, um mehrere Graphen zu berücksichtigen.

Abstract

In recent years, the property graph data model has gained increasing popularity due to its ability to represent complex and highly interconnected data. One of the key reasons for the widespread adoption of graph databases is their inherent flexibility. They typically do not require a predefined schema and impose no rigid structure. This flexibility makes them well-suited for rapidly evolving environments while also supporting scalable data processing. However, while this schema-less approach is desirable, it also presents several challenges. It can lead to potential inconsistencies, hinder query optimization or complicate data integration. Manually defining a schema is often impractical due to the required domain knowledge and the size of graph databases, giving rise to the schema discovery problem. In this thesis, we address the schema discovery problem in property graphs and present a novel schema discovery method based on Formal Concept Analysis. Formal Concept Analysis is a mathematical framework primarily used for data analysis, and its tools naturally lend themselves to schema extraction. We propose and implement a fully automatic schema discovery method for property graphs, leveraging Formal Concept Analysis to identify node and edge types based on the similarity of their labels and properties. Our method is experimentally evaluated against state-of-the-art approaches, and we develop a process for synthetic graph generation to facilitate a more intricate evaluation. Additionally, this thesis addresses the challenge of extracting schemas from multiple property graphs by introducing a schema merging method, which can be iteratively applied to the extracted schemas to accommodate multiple graphs.

Contents

CHAPTER 1

Introduction

In recent years, property graphs have gained significant traction due to their ability to model complex and interconnected data with rich semantic information. They are extensively utilized in domains such as social and transportation networks, biological networks, logistics, finance and cybersecurity. The property graph data model effectively represents interconnected, multi-labeled data enhanced with properties defined by key/value pairs [SBV+21]. The rich formalism of property graphs allows to represent complex relationships in heterogeneous data while maintaining a compact and expressive data model. Unlike traditional relational databases, graph databases typically do not enforce strict schema constraints, making them highly flexible and adaptable to evolving data. This inherent *a priori* schemaless nature is particularly advantageous in scenarios involving large-scale and rapidly changing datasets [BDM22]. While this flexibility facilitates scalable storing of data, it poses significant challenges for various data management tasks, such as data integration, query optimization, visualization, and metadata management [SMS+20]. In these contexts, the absence of an explicit schema can hinder efficiency and usability. A well-defined schema is essential for structuring and interpreting data effectively. Manually designing an appropriate schema requires both data modeling expertise and domain knowledge, making it a time-consuming and complex task. For large-scale property graphs, manually inspecting the data to infer structural patterns is nearly infeasible. Therefore, automated schema extraction techniques are necessary to bridge this gap, providing structured representations of data while preserving the flexibility of property graphs. The need for schema discovery in graph databases extends beyond property graphs, as evidenced by the wide variety of existing methods for schema extraction [KMKT+22]. Most of these methods focus on Resource Description Framework (RDF) graphs and are not directly applicable to databases following the property graph data model. Currently, state-of-the-art schema discovery methods for property graphs comprise only three approaches [BDM22, LBH21, Lei21], all of which primarily rely on clustering techniques. Although these methods have achieved remark-

able results, they have been evaluated on relatively simple property graphs. Moreover, an important aspect of schema discovery is the detection of type hierarchies, which has not yet been successfully achieved. While Bonifati et al. [BDM22] address this issue, they provide a limited evaluation of their method's ability to capture hierarchical relationships. Moreover, none of these methods provide a solution for incremental maintenance, making it challenging to handle updates in evolving graphs. This highlights the need for further research in schema discovery for property graphs.

1.1 Solution Concept

In addition to machine learning techniques such as clustering, formal methods have also been explored for schema discovery in graph data [KMKT+22]. One such method is Formal Concept Analysis (FCA), a mathematical framework primarily used for data analysis, which has already been applied to schema extraction by Kirchberg et al. [KLT+12]. More recently, Bonifati et al. [BDM22] suggested investigating FCA for schema discovery in property graphs and comparing its effectiveness to their approach. In this thesis, we build upon this suggestion and evaluate FCA as a schema extraction method for property graphs.

A central task in schema extraction is the identification of types, which group structurally similar instances. In property graphs, this structure is defined by labels and properties. For example, a `PersonType` may represent nodes labeled "Person" that share properties such as "Name" and "Age". FCA provides a formal mechanism to analyze relationships between objects and attributes through a structured representation known as a formal context. This context is typically modeled as a cross-table, where rows correspond to objects (e.g., graph nodes), columns correspond to attributes (e.g., labels or properties), and an entry in the table indicates the presence of an attribute in a given object. By leveraging FCA, we can systematically uncover types based on how nodes and edges are associated with labels and properties.

To illustrate how FCA can facilitate schema extraction, consider the example property graph shown in Figure 1.1. For simplicity, we focus only on nodes and their labels, disregarding edges and properties. One could consider a graph with more than just four nodes, where all additional nodes are similar to the given four. These nodes may be connected by relationships such as `is_friend_of` or `works_for`. However, for readability, we limit our example to four nodes. Given this graph, we construct a formal context where nodes serve as objects and labels as attributes. Applying FCA to this context produces a set of concepts, each represented as a tuple containing a set of objects and a set of attributes. A concept ensures that all objects within it share the same attributes, and all attributes within it are present in the same objects. The resulting concepts, depicted in Figure 1.2, provide a preliminary grouping of instances based on their structural similarities.

Beyond identifying concepts, FCA also generates a concept lattice, shown in Figure 1.2, which captures hierarchical relationships among concepts. In this lattice, a subconcept's

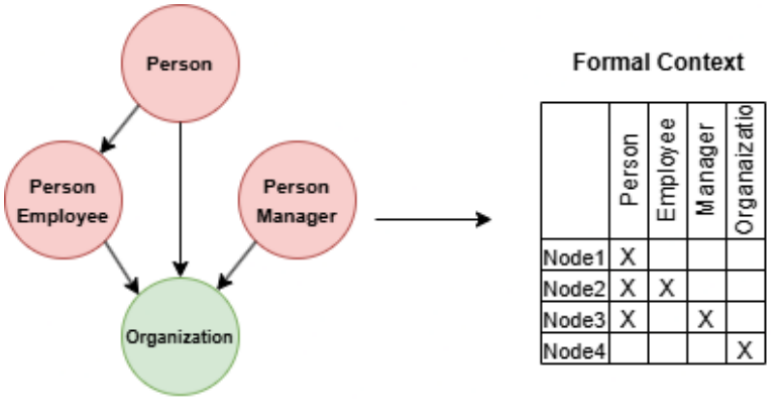

Figure 1.1: Example of a property graph with nodes and their labels and the corresponding formal context.

objects form a subset of those in its superconcept, while a superconcept's attributes form a subset of those in its subconcept. These hierarchical relationships provide valuable insights into potential subtype and supertype associations. For instance, in our example, if we consider the concepts to be types, `PersonType` could be considered a supertype of both `EmployeeType` and `ManagerType`, as these subtypes retain all the labels of `PersonType` while introducing additional distinguishing labels. This example demonstrates that the concept lattice provided by FCA constitutes a strong starting point for identifying schema-related structures.

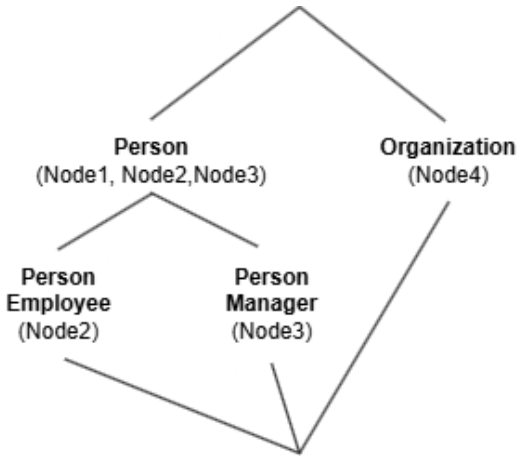

Figure 1.2: Concept lattice from the formal context of Figure 1.1.

A key advantage of FCA for schema discovery is its ability to handle overlapping types and hierarchical relationships, both crucial features, as highlighted by Kirchberg et al. [KLT+12]. Because FCA allows concepts to share objects, it naturally accommodates overlapping types. Additionally, the concept lattice structure helps identify type

hierarchies, offering a systematic way to infer inheritance relationships within the data.

However, while FCA provides a strong foundation for schema extraction, the raw concept lattice can be overly complex, making direct schema derivation impractical. To refine the extracted schema, additional post-processing steps are necessary, including outlier removal, type merging to create more general types, and type enrichment to enhance schema quality. Furthermore, FCA can be extended beyond node types to incorporate properties into the formal context, and the same methodology can be applied to extract edge types, broadening its applicability in schema discovery.

In this thesis, we implement an automatic schema discovery method based on FCA, incorporating a validation process to validate that the input graph conforms to the extracted schema. The schema produced by our method will be based on **PG-Schema**, the schema language introduced by Angles et al. [ABD+23], which represents the first proposal for a standardized schema language for property graphs. As discussed in [BDM22] and [KMKT+22], a key requirement for schema discovery methods is their ability to handle multiple and evolving graphs, i.e., datasets that change over time. To address this aspect, our method includes a schema merging process that allows the user to provide an existing schema as input. The extracted schema and the input schema will be a merged such that the result incorporates the structural elements of both schemas, ensuring that any graph valid under either of the original schemas remains valid in the merged schema. To evaluate our method, we compare its performance against state-of-the-art approaches on real-world datasets. Additionally, we implement a **PG-Schema** parser and a graph generator that constructs graphs based on a given schema. This approach enables us to systematically generate graphs with diverse schema structures and assess whether our method can accurately reproduce them.

1.2 Research Questions

This thesis is guided by the following research questions:

RQ 1: *How can FCA be effectively applied to discover schemas, including types and type hierarchies, in property graphs?*

RQ 2: *How does the performance of the FCA-based schema extraction framework compare to existing schema discovery methods in terms of accuracy and scalability?*

RQ 3: *What are the limitations of using FCA for schema discovery in property graphs?*

1.3 Thesis Structure

This thesis is structured as follows. Section 2 lays the foundation by introducing the key concepts and frameworks necessary to understand the methodology and contributions of this work. In Section 3, we review the relevant literature and state-of-the-art approaches in schema discovery, schema merging, and FCA. Section 4 provides a formal explanation

of our proposed method, which is further elaborated in Section 5, detailing its practical implementation. The effectiveness of our approach is then assessed in Section 6, where we present a comprehensive evaluation of our method. Finally, Section 7 summarizes our findings, discussing both the strengths and limitations of our approach while outlining potential directions for future research.

CHAPTER 2

Background

This Section lays the foundation for the concepts and frameworks necessary to understand the methodology and contributions of this work. It begins by introducing property graphs and then explains the specific fragment of the PG-Schema used for the proposed method. Then the concept of schema discovery, as applied in the context of this work, will be defined. Next, it examines the foundational principles of FCA, highlighting its theoretical framework and the concepts that make it a powerful tool for data analysis and organization. Finally this Section examines schema merging. This comprehensive overview ensures clarity and provides the conceptual tools required for the discussions and analyses in subsequent sections.

2.1 Property Graphs

Property graph is a type of graph data model that extends traditional graph structures by allowing both nodes and edges to hold properties, represented as key-value pairs. More specifically, a property graph is a directed labeled multi-graph that allows for properties on both nodes and edges. We follow the formal definition of a property graph as presented in [Ang18]. For the following definitions, let for a set X, 2^X denote the set of all finite subsets of X, $\boldsymbol{L}$ is an infinite set of *labels*, $\boldsymbol{K}$ is an infinite set of *property names* (keys), $\boldsymbol{V}$ is an infinite set of atomic *values*. Let $\boldsymbol{R}$ be the set of all records. A record with keys from $\boldsymbol{K}$ and values from $\boldsymbol{V}$ is a finite-domain partial function o: $\boldsymbol{K} \rightarrow \boldsymbol{V}$.

Definition 2.1.1 (Property Graph)**.** A *property graph* is a tuple $G = (N, E, \rho, \lambda, \sigma)$ where:

1. N is a finite set of *nodes*,
2. E is a finite set of *edges* such that N and E have no elements in common,

3. $\rho : E \to (N \times N)$ is a total function that associates each edge in E with a pair of nodes in N,
4. $\lambda : (N \cup E) \to 2^{\boldsymbol{L}}$ is a partial function that associates a node or edge with a set of labels from $\boldsymbol{L}$,
5. $\sigma : (N \cup E) \to \boldsymbol{R}$ is a function that maps nodes and edges to records.

Property graphs enable the representation of complex, interconnected data where entities (represented as nodes/vertices) are connected by relationships (represented as edges). Each node and edge can hold arbitrary labels and properties, offering a highly flexible and extensible way to model real-world data. For example, a node representing a "Person" might have properties such as `name`, `age`, and `address`. While an edge representing a "Friendship" might include a property like `since`, denoting when the relationship began.

The key advantage of property graphs lies in their ability to model both structure and semantics in a single framework, making them particularly well-suited for applications in social networks, knowledge graphs, and the Internet of Things, among others [AAB+17]. Unlike relational databases, where data is stored in tables with fixed schemas, property graphs allow for more dynamic, schema-less representations, making them ideal for domains where the relationships and properties between entities may evolve over time [AFB24].

For illustration, consider the following example of a property graph $G = (N, E, \rho, \lambda, \sigma)$ defined below and visually depicted in Figure 2.1, where:

- $N = \{n_1, n_2, n_3, n_4\}$
- $E = \{e_1, e_2, e_3, e_4\}$
- $\rho : e_1 \to (n_1, n_2), \quad e_2 \to (n_1, n_3), \quad e_3 \to (n_2, n_4), \quad e_4 \to (n_3, n_4)$
- $\lambda : n_1 \to$ {"Person"}, $\quad n_2 \to$ {"Person"}, $\quad n_3 \to$ {"Business"}, $\quad n_4 \to$ {"City"},
  $e_1 \to$ {"knows"}, $\quad e_2 \to$ {"works_at"}, $\quad e_3 \to$ {"lives_in"}, $\quad e_4 \to$ {"is_located"}
- $\sigma : n_1 \to$ {"name": "Peter", "age": 30}, $\quad n_2 \to$ {"name": "Sonja", "age": 42},
  $n_3 \to$ {"founded": 2002}, $\quad n_4 \to$ {"name": "Vienna", population: 2M},
  $e_1 \to$ {"since": 2013}, $\quad e_4 \to$ {"address": "First Street"}.

2.2 PG-Schema

In [ABD+23], PG-Schema was proposed as the standardized schema language for property graphs. The method developed in this thesis generates schemas that conform to the PG-Schema definition. However, due to inherent limitations in the information that can be extracted from datasets, the schemas produced by our method utilize only a subset of the PG-Schema. Consequently, our approach focuses on a specific fragment of PG-Schema, which we describe in this section.

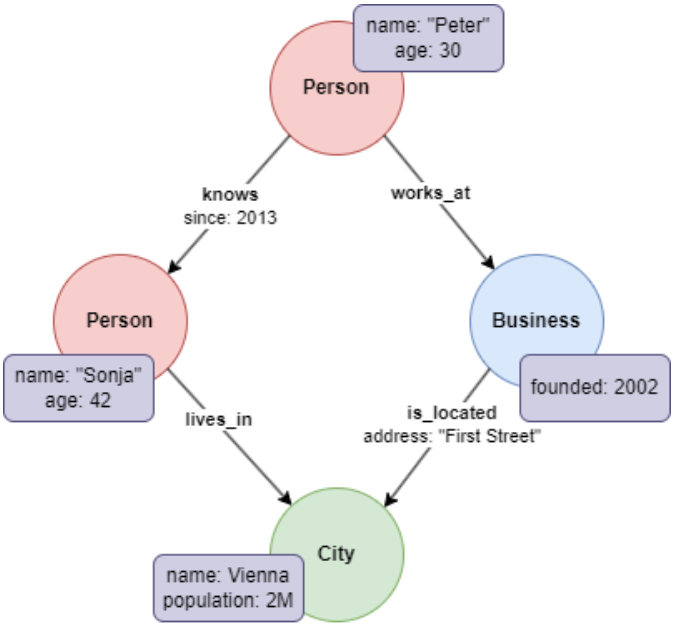

Figure 2.1: Visualization of the Example Property Graph.

PG-Schema consists of two main components:

- **PG-Types:** These define the fundamental topological structure of the graph, including node types, edge types, and their associated labels and properties.
- **Constraints:** These specify additional conditions on the property graph, such as cardinality or value constraints.

Our method outputs only the **PG-Types** component and does not include constraints, as the extraction of such detailed information is beyond the scope of the current implementation. In addition to omitting constraints, our approach restricts other features of PG-Schema as follows:

1. **Single Graph Type:** PG-Schema supports the definition of multiple graph types and allows one graph type to import another. Since our method generates a schema for a single graph instance, we limit the schema to a single graph type with no imports.
2. **Strict Type Graph:** PG-Schema allows to define a graph type as `STRICT` or `LOOSE`, which specifies how the graph should be typed against the schema. If the schema is `STRICT`, for a graph to conform to the schema, every node and edge in the graph must have at least one type it conforms to. A loose schema allows for partial validation. For this thesis, we only consider strict schemas.

3. **Simplified Label Grammar:** PG-Schema allows the use of both the &-operator (and-operator) and the |-operator for combining labels and supertypes. In our fragment, we only permit the &-operator. In this form a type can be seen as a list of possible labels and properties. The |-operator can be utilized in two distinct ways. First, it can define a *union type* of supertypes, allowing a subtype to inherit from either one of the supertypes. Second, the |-operator can serve as a *choice operator* between labels, permitting the presence of one label or the other. However this we omit the |-operator, as we cannot derive this information using our FCA-based approach. We further discuss this limitation in the conclusion 7.

Despite these restrictions, the schemas produced by our method conform to the syntax and semantics of PG-Schema as defined in [ABD+23]. The resulting schemas are:

- **Constraintless:** They include only type definitions without additional constraints.
- **Single Strict Graph Type:** They define types for nodes and edges within a single strict graph instance.
- **Without Choice Operator:** They rely exclusively on the &-operator for combining labels and supertypes.

2.2.1 Syntax and Semantic

In this section we formally define the syntax and semantic of the relevant parts of PG-Schema. For the specification of the grammar we refer to the original work [ABD+23]. We begin by outlining the syntax independent definition of a formal graph type and when a property graph conforms to a formal graph type. Next, we present the syntactical representation of a graph type. Then we provide an explanation of how the syntactical representation relates to the formal definition, establishing the semantics for the schema by detailing the connection between the two.

We first define the concept of a formal base type, which serves as the building block for graph elements such as nodes and edges.

Definition 2.2.1 (Formal Base Type)**.** Let $\boldsymbol{R}$ be the set of all records. A record with keys from $\boldsymbol{K}$ and values from $\boldsymbol{V}$ is a finite-domain partial function o: $\boldsymbol{K} \to \boldsymbol{V}$. A *formal base type* is a pair (L, R), where:

- $L \subseteq \boldsymbol{L}$ is a set of labels, and
- $R \subseteq \boldsymbol{R}$ is a set of records (property values).

We write $\mathcal{T}$ for the set of all base types. A node or edge with label set K and content o *conforms* to a formal base type (L, R) if $K = L$ and $o \in R$. Building on the formal base type, we now introduce the *formal graph type*.

Definition 2.2.2 (Formal Graph Type)**.** A *formal graph type* is a tuple $S = (N_S, E_S, \nu_S, \eta_S)$, where:

- N_S is a finite set of node type names, and E_S is a finite set of edge type names.
- $\nu_S : N_S \to 2^{\mathcal{T}}$ is a function that maps node type names to sets of formal base types.
- $\eta_S : E_S \to 2^{\mathcal{T} \times \mathcal{T} \times \mathcal{T}}$ is a function that maps edge type names to sets of triples of formal base types, specifying the source node, the edge itself, and the target node.

Building on this formalization, we now define the notion of conformance, which establishes when a property graph adheres to a given formal graph type

Definition 2.2.3 (Conformance of Graph Elements)**.** Let $G = (N_G, E_G, \lambda_G, \rho_G, \sigma_G)$ be a property graph and $S = (N_S, E_S, \nu_S, \eta_S)$ be a formal graph type.

- A node $v \in N_G$ conforms to a node type $\tau \in N_S$ if it conforms to a formal base type in $\nu_S(\tau)$.
- An edge $e \in E_G$ conforms to an edge type $\sigma \in E_S$ if for the pair $(v_1, v_2) = \rho_G(e)$, there exists a triple $(t_1, t, t_2) \in \eta_S(\sigma)$ such that v_1 conforms to t_1, e conforms to t, and v_2 conforms to t_2.

A property graph G conforms to a formal graph type S if every element in G conforms to at least one type in S.

With the conformance criteria established, we now introduce the syntactical representation of graph types. A graph type is syntactically represented by the following components:

1. **Node Types** are denoted as $(\tau : F)$, where F is an expression constructed from:

$$F = E_1 \ \& \ E_2 \ \& \ \ldots \ \& \ E_m \ [\texttt{OPEN}] \ r \, ,$$

where $E_i \in \{\ell[?], \sigma\}$, is either label $\ell \in \boldsymbol{L}$ or a node type name σ. A label is optionally followed by "?", determining if it is mandatory or optional. OPEN is an optional keyword allowing additional labels beyond those explicitly defined. The content description r is specified as:

$$r = \{[\texttt{OPTIONAL}]k_1 b_1, \ldots, [\texttt{OPTIONAL}]k_n b_n, [\texttt{OPEN}]\}$$

, where square brackets indicate optional elements, k_i are keys from K and b_i are base property types, such as `INT`, `STRING`, `DATE`, etc.

2. **Edge Types** are denoted as:

$$(: F_{\text{src}}) - [\tau : F] \rightarrow (: F_{\text{tgt}}),$$

where F_{src} and F_{tgt} represent the source and target endpoints (nodetypes). $[\tau : F]$ is syntactically defined similarly to node types, following the same structure described above with the exception that it can contain edge types names instead of node type names.

With the syntactical structure in place, we now describe the process of schema compilation. This process bridges the gap between syntax and semantics by interpreting the syntactical expressions in terms of formal base types.

Schema Compilation. We now define how the syntactical representation of node and edge types is to be interpreted. The expression F defines the set $\|F\| \subseteq \mathcal{T}$ of formal base types allowed for the corresponding type τ.

Let $t_\emptyset = (\emptyset, \{\bot\})$ and $t_l = (\{l\}, \{\bot\})$, with $\bot$ being the empty record. These are the formal base types for the empty type and a type with a single label.

We proceed by adding the content description r. Let B_i be the extent of b_i (e.g. $\mathbb{Z}$ for INT), and $\text{dom}(o)$ denote the keys for a record o. If r contains the keyword `OPEN`, then the semantics $\|r\|$ of r is the set of all records $o \in \boldsymbol{R}$, such that for every $i \leq n$, if $k_i \in \text{dom}(o)$, then $o(k_i) \in B_i$. Additionally, k_i is required to be an element of $\text{dom}(o)$, unless it is prefixed by the keyword `OPTIONAL`. In the absence of the keyword `OPEN`, it is additionally required that $\text{dom}(o) \subseteq \{k_1, \ldots, k_n\}$.

So far, the semantics of node types with a single label and a content description have been defined. For more complex types, we first have to define how to combine formal base types. For the combination of records, we first define the notion of *compatibility*. Two records $o_1, o_2 \in \boldsymbol{R}$ are *compatible* if $o_1(k) = o_2(k)$ for each $k \in \text{dom}(o_1) \cap \text{dom}(o_2)$.

The *combination* of two *compatible* records o_1 and o_2 is defined as:

$$(o_1 \oplus o_2)(k) = \begin{cases} o_1(k) & \text{for } k \in \text{dom}(o_1), \\ o_2(k) & \text{for } k \in \text{dom}(o_2) \setminus \text{dom}(o_1). \end{cases}$$

For two sets $O_1, O_2 \subseteq \boldsymbol{R}$, their *combination* $O_1 \oplus O_2$ is defined as the set of all records of the form $o_1 \oplus o_2$ for compatible $o_1 \in O_1$ and $o_2 \in O_2$. The *combination* of two formal base types is then defined as:

$$(L_1, R_1) \oplus (L_2, R_2) = (L_1 \cup L_2, R_1 \oplus R_2).$$

Now we can define the semantics recursively for all subexpressions of F as follows:

$$\begin{aligned}
\|\ell\| &= \{t_\ell\},\\
\|\sigma\| &= \nu_S(\sigma),\\
\|F_1?\| &= \|F_1\| \cup \{t_\emptyset\},\\
\|F_1 \ \&\ F_2\| &= \{(L_1, R_1) \oplus (L_2, R_2) | (L_i, R_i) \in \|F_i\| \text{ for i} = 1, 2\},\\
\|F_1 \ \texttt{OPEN}\| &= \{(L, R) \mid \exists L' \subseteq L \text{ such that } (L', R) \in \|F_1\|\},\\
\|F_1 \ r\| &= \{(L, R \oplus \|r\|) | (L, R) \in \|F_1\|\}.
\end{aligned}$$

With that the semantics for a node type τ is defined as

$$\nu_S(\tau) = \|F\|$$

. We now define the semantics for edge types. Consider an edge type defined as:

$$(: F_{\text{src}}) - [\tau : F] \rightarrow (: F_{\text{tgt}})$$

, where F_{src} and F_{tgt} specify the source and target endpoints. The node types inside the endpoint specifications are concatenated with "|", like $(F_1|F_2|...|F_n)$, and are interpreted as $\|F_1\| \cup \|F_2\| \cup ... \cup \|F_n\|$. The expression F defines the set $|\langle F\rangle| \subseteq \mathcal{T} \times \mathcal{T} \times \mathcal{T}$ of triples of formal base types by the following rules:

$$\begin{aligned}
|\langle\ell\rangle| &= \{(t_\emptyset, t_\ell, t_\emptyset)\},\\
|\langle\sigma\rangle| &= \eta_S(\sigma),\\
|\langle F_1?\rangle| &= |\langle F_1\rangle| \cup \{(t_\emptyset, t_\emptyset, t_\emptyset)\},\\
|\langle F_1 \ \&\ F_2\rangle| &= |\langle F_1\rangle| \oplus |\langle F_2\rangle|,\\
|\langle F_1 \ \texttt{OPEN}\rangle| &= \{(t_1, (L, R), t_2) \mid \exists L' \subseteq L \text{ such that } (t_1, (L', R), t_2) \in |\langle F_1\rangle|\},\\
|\langle F_1 \ r\rangle| &= \{(t_1, (L, R \oplus \|r\|), t_2) \mid (t_1, (L, R), t_2) \in |\langle F_1\rangle|\}.
\end{aligned}$$

where the $\oplus$ operator for two sets $Y_1, Y_2 \subseteq \mathcal{T} \times \mathcal{T} \times \mathcal{T}$ of triples of formal base types is defined as:

$$Y_1 \oplus Y_2 = \{(s_1 \oplus t_1, s \oplus t, s_2 \oplus t_2) \mid (s_1, s, s_2) \in Y_1, (t_1, t, t_2) \in Y_2\}\,.$$

.

The semantics of an edge type τ is then specified by $(: F_{\text{src}}) - [\tau : F] - > (: F_{\text{tgt}})$, as:

$$\eta_S(\tau) = (\|F_{\text{src}}\| \times \{t_0\} \times \|F_{\text{tgt}}\|) \oplus |\langle F\rangle|$$

.

Having established the semantics for node and edge types, we now examine an example of a syntactic representation of a graph type, the formal base types corresponding to

```
CREATE GRAPH TYPE ExampleGraphType STRICT{
  (PersonType: Person {name STRING, OPTIONAL age INT}),
  (EmployeeType: PersonType & Employee {role STRING}),
  (CityType: City {name STRING, population INT}),
  (BusinessType: Business {founded DATE}),
  (:PersonType)-[KnowsType: knows {since DATE}]→(:PersonType),
  (:EmployeeType)-[WorksType: works_at]→(:BusinessType),
  (:PersonType)-[LivesType: lives_in]→(:CityType),
  (:BusinessType)-[LocatedType: is_loacted {address STRING}]→(:CityType)
}
```

Figure 2.2: Example of a graph type.

its node and edge type definitions, and two property graphs: one that conforms to the graph type and one that does not. In Figure 2.2 we can see a quite simple example of a schema, defining four node types and 4 edge types.

Unraveling the formal base types for each of the types present in the example schema, we get the following formal base types, where we omit to explicitly repeat the formal base types for the source and target endpoints of the edge types:

- $\|\text{PersonType}\| = \{(\{\text{Person}\}, \{\{\text{name: STRING}, \text{age: INT}\}, \{\text{name: STRING}\}\})\}$
- $\|\text{EmployeeType}\| = \{(\{\text{Person}, \text{Employee}\}, \{\{\text{name: STRING}, \text{age: INT}, \text{role: STRING}\}, \{\text{name: STRING, role: STRING}\}\})\}$
- $\|\text{CityType}\| = \{(\{\text{City}\}, \{\text{name: STRING}, \text{population: INT}\})\}$
- $\|\text{BusinessType}\| = \{(\{\text{Business}\}, \{\text{founded: DATE}\})\}$
- $\|\text{KnowsType}\| = \{(\nu_S(\text{PersonType}), (\{\text{knows}\}, \{\text{since: DATE}\}), \nu_S(\text{PersonType}))\}$
- $\|\text{WorksType}\| = \{(\nu_S(\text{EmployeeType}), (\{\text{works\_at}\}, \{\bot\}), \nu_S(\text{BusinessType}))\}$
- $\|\text{LivesType}\| = \{(\nu_S(\text{PersonType}), (\{\text{lives\_in}\}, \{\bot\}), \nu_S(\text{CityType}))\}$
- $\|\text{LocatedType}\| = \{(\nu_S(\text{BusinessType}), (\{\text{is\_located}\}, \{\text{address: STRING}\}), \nu_S(\text{CityType}))\}$

Now we take a look at two property graphs depicted in Figure 2.3. The first one (left one) conforms to the schema, as every node and edge conforms to at least one type in the schema. The second property graph (right one) does not conform to the schema. The upper node is missing the mandatory property "name" to still conform to `Persontype` and the lower node, which previously conformed to `CityType`, now has a different label and therefore does not conform to any type in the schema.

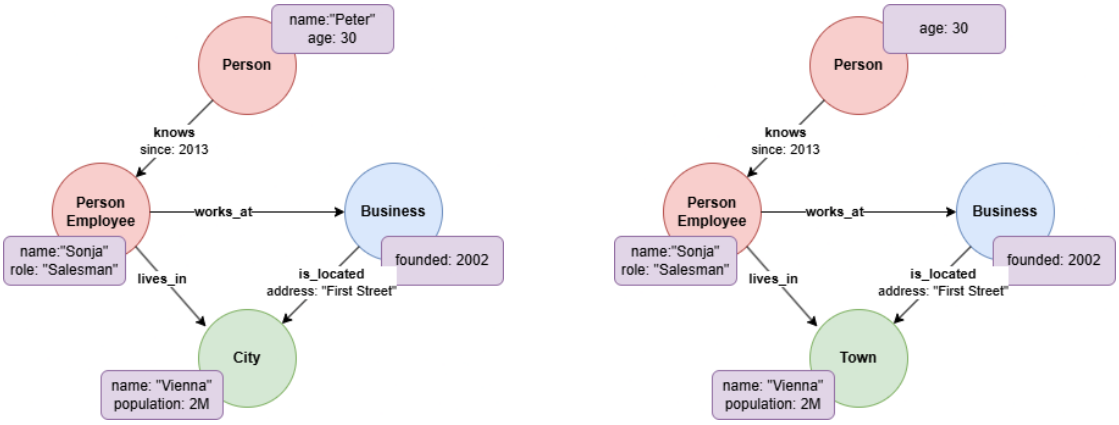

Figure 2.3: Two property graphs: the first conforms to the example graph type and the second one does not.

2.3 Schema Discovery

Schema discovery is the process of uncovering the implicit structure of data within semi-structured or schema-less formats, by identifying patterns, types, and relationships [KMKT+22]. In the context of this work, schema discovery aims to extract a schema that enables better understanding, querying, and integration of graph-based datasets. To position the method proposed in this thesis within the research field of schema discovery, we examine the characteristics of schema discovery methods outlined in [KMKT+22] and other related works. By classifying schema discovery along these dimensions, we clarify the specific type of schema discovery targeted in this thesis and its suitability for addressing the challenges of property graph schema inference. These characteristics include:

1. **Target Problem**: Schema discovery methods can address different objectives, such as:

   - **Implicit Schema Discovery**: The goal is to identify new classes or types without relying on existing schema declarations.
   - **Explicit Schema Enrichment**: The goal is to enrich existing schemas.

   The main target problem of this thesis can be classified as Implicit Schema Discovery. Formally the target problem is:

   Definition 2.3.1 (Schema Discovery Problem)**.** Given a property graph $G = (N, E, \rho, \lambda, \sigma)$, to extract a formal graph type $S = (N_S, E_S, \nu_S, \eta_S)$ such that G conforms to S.

But since our method also tries to tackle the problem of evolving graphs by incrementally building on a previous discovered schema, this work can also partly be seen as an Explicit Schema Enrichment.

2. **Underlying Techniques**: Methods leverage various techniques such Machine Learning (e.g. clustering), statistical methods or formal methods. We try to leverage FCA, a formal method, which is further described in Section 2.4.

3. **Features**: Schema discovery can further be characterized by the following features:

   - **Scalability**: This features determines the ability of a given schema extraction approach to deal with huge datasets.
   - **Incrementality**: Describes the ability to adapt schemas to dynamic datasets with insertions and deletions.
   - **Stability**: A schema discovery method is stable, if it always provides the same schema for a given input dataset.
   - **Online capability**: Describes the ability to process remote data sources.
   - **Hybrid**: A schema discovery approach is hybrid, if uses both structure of the instances and existing schema information.

   Which of these characteristics apply to this method will be discussed further in Section 6.7.

4. **Inputs**: The current implementation of our method allows for a property graph instance running in Neo4j [neo]. However, the design is flexible and can be extended to accommodate other property graph formats or databases. Furthermore, the method requires several parameters to be specified prior to execution. A comprehensive description of these parameters and their usage can be found in Section 5.1.

5. **Outputs**: Schema discovery approaches can produce various outputs. The outputs of our method are specified in 5.5:

6. **Quality Aspects**: These include for example schema relevance, schema completeness and class accuracy. The quality aspects will be further discussed in Section 6.

2.4 Formal Concept Analysis

Formal Concept Analysis (FCA) is a mathematical framework primarily used for data analysis, knowledge representation, and information retrieval. It has been successfully applied in various domains such as text mining, ontology engineering, and software engineering to uncover hidden structures and relationships within datasets [FHK+20]. The framework of FCA is fully described in [GW12]. In this section we formally define the elements of FCA that are integral to the methodology of this thesis.

In its basic form, the input data for FCA is a table (also called "cross-table") capturing the relationship between two sets: objects, which are represented by the rows, and attributes, which are represented by the columns. An entry in the table in row X and column Y indicates that object X has attribute Y. This table is called *Formal Context* and formally defined as:

Definition 2.4.1 (Formal Context)**.** A *formal fontext* is a triplet $K = (G, M, I)$, where:

- G is a non-empty and finite set of *objects*.
- M is a non-empty and finite set of *attributes*.
- $I \subseteq G \times M$ is a binary relation between G and M, indicating which objects have which attributes. If $(g, m) \in I$, we say that object g has attribute m.

Next we take a look at the fundamental building block of FCA, the formal concept. In terms of the cross-table, a formal concept can be understood as a cluster of objects, defined by attribute sharing. Formally defined as:

Definition 2.4.2 (Formal Concept)**.** A *formal concept* of a context $K = (G, M, I)$ is a pair (A, B), where:

- $A \subseteq G$ is the *extent* (the set of objects in the concept).
- $B \subseteq M$ is the *intent* (the set of attributes common to all objects in A).
- The pair (A, B) satisfies the following conditions:

$$A = \{g \in G \mid \forall m \in B, (g, m) \in I\},$$

$$B = \{m \in M \mid \forall g \in A, (g, m) \in I\}.$$

  This means A is the set of all objects sharing the attributes in B, and B is the set of all attributes shared by the objects in A.

Based on the definition of a formal concept we continue with the concept lattice based on a partial ordering of formal concepts.

Definition 2.4.3 (Concept Lattice)**.** The *concept lattice* of a formal context $K = (G, M, I)$ is a collection of all formal concepts of K, ordered by the *subconcept-superconcept* relation:

$$(A_1, B_1) \leq (A_2, B_2) \iff A_1 \subseteq A_2 \text{ (equivalently, } B_2 \subseteq B_1).$$

Key properties of the *concept lattice*:

- (A_1, B_1) is a *subconcept* of (A_2, B_2) if every object in A_1 also belongs to A_2, and the attributes of B_2 include those of B_1.

- The lattice has a *greatest concept*(supremum), representing all objects and their shared attributes.
- The lattice has a *least concept*(infimum), representing all attributes and the objects that share them.

The concept lattice provides a hierarchical visualization of the relationships between objects and attributes, which is what we want to use for finding types and type hierarchies in a property graph. There exists several algorithms on how to compute the formal concepts and the corresponding concept lattice [FHK+20]. In Section 5.3, we further discuss the algorithms that are used for our method.

For illustration we reuse the example from [GSW03] about the destinations of Star Alliance airlines. In Figure 2.4 we can see the cross-table representing the formal context, showing which airline has which destinations.

	Latin America	Europe	Canada	Asia Pacific	Middle East	Africa	Mexico	Caribbean	United States
Air Canada	X	X	X	X	X		X	X	X
Air New Zealand		X		X					X
All Nippon Airways		X		X					X
Ansett Australia				X					
The Austrian Airlines Group		X	X	X	X	X			X
British Midland		X							
Lufthansa	X	X	X	X	X	X	X		X
Mexicana	X		X				X	X	X
Scandinavian Airlines	X	X		X		X			X
Singapore Airlines		X	X	X	X	X			X
Thai Airways International	X	X		X				X	X
United Airlines	X	X	X	X			X	X	X
VARIG	X	X		X		X	X		X

Figure 2.4: Formal Context of the destinations of the Star Alliance members from [GSW03].

In Figure 2.5 we can see the resulting concept lattice. In the lattice, it becomes evident which destinations are served by nearly all airlines and which airlines operate flights to almost every destination.

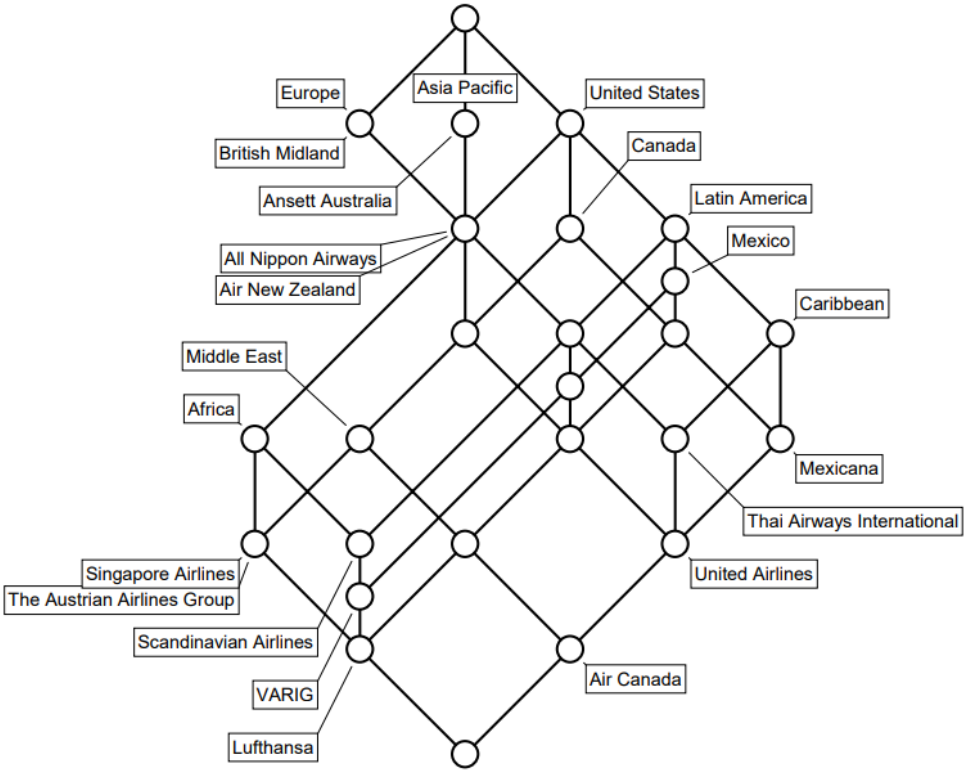

Figure 2.5: Concept lattice of the airline example from [GSW03].

2.5 Schema Merging

Schema merging is an essential operation for resolving heterogeneity between different data sources, particularly in the context of multiple or evolving data models. In this work, we extend the schema extraction process by providing the option to specify an existing schema, which is then merged with the extracted schema. This approach is designed to address the challenge of *incrementality* as discussed in Section 2.3. The core idea is to handle multiple graphs or changes in a graph by allowing users to provide an existing schema (which could have been created by our method or elsewhere) to the merging process. When a second graph is provided or the original one evolves, its new type information can be extracted and merged into the existing schema, ensuring that the schema adapts to both graphs or to the changes of the original one.

Formally we consider the Schema Merging Problem as follows:

Definition 2.5.1 (Schema Merging)**.** Given two formal graph types $S_1 = (N_{S_1}, E_{S_1}, \nu_{S_1}, \eta_{S_1})$ and $S_2 = (N_{S_2}, E_{S_2}, \nu_{S_2}, \eta_{S_2})$, the *schema merging* operation produces a new formal graph type $S_{\text{merged}} = (N_{\text{merged}}, E_{\text{merged}}, \nu_{\text{merged}}, \eta_{\text{merged}})$ such that:

$$\forall G_1 \in \mathcal{G}(S_1), G_1 \in \mathcal{G}(S_{\text{merged}}) \quad \text{and} \quad \forall G_2 \in \mathcal{G}(S_2), G_2 \in \mathcal{G}(S_{\text{merged}}),$$

where $\mathcal{G}(S)$ denotes the set of graphs that conform to the schema S. The merged schema S_{merged} is more general than the input schemas S_1 and S_2, in the sense that any graph valid under either S_1 or S_2 remains valid under S_{merged}. This ensures that the merged schema can accommodate both sets of data without violating integrity or constraints.

However, a key design goal is to avoid excessive generalization. A trivial solution would be to create a schema that allows all possible types, which would render the schema meaningless. Instead, our approach aims to construct the *least general* merged schema that still preserves validity. That is, S_{merged} should be as specific as possible while still accommodating both input schemas, ensuring meaningful type constraints and structural coherence.

To further define our schema merging approach, we draw upon principles from the field of ontology integration, as discussed in [OYD21], which are applicable to schema merging. Based on these principles, we outline the key features of our schema merging approach:

- **Schema Language**: Both schemas involved in the merging process adhere to the fragment structure defined in 2.2. This ensures that the schemas are compatible and focused on a specific subset of the overall data model, making the merging process more efficient and precise.
- **Pairwise Merging**: Our approach follows a *pairwise* schema merging strategy, where two schemas are merged into a single new schema. This is a straightforward method that enables clear comparison and alignment of the two input schemas.

- **Generalization of the Merged Schema**: One of the core requirements of our merging process is that the resulting schema must be more *general* than the input schemas. This means that any graph that is valid under Schema 1 must also remain valid under the merged schema, and the same holds for graphs valid under Schema 2.
- **Entity Similarity Based on Structure**: The similarity between entities in the two schemas is determined by the structure and the values of labels and properties, rather than relying on name-based matching. This ensures that the merging process considers the underlying semantics of the entities, avoiding potential mismatches due to differences in names.
- **Oriented Merging**: Our approach adopts an *oriented* merging strategy, where the merging is unidirectional, from a *source schema* to a *target schema*. This orientation is important because certain information, such as the names of types or classes, is taken from the input schema (the provided schema) rather than from the extracted schema. This ensures that the new schema reflects the most accurate and up-to-date information from the user-specified schema.
- **Full Merge**: According to the taxonomy in [OYD21], our merging approach is considered a *"full"* merge. This means that instead of merely providing alignments between similar entities (as in other approaches), our method generates an entirely new schema. This new schema integrates the structures and relationships from both input schemas, ensuring consistency and compatibility without simply linking similar elements.

By following these principles our schema merging approach ensures that the resulting schema is both general and that both graphs still conform to the merged schema if they conformed to the original ones. Additionally, it maintains the ability to accommodate evolving data by merging newly extracted type information with existing schema structures. This approach provides a flexible and robust solution to handle schema changes, making it suitable for dynamic and evolving graph-based datasets.

CHAPTER 3

Related Work

In the previous chapter, we laid the groundwork for this thesis by introducing the necessary preliminaries. This included defining property graphs, the PG schema and FCA, as well as clarifying the concepts of schema discovery and schema merging within the scope of this work.

Building on that foundation, this chapter examines related works. We explore research that addresses methods for schema or type discovery in graph databases, including but not limited to property graphs. Additionally, we review studies that leverage FCA for data analysis in ways similar to our approach and consider works that focus on schema merging.

3.1 Schema Discovery

In this section, we focus on studies related to schema discovery, beginning by discussing the studies that address schema extraction specifically for property graphs. Following this, we extend our discussion to studies that explore schema discovery in other graph data models.

The masters thesis by Xue Lei [Lei21] presents a framework for schema extraction in property graphs, notable for incorporating topology as a similarity measure alongside labels and properties. The framework includes a bottom-up approach, which computes attribute and topology similarities to partition objects and infer node and edge types, and a top-down approach, which starts with all objects in a single partition and iteratively refines them based on these similarities.

Lbath et al. [LBH21] propose a method based on MapReduce, which combines individual type inference (Map operation) with type reduction (Reduce operation) to merge types based on equivalence relations. Additionally, they attempt to detect node hierarchies by analyzing the inferred node types. Their approach considers either label or property

equivalence for type inference. Building on this work, Bonifati et al. [BDM22], who also contributed to the MapReduce method, introduce a hierarchical clustering approach that, according to their findings, outperforms their previous approach. This new method employs a Gaussian Mixture Model and is capable of simultaneously accounting for both label and property similarities.

Though there are relatively few studies on schema inference for property graphs, a wide range of research exists for RDF (Resource Description Framework) graphs [MMM14]. An RDF graph is represented as a labeled directed multi-graph, constructed from triples in the subject-predicate-object format. Each triple can represent either a property of a node or a relationship (edge) between two entities. This dual interpretation presents challenges unique to RDF schema inference, as opposed to property graphs, where labels and properties are explicitly associated with nodes and edges. Additionally, RDF graphs lack explicit edge types, and edges cannot have properties, as they are solely characterized by predicates. Despite these differences, both data models describe directed graphs with labels on nodes and edges, making the techniques and insights from schema inference studies in RDF graphs highly relevant. These studies provide valuable approaches and methodologies that can inform schema discovery in a broader context.

A study closely related to our work, which leverages FCA to identify concepts from RDF data, was published by Kirchberg et al. [KLT+12]. They explore the application of FCA tools and algorithms to the Semantic Web and examine whether state-of-the-art concept discovery algorithms can scale with the number of data objects retrieved from the Web. Tsuboi and Suzuki [TS19] adopt a clustering approach using KMeans++ to derive types described by Shape Expressions. Lutov et al. [LRKCM18] propose a hierarchical clustering method using a statistical type inference approach called StaTIX, which infers types from RDF data in a fully unsupervised manner.

Christodoulou et al. [CPF13] employ hierarchical clustering based on Jaccard similarity to infer types as well as hierarchical and semantic links between these types. Similarly, Chen et al. [CR14] use hierarchical clustering with Jaccard similarity but differ in allowing instances to correspond to multiple inferred types. Kellou-Menour and Kedad [KMK15, KMK16] introduce a method combining density-based clustering with Jaccard similarity to extract types along with hierarchical and semantic links from RDF data. Bouhamoum et al. [BKMLK18, BKL20] also apply density-based clustering; however, their method derives only types and does not infer hierarchical relationships.

For a comprehensive survey of schema discovery in semi-structured data, we refer to the works of Kellou-Menour et al. [KMKT+22] and Gomez et al. [GEMC18].

3.2 Formal Concept Analysis

In the field of schema discovery and ontology creation, several studies have explored the use of FCA. Razieh Mehri Dehnavis work in [Deh14] applies FCA to identify and formalize conceptual relationships within linked data to infer missing schema elements. Similarly,

Uta Priss's "Formal Concept Analysis in Information Science" [Pri06] investigates the application of FCA in uncovering hidden structures and relationships within semantic web data, emphasizing the theoretical foundations and practical applications of FCA in information science. The study in [Kri24] focuses on using FCA to create efficient and structured OWL 2 EL ontologies, highlighting the method's effectiveness in ontology axiomatization. In the healthcare domain, Cristea et al. [CSS20] demonstrate the utility of FCA in extracting and visualizing meaningful patterns from complex healthcare datasets. The studies [Haa04] and [Fu16] both propose semi-automated methods for ontology design and data integration, using FCA to streamline the process and ensure consistency. Lastly, Touzi et al. [TMA13] combine FCA with clustering techniques to automatically generate ontologies for data mining, enhancing the capability to manage and interpret large datasets. These works collectively showcase the versatility of FCA in various domains and its potential to improve schema discovery, ontology creation, and data integration through structured and formalized approaches.

FCA has also been applied extensively in the Semantic Web. For instance, Tane et al. [TCH06] utilized FCA to develop a browsing interface for ontological data, enabling both querying and visualization of the data. Maedche and Staab [MS01] incorporated FCA into an ontology-learning workbench to interactively merge conceptual structures into a unified ground truth. Cimiano et al. [CHS05] employed FCA to derive taxonomies and conceptual hierarchies from textual data. Similarly, Völker and Rudolph [VR08] used FCA in interactive ontology design to resolve underspecified logical dependencies within textual definitions of ontology axioms.

3.3 Schema Merging

Resolving semantic heterogeneity in data integration, caused by differences in formats, terminologies, structures, or, as in our case, through multiple and evolving graph databases, is a well-known and significant challenge [RAAI24]. In the context of semantic data modeling, terms such as schema, vocabularies, taxonomies, and ontologies are often used interchangeably. These concepts fall under the broader umbrella of ontologies [PHP24]. Therefore, we investigate the broader field of ontology integration, matching, and merging, understanding these methods as applicable to the matching of semantic models in general, including schemas.

One of the earliest approaches for onotlogy merging is PROMPT [NM03], which provides merging suggestions and aftereffects correspodning to the suggestions. CoMerger [BKR23] is a fully automatic method capable of handling multiple ontologies. It operates by grouping similar concepts across the input ontologies into partitions. Within these partitions, the concepts are first merged locally, followed by merging across partitions to produce a unified ontology. Lou et al. [LPWJ20] address the computational cost of similarity calculations in ontology fusion by leveraging binary metrics. Their tool, BMOnto, is a fully automated fusion method designed to merge two ontologies with a focus on scalability. Stoilos et al. [SGSK18] adopt an incremental approach, starting with

an initial seed ontology, which is iteratively enriched by incorporating new ontologies. For detailed survey on onotlogy integeration, we refer to the study of Osman et al. [OYD21]

FCA has also been applied to ontology merging. Stumme and Maedche [SM01] propose a method that constructs a formal context using instances extracted from the textual descriptions of ontologies. The contexts are then merged, and a concept lattice is generated. This lattice serves as the foundation for the manual selection of concepts and relations to form the target ontology.

An important aspect of schema merging is the method by which entities or types are matched during the integration process. Cupid [MBR01], for instance, matches entities between schemas by computing similarity based on several criteria. It evaluates the overlap of attributes, corresponding to properties in property graphs, by analyzing the syntactic similarity of attribute names, the data types of their values, and their semantic overlap, which corresponds to the overlap of labels in property graphs. COMA [DR02] extends this approach by incorporating additional criteria into its similarity computation. In particular, it also considers the actual values of properties or attributes. For a comprehensive evaluation and survey of ontology and schema matching methods, we refer readers to the works of Portisch et al. [PHP24] and Koutras et al. [KSI+21].

CHAPTER 4

Methodology

The previous sections established the foundation of this thesis by introducing the necessary concepts and frameworks, as well as discussing relevant related work. This section presents the proposed method of this thesis. We begin by outlining the schema discovery process, detailing how FCA is utilized to extract a schema from a property graph. Finally, we explore the schema merging process.

4.1 Types

The primary objective of schema extraction is to identify types that define the structure of groups of nodes or edges in a property graph. As outlined in the introduction, we define a type in a property graph based on its labels and properties, aligning with the definition used in PG-Schema 2.2. In the case of edge types, their definition additionally depends on the node types of their source and target endpoints.

One could argue that the topology of instances in a property graph also serves as a defining characteristic of types, as proposed by Lei [Lei21]. However, in our approach, we restrict the defining attributes to labels and properties. This decision is motivated by the interdependence between node and edge types. Extracting edge types requires knowledge of the corresponding node types to define endpoint types, whereas incorporating topology into node type extraction would necessitate prior knowledge of edge types. A possible solution to this challenge is an iterative approach, where node types are first extracted, followed by edge types, and subsequently refined based on the extracted edge type information. However, integrating this information would require encoding additional constraints within the PG-Schema language. Since this lies beyond the scope of our work, we leave it as an avenue for future research.

In Section 2.2 we already defined a syntactical representation of a schema, node and edge types for the PG-Schema language and their semantic interpretation onto formal base

types. However, this syntactical representation is not well-suited for implementation or for clearly explaining the method developed in this thesis. Therefore, we introduce our own notion of a property graph schema, including node and edge types, to provide a more practical and intuitive framework. Our definition closely mirrors the structure of objects used in the implementation of this method. Additionally, we present a translation from our representation to the PG-Schema language, demonstrating how the PG-Schema is derived from our model at the final stage of our method and establishing their equivalence.

Definition 4.1.1 (Property Graph Schema)**.** A Property Graph Schema is a tuple $(\mathcal{NT}, \mathcal{ET})$ with $\mathcal{NT}$ a set of node types and $\mathcal{ET}$ a set of edge types.

Definition 4.1.2 (Node Type)**.** A node type N is a tuple defined as:

$$N = (L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}, P_{\text{opt}}, S)\,,$$

where:

- $L_{\text{mand}} \subseteq \boldsymbol{L}$ is a set of (mandatory) labels that are not optional for the type T.
- $L_{\text{opt}} \subseteq \boldsymbol{L}$ is a set of labels that are optional for the type T.
- $P_{\text{mand}} \subseteq \boldsymbol{R}$ set of properties/records that are not optional for the the type T.
- $P_{\text{opt}} \subseteq \boldsymbol{R}$ set of properties/records that are optional for the the type T.
- S: Set of supertypes of $T \in \mathcal{NT}$ (node types).

Definition 4.1.3 (Edge Type)**.** An edge type E is a tuple defined as:

$$E = (L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}, P_{\text{opt}}, S, T_s, T_t)\,,$$

where:

- $L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}$ and P_{opt} are defined as for node types.
- SE: Set of supertypes of $T \in \mathcal{ET}$ (edge types).
- T_s: Set of valid source node types $T \in \mathcal{NT}$ (node types).
- T_t: Set of valid target node types $T \in \mathcal{NT}$ (node types).

In our method each of the types are associated with a unique name. Note that these definitions align with the syntactic representation of types in the PG-Schema, as introduced in Section 2.2, where we restrict types to the "&"-operator. As a result, the node and edge types defined in this section correspond precisely to the types of PG-Schema and can be interpreted accordingly. To formally establish this correspondence, we now provide a translation from our structured representation to the PG-Schema notation.

In PG-Schema, a node type is expressed as

$$(\tau : F)\,,$$

where F is an expression combining labels, supertypes, and property specifications. The translation proceeds as follows. For a nodetype of our representation

$$N = (L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}, P_{\text{opt}}, S)$$

1. Each supertype $s \in S$ is included in F as a conjunctive component. Thus, if $S = \{s_1, s_2, \ldots, s_m\}$, then the supertype expression in PG-Schema is given by:

$$s_1 \& s_2 \& \ldots \& s_m.$$

2. Each mandatory label $\ell \in L_{\text{mand}}$ is directly included in F, whereas each optional label $\ell \in L_{\text{opt}}$ is marked with ?:

$$\ell_1 \& \ell_2 \& \ldots \& \ell_k \& \ell'_1? \& \ell'_2? \& \ldots \& \ell'_{k'}?,$$

   where $L_{\text{mand}} = \{\ell_1, \ldots, \ell_k\}$ and $L_{\text{opt}} = \{\ell'_1, \ldots, \ell'_{k'}\}$.

3. The property set is expressed as a content description r, where each mandatory property $p \in P_{\text{mand}}$ is represented as kb, with k denoting the property key and b its associated base type. Optional properties $p \in P_{\text{opt}}$ are prefixed with `OPTIONAL`:

$$r = \{[\texttt{OPTIONAL}]k_1 b_1, \ldots, [\texttt{OPTIONAL}]k_n b_n, [\texttt{OPEN}]\}.$$

   If additional labels or properties beyond those explicitly defined are permitted, the keyword `OPEN` is included.

Thus, a node type N in our representation is translated into the PG-Schema expression:

$$(\tau : s_1 \& s_2 \& \ldots \& s_m \& \ell_1 \& \ldots \& \ell_k \& \ell'_1? \& \ldots \& \ell'_{k'}?\ [\texttt{OPEN}]\ \{[\texttt{OPTIONAL}]k_1 b_1, \ldots, [\texttt{OPTIONAL}]k_n b_n, [\texttt{OPEN}]\}).$$

In PG-Schema, an edge type is represented as:

$$(: F_{\text{src}}) - [\tau : F] \rightarrow (: F_{\text{tgt}}),$$

where F_{src} and F_{tgt} encode constraints on the source and target node types. The translation proceeds as follows. For an edge type of our representation

$$E = (L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}, P_{\text{opt}}, S, T_s, T_t)\,,$$

where in addition to the elements defined for node types, $T_s \subseteq \mathcal{NT}$ and $T_t \subseteq \mathcal{NT}$ define the sets of permissible source and target node types respectively:

1. Labels, supertypes, and properties are translated analogously to the node type case.
2. The sets T_s and T_t determine the valid source and target types. Each node type $\tau_s \in T_s$ contributes to F_{src}, and each $\tau_t \in T_t$ contributes to F_{tgt}.

This translation formally establishes the equivalence between our structured representation and PG-Schema notation, preserving the constraints and semantics of property graph schemas.

An essential concept for our method is the supertype relation. In the semantics of the syntactical type expressions in PG-Schema, the formal base types derived from such expressions inherit all labels and properties from the formal base types of their supertypes. To ensure that the supertype relation of our representation is also valid in PG-Schema's notion after translation, we define the relation as follows:

Definition 4.1.4 (Supertype Relation)**.** A node type $T^1 = (L^1_{\text{mand}}, L^1_{\text{opt}}, P^1_{\text{mand}}, P^1_{\text{opt}}, S^1) \in \mathcal{NT}$ is a supertype of a node type $T^2 = (L^2_{\text{mand}}, L^2_{\text{opt}}, P^2_{\text{mand}}, P^2_{\text{opt}}, S^2) \in \mathcal{NT}$ iff:

- $L^1_{\text{mand}} \subseteq L^2_{\text{mand}}$ and $P^1_{\text{mand}} \subseteq P^2_{\text{mand}}$.

An edge type $T^1 = (L^1_{\text{mand}}, L^1_{\text{opt}}, P^1_{\text{mand}}, P^1_{\text{opt}}, S^1, T^1_s, T^1_t) \in \mathcal{ET}$ is a supertype of an edge type
$T^2 = (L^2_{\text{mand}}, L^2_{\text{opt}}, P^2_{\text{mand}}, P^2_{\text{opt}}, S^2, T^2_s, T^2_t) \in \mathcal{ET}$ iff:

- $L^1_{\text{mand}} \subseteq L^2_{\text{mand}}$ and $P^1_{\text{mand}} \subseteq P^2_{\text{mand}}$.
- For every node type $N^1 \in T^2_s$, there exists a node type $N^2 \in T^1_s$, such that $N^1 = N^2$ or N^2 is a supertype of N^1.
- For every node type $N^1 \in T^2_t$, there exists a node type $N^2 \in T^1_t$, such that $N^1 = N^2$ or N^2 is a supertype of N^1.

In our approach, we initially consider all supertype relations suggested by the superconcept relations in the concept lattice. However, these relations may not always align with the semantics of PG-Schema. For node types, the extracted supertype relations from the concept lattice inherently satisfy our defined constraints. However, for edge types, we must verify whether their endpoints adhere to the supertype constraints, as the relations suggested by the concept lattice do not always guarantee compliance. Furthermore, merging types during the extraction process or in schema merging (Section 4.4) may introduce violations of these constraints. Therefore, after each merging operation, we must validate that all supertype relationships continue to satisfy the required conditions.

It is important to note that in the resulting PG-Schema, the subtype also inherits optional labels and properties of its supertype. However, the constrains that have to be satisfied for it to be a valid supertype relation are defined above.

4.2 Schema Discovery with FCA

For schema extraction, we consider a single property graph as input. In the Introduction, we provided a brief example demonstrating how FCA can be leveraged to extract types from a property graph. In that example, the formal context was constructed using nodes and labels. However, as discussed above, properties also play a crucial role in defining types, and our method must also support the extraction of edge types.

The definition of types in property graphs varies. Some graphs define types solely through labels, while others omit labels entirely, making properties the primary distinguishing characteristic. In many cases, types are determined by a combination of both labels and properties. To accommodate this diversity, we propose three different schema extraction approaches: one that relies primarily on labels, another that focuses on properties, and a third that considers both equally in type definition and differentiation.

Our schema discovery method utilizes the concept lattice derived from FCA to identify types and potential supertype relationships among them. The first step in this process is constructing the formal context. Given a property graph as input, we define the following three mappings onto the formal context. For now, we assume that the set of properties in the input graph contains no conflicts—i.e., there are no properties with the same key but different data types for their values. Practical handling of such conflicts is further discussed in Section 5.

Definition 4.2.1 (FCA-based Schema Discovery Mappings)**.** Given a *property graph* $P = (N, E, \rho, \lambda, \sigma)$, the mappings for schema discovery using FCA are defined as follows:

1. **Label-based Mapping:** A formal context $K = (G, M, I)$ where:
   - G is the set of objects, which can be either N (nodes) or E (edges).
   - $M \subseteq \boldsymbol{L}$, the set of labels associated with the objects in G.
   - $I \subseteq G \times M$, such that $(g, m) \in I$ if and only if $m \in \lambda(g)$, where $g \in G$ and $m \in M$.
2. **Property-based Mapping:** A formal context $K = (G, M, I)$ where:
   - G is the set of objects, which can be either N (nodes) or E (edges).
   - $M \subseteq \boldsymbol{K}$, the set of property keys associated with the objects in G.
   - $I \subseteq G \times M$, such that $(g, m) \in I$ if and only if $m \in \text{dom}(\sigma(g))$, where $\text{dom}(\sigma(g))$ is the domain of the record $\sigma(g)$.
3. **Label and Property-based Mapping:** A formal context $K = (G, M, I)$ where:
   - G is the set of objects, which can be either N (nodes) or E (edges).
   - $M \subseteq \boldsymbol{L} \cup \boldsymbol{K}$, the combined set of labels and property keys associated with the objects in G.

- $I \subseteq G \times M$, such that:

$$(g, m) \in I \iff \begin{cases} m \in \lambda(g), & \text{if } m \in \boldsymbol{L} \\ m \in \text{dom}(\sigma(g)), & \text{if } m \in \boldsymbol{K} \end{cases}$$

One formal context is created for the nodes and a seperate one for the edges. Each of them can be created in one of the three mappings defined above and do not have to be similar in the mapping approach. To compute the set of all concepts and their respective concept lattice, the Lindig-Algorithm [LG00] is used, resulting in one concept lattice for potential node types and one concept lattice for potential edge types. Since edge types include a definition of their source and target endpoint types, we first need to determine the node types of the schema.

4.3 Type Extraction

In this section, we describe how the concepts from a concept lattice, derived from a formal context as discussed in the previous section, are mapped onto types and how these types are further processed. To provide a structured overview of this process, we first present pseudocode outlining the key steps of our method (see Algorithm 4.1). This is followed by a detailed explanation of each step, including the additional processing required for edge types.

Given a formal concept lattice, the types are created as follows:

Initial Types: Create a node type $T = (L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}, P_{\text{opt}}, S)$ for each concept $C = (A_C, B_C)$ in the lattice, except the supremum and infimum:

- The **supremum** is only considered as a type if its intent is non-empty or there exist graph elements with no labels/properties, depending on the approach.
- The **infimum** is only considered as a type if its extent is non-empty.
- Each type is assigned a name associated with the concept id ensuring a unique name.

Concept-to-Type-Mapping: Let $IC(T)$ denote the set of instances that conform to type T. Map each concept to its corresponding node type:

- **Label-Based**: Set $L_{\text{mand}} = B_C$. The mandatory labels of T correspond to the intent of C. The concepts extent A_C are nodes and correspond to $IC(T)$.
- **Property-Based**: Set $P_{\text{mand}} = B_C$. The mandatory properties of T correspond to the intent of C. The concepts extent A_C are nodes and correspond to $IC(T)$.

Algorithm 4.1: FCA Type Extraction

Input : P - property graph,
nte - node type extraction approach (label/property/label-property),
ete - edge type extraction approach,
$allowOpt$ - allow optional labels/properties (bool),
$\tau_{outlier}$ - outlier threshold,
τ_{label} - min occurrences for label inclusion,
$\tau_{property}$ - min occurrences for property inclusion,
τ_{sim} - type similarity threshold,
τ_{abs} - abstract type merging threshold,
$mergeMax$ - limit number of types (bool),
$maxTypes$ - maximum number of types allowed.

Output : $\mathcal{NT}$ - set of extracted node types,
$\mathcal{ET}$ - set of extracted edge types.

```
// Extract Node Types
    K_n ← BuildFormalContext(P, nte) ;
    L_n ← BuildConceptLattice(K_n) ;
    NT ← ExtractNodeTypes(L_n, nte) ;
    RemoveOutliers(NT, τ_outlier) ;
    EnrichTypes(NT, P, allowOpt, τ_label, τ_property) ;
    if allowOpt then
        if mergeMax then
            MergeToMaxTypes(NT, maxTypes) ;
        end
        else
            MergeTypes(NT, τ_sim) ;
        end
    end
    FindAbstractTypes(NT, τ_abs) ;
// Extract Edge Types
    K_e ← BuildFormalContext(P, ete) ;
    L_e ← BuildConceptLattice(K_e) ;
    ET ← ExtractEdgeTypes(L_e, ete) ;
    RemoveOutliers(ET, τ_outlier) ;
    EnrichTypes(ET, P, allowOpt, τ_label, τ_property) ;
    if allowOpt then
        if mergeMax then
            MergeToMaxTypes(ET, maxTypes) ;
        end
        else
            MergeTypes(ET, τ_sim) ;
        end
    end
    ComputeEndpoints(P, ET, NT) ;
return NT, ET ;
```

- **Label-Property-Based**: Set $L_{\text{mand}} = \{\ell | \ell \in B_C \wedge \ell \in \boldsymbol{L}\}$ and $P_{\text{mand}} = \{k | k \in B_C \wedge k \in \boldsymbol{K}\}$. The concepts extent A_C are nodes and correspond to $IC(T)$.
- For all three approaches: Let T_1 and T_2 be the corresponding types for arbitrary concepts C_1 and C_2 respectively. If concept C_1 is a superconcept of C_2 in the concept lattice, i.e.,

  $$C_1 \geq C_2,$$

  then T_1 is included in the set of supertypes of T_2, formally:

  $$T_1 \in S_2.$$

  As per definiton of *subconcept-superconcept* 2.4.3, a superconcept's attributes are a subset of the subconcept's attributes. Following the mapping from a concept to a type, the superconcept relations can be exactly translated as a supertype relation between the corresponding types.

Note that, up to this point, we have only considered the keys of properties, as records themselves cannot be directly represented in the formal context. Moreover, for the purpose of distinguishing types, it suffices to use property keys under the assumption that no two records share the same key while differing in their data types. The correct data type for each key is determined separately and incorporated into the type definition after extracting the types from the concept lattice, as outlined in Section 5.2. At this stage, the list of node types could already serve as a schema. However, in most cases, it would lack expressiveness, practicality, and might not even fully conform to the input graph. Depending on the extraction approach, the schema may only capture types based on either labels or properties, leading to an incomplete representation. This limitation highlights the need for additional processing steps to refine and enhance the resulting types.

Outlier Detection: Even a single node with a unique combination of labels and properties results in its own concept and, consequently, its own node type. However, erroneous or inconsistent data in the input graph can lead to highly specific types that do not meaningfully contribute to a well-structured schema. To address this, we introduce a user-defined threshold $\tau_{outlier}$, which specifies the minimum number of nodes that must conform to a type for it to be retained in the schema. Any type with fewer conforming nodes than this threshold is removed. It is important to note that this filtering process results in a schema where the input graph does not fully conform to the schema, as nodes corresponding to the removed types will no longer have a designated type in the schema.

In the next step, each type is enriched with additional information that was not yet extracted in the **Concept-to-Type-Mapping**. To achieve this, we analyze the set $IC(T)$ for every node type $T \in \mathcal{NT}$. Labels and properties that appear in all nodes of $IC(T)$ are declared as mandatory attributes, whereas labels and properties that occur only in a subset of $IC(T)$ can be considered optional attributes. Since optional labels and properties may not always be desired in a schema, we allow the user to specify whether

optional attributes should be included. Additionally, we introduce two user-defined thresholds. τ_{label}: the minimum number of nodes in $IC(T)$ that must contain a label for it to be included in the schema. $\tau_{property}$: the minimum number of nodes in $IC(T)$ that must contain a property for it to be included in the schema.

Enrichment: For each node type $T \in \mathcal{NT}$, examine the set $IC(T)$ of nodes that conform to it:

- **Label-based Approach**:
  - Properties that are present in all nodes of $IC(T)$ are added to the set P_{mand} of mandatory properties of T.
  - If optional properties are allowed, properties that appear in more nodes in $IC(T)$ than $\tau_{property}$ are added to the set P_{opt} of optional properties of T.
- **Property-based Approach**:
  - Labels that are present in all nodes of $IC(T)$ are added to the set L_{mand} of mandatory labels of T.
  - If optional labels are allowed, labels that appear in more nodes in $IC(T)$ than τ_{label} are added to the set L_{opt} of optional labels of T.
- **Label-property-based Approach**: All labels and properties are already mapped to the types.

In most cases, the number of labels and properties results in a concept lattice that is overly granular for a direct one-to-one type mapping. This fine-grained structure may not align with the desired level of abstraction for defining types. Therefore, it is essential to identify types that share similar attributes and could potentially be merged.

For example, consider a group of nodes that exhibit nearly identical attributes. Ideally, we would identify a single type to represent them. However, some nodes may possess additional attributes not found in others. While these attributes could be treated as optional, the concept lattice initially assigns each distinct attribute to a separate concept, leading to an unnecessarily fragmented type structure.

To address this, if two types derived from the lattice share, for instance, 70% of their attributes, they can likely be merged into a more general type. The attributes that are not shared between them can then be designated as optional, ensuring a more practical and generalizable schema.

The computation of similarity of two types is described in detail in Section 4.3.1. It is important to note that merging types is only possible if the user has allowed optional labels and properties. This is because merging two types may result in some mandatory labels and properties becoming optional in the resulting type.

Merge Types: Our method generally first looks for similar types along the sub/supertype relations. As merging along these relations, makes it easier to preserve topological information across the types, see 4.3.2. The method looks for the best pair of sub/supertype related types based on their similarity score (see simliarity of types 4.3.1). If the score exceeds the predefined threshold τ_{sim}:, the types are merged. This iterative process continues until no pairs remain or the best similarity score falls below the threshold. Users can also specify constraints on the resulting schema by setting a maximum number of node types and edge types. If these constraints are active, the method prioritizes merging types with no subtypes but existing supertypes, progressing bottom-up along the lattice from the most specific types to the most general ones. The most similar pair within this subset of sub/supertype-related types is selected and merged. If no further sub/supertype-related pairs remain, the method proceeds to merge the overall most similar pairs of types until the schema meets the specified maximum type limits.

In the last step of the type extraction, all labels and properties of each type that are inherited are removed from the a type definition before translating it to PG-Schema, ensuring that there are no redundant specifications that are already defined in a supertype. After this process the node type extraction is finished. Having extracted the node types succesfully the method continues with discovering edge types. In this process it is necessary to already have the node types as we also have to identify the enpoint types of an edge, which are based on the node types.

Edge Type Extraction: The process of extracting edge types begins with computing a concept lattice, following the same three approaches available for node type extraction: label-based, property-based, or label-property-based. It is worth noting that the approaches for node and edge type extraction can differ. For instance, the node concept lattice might be label-based while the edge concept lattice is property-based. The extraction of edge types follows the same process described for node types. Each edge type is initially associated with a concept from the lattice, enriched with labels and properties according to the chosen approach, and may get merged with other edge types. However, an additional step is required to determine the endpoint types (source and target node types) of the edge types. For this, the set $IC(T)$ of edges that conform to an each edge type T (i.e. the edges classified as this type in the input graph) is analyzed. The algorithm iterates through these edges, examining their source and target nodes. Since node types are already known from the node type extraction process, the following procedure is applied for each edge type T:

1. For each source node type $N \in \mathcal{NT}$ present in the edges of $IC(T)$, count the number of edges that originate from nodes of this type.
2. If the count for a specific source node type $N \in \mathcal{NT}$ exceeds a predefined threshold, this node type is added as a source node type for T.
3. Repeat the same procedure for the target nodes of the edges to determine the target node types.

After determining the endpoint types, we must verify whether the supertype relations remain valid under the constraints defined in 4.1.4. If any relation fails to satisfy these constraints, it is removed. This step concludes the type extraction process in our method, having successfully inferred the node types and edge types that constitute a Property Graph Schema $(\mathcal{NT}, \mathcal{ET})$ from the input graph by leveraging the constructed concept lattices.

4.3.1 Similarity of Types

In previous works for property graph schema discovery, the Dice Metric and the Jaccard similarity were popular choices to compute the similartiy between types or graph entities ([BDM22], [Lei21]). In our method the similarity is calculated as follows:

The Jaccard similarity between two node or edge types T^1 and T^2 is computed as:

$$S(T^1, T^2) = \frac{\sum_{X \in \{L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}, P_{\text{opt}}\}} \left(\frac{|X^1 \cap X^2|}{|X^1 \cup X^2|} \times |X^1 \cup X^2| \right)}{\sum_{X \in \{L_{\text{mand}}, L_{\text{opt}}, P_{\text{mand}}, P_{\text{opt}}\}} |X^1 \cup X^2|}$$

where for each set X, X^1 and X^2 represent the respective sets in T^1 and T^2.

The weighted Jaccard similarity metric was chosen as it delivered the most satisfactory results in the experiments conducted. Future work could include exploring alternative metrics and their variations. However, it is important to note that the performance of similarity metrics is inherently dependent on the data of the input graph. Consequently, the effectiveness of a given metric is influenced by the nature of the input dataset, including the variability and distribution of labels and properties.

Currently, the similarity computation between two types is based solely on their labels and properties. Additional topological information could be considered for the similarity calculation, such as:

1. **Neighborhood Information:** Analyzing adjacent nodes to capture shared context or structural patterns within the graph. This would help distinguish types that appear similar in attributes but differ in their connectivity.
2. **Incident Edges:** Considering the types and properties of edges connected to the nodes could enrich the representation of node types.
3. **Edge Endpoint Types:** For edge types, similarity computation could also take into account the types of their source and target nodes. This would reflect the semantic roles of edges more comprehensively.

4.3.2 Type Merging

When merging two types, it is crucial to ensure that the resulting type does not change the schema such that the input graph does not conform to it anymore. This means

that the new type must be more general, ensuring that every node or edge previously conforming to type T^1 or T^2 also conforms to the newly merged type T^3. To achieve this, mandatory labels and properties can remain mandatory only if they were mandatory in both types, otherwise, they become optional.

Formally, if two node types $T^1 \in \mathcal{NT}$ and $T^2 \in \mathcal{NT}$ are merged into a new node type $T^3 \in \mathcal{NT}$, then:

$$\begin{aligned} L^3_{\text{mand}} &= L^1_{\text{mand}} \cap L^2_{\text{mand}}\,, \\ L^3_{\text{opt}} &= L^1_{\text{opt}} \cup L^2_{\text{opt}} \cup (L^1_{\text{mand}} \Delta L^2_{\text{mand}})\,, \\ P^3_{\text{mand}} &= P^1_{\text{mand}} \cap P^2_{\text{mand}}\,, \\ P^3_{\text{opt}} &= P^1_{\text{opt}} \cup P^2_{\text{opt}} \cup (P^1_{\text{mand}} \Delta P^2_{\text{mand}})\,, \end{aligned}$$

where Δ denotes the symmetric difference.

For edge types $T^1 \in \mathcal{ET}$ and $T^2 \in \mathcal{ET}$ the merging process follows the same rules as for node types. Additionally, the source and target node type sets are combined to preserve all possible connections:

$$\begin{aligned} T^3_s &= T^1_s \cup T^2_s\,, \\ T^3_t &= T^1_t \cup T^2_t\,. \end{aligned}$$

This ensures that the newly merged edge type T^C retains the connectivity characteristics of the original edge types T^1 and T^2.

When merging two types, we must consider the impact on the sub/supertype relationships within the schema. We differentiate two cases:

1. Merging Along Sub/Supertype Relations: When merging two types T^1 and T^2 where T^1 is a supertype of T^2, forming a new type T^3, we get:

- Since T^1 is a supertype of T^2, its mandatory labels and properties are a subset of those of T^2:
$$L^1_{\text{mand}} \subseteq L^2_{\text{mand}}, \quad P^1_{\text{mand}} \subseteq P^2_{\text{mand}}\,.$$
- Since mandatory attributes of T^3 are defined as their intersection:
$$L^3_{\text{mand}} = L^1_{\text{mand}} \cap L^2_{\text{mand}}\,, \quad P^3_{\text{mand}} = P^1_{\text{mand}} \cap P^2_{\text{mand}}\,,$$
it follows that:
$$L^3_{\text{mand}} = L^1_{\text{mand}}, \quad P^3_{\text{mand}} = P^1_{\text{mand}}\,.$$

Thus, T^3 maintains the same mandatory attributes as T^1, preserving its sub/supertype relations. Consequently:

- Any supertype of T^1 remains a supertype of T^3.

- Any type that previously had T^1 as its supertype now has T^3 as its supertype instead.
- By transitivity, any type that had T^2 as its supertype (and thus, indirectly T^1) now has T^3 as its supertype.
- However, supertypes of T^2 may no longer remain supertypes of T^3. This topological information is effectively "lost" during the merging process.

2. Merging Non-Related Types: When merging two unrelated types T^1 and T^2 into a new type T^3, the following conditions hold:

- Any subtype of T^1 remains a subtype of T^3.

  For any subtype of T^1 to remain a subtype of T^3, the mandatory attributes of T^3 must be a subset of those of the subtype. Since mandatory attributes are defined as the intersection:

  $$L^3_{\text{mand}} \subseteq L^1_{\text{mand}}, \quad P^3_{\text{mand}} \subseteq P^1_{\text{mand}},$$

  and since any subtype of T^1 must have at least the same mandatory attributes as T^1, it follows that:

  $$L^3_{\text{mand}} \subseteq L^1_{\text{mand}} \subseteq L_{\text{mand}}(\text{subtype of } T^1),$$

  $$P^3_{\text{mand}} \subseteq P^1_{\text{mand}} \subseteq P_{\text{mand}}(\text{subtype of } T^1).$$

  Thus, all subtypes of T^1 remain valid subtypes of T^3.
- Any subtype of T^2 remains a subtype of T^3 (by the same argument).
- The supertypes of T^1 and T^2 cannot be guaranteed to also be supertypes of T^3.

To demonstrate the impact of different merging strategies, we again consider the example from the introduction where the graph contains nodes representing three types: `PersonType`, `EmployeeType`, and `ManagerType`. In addition to their inherited attributes, both `EmployeeType` and `ManagerType` have their own distinct properties, including a role-specific ID (`employee_id` for employees and `manager_id` for managers), as well as optional properties such as an email address and a work location.

A possible type hierarchy, derived from the concept lattice of this example, is shown in Figure 4.1, where the topmost type represents `PersonType`, the bottom-left types correspond to `EmployeeType`, and the bottom-right types correspond to `ManagerType`. This hierarchy is constructed using the label-property-based approach, resulting in five distinct types. This occurs because FCA generates separate concepts for nodes that contain optional properties. The supertype relationships are indicated by edges, with edges pointing downward from more general types (supertypes) to more specific types (subtypes).

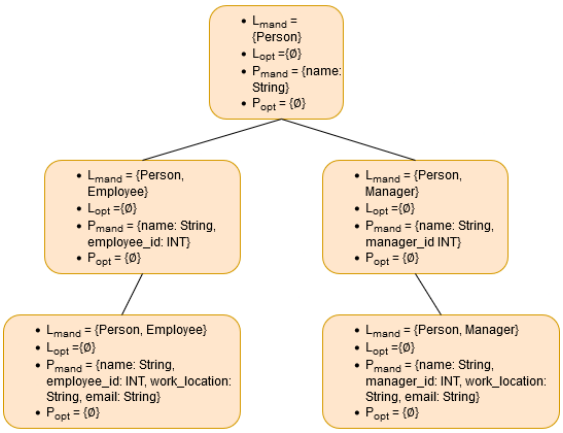

Figure 4.1: Hierarchical type representation of initial types extracted from a concept lattice before merging.

If we merge purely based on similarity, without considering the hierarchy, the two most similar types, the two types at the bottom of the hierarchy, would be merged first. This results in a new type where the previously mandatory labels (`Employee`, `Manager`) and mandatory properties (`employee_id`, `manager_id`) become optional. Otherwise, instances of these types would not conform to any type in the schema. However, since these attributes are no longer mandatory, the supertype relations that previously held the hierarchy together become invalid. As a consequence, the hierarchical structure is lost, and we end up with an isolated, generic type that does not accurately reflect the data structure. This issue is illustrated in Figure 4.2.

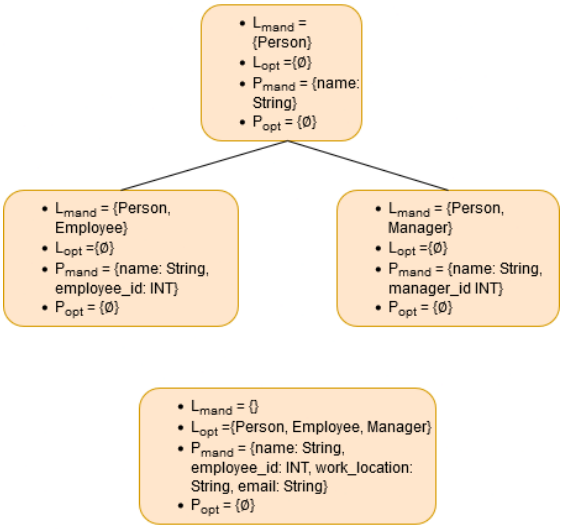

Figure 4.2: Hierarchical type representation after merging non-related types.

Alternatively, if we prioritize merging along existing supertype-subtype relations, we

first merge the two bottom-most types into their respective supertypes. Since each of the bottom types inherits from a different intermediate type, merging them directly into their supertypes maintains the relationships between the types while avoiding the creation of an isolated type with weakened constraints. As a result, the final schema includes the three node types that accurately represent `PersonType`, `EmployeeType`, and `ManagerType`. The mandatory labels and properties remain properly assigned, and all instances continue to conform to a well-defined type structure. This approach avoids the loss of important type relationships and maintains the integrity of the type system, as depicted in Figure 4.3.

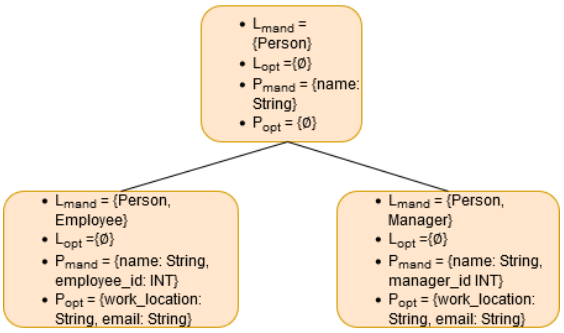

Figure 4.3: Hierarchical type representation after merging related types.

To preserve as much information as possible about the hierarchical structure of the schema, our method prioritizes merging types along existing supertype relationships.

4.3.3 Identify Abstract Types

Within the FCA framework, our method identifies only those types that have actual instances present in the data. Consider the example used in the introduction 1.1, where the graph contains nodes corresponding to the types `PersonType`, `EmployeeType`, and `ManagerType`. In many cases, data models include abstract types, types that do not have direct instances in the dataset. For instance, a graph may contain only instances of `EmployeeType` and `ManagerType`, while `PersonType` exists solely as an abstraction to group shared attributes. Since no node explicitly represents `PersonType` in the data, our method would not detect it, resulting in `EmployeeType` and `ManagerType` being treated as independent types without a hierarchical relationship, even if they share common labels and properties.

To address this limitation, data models often introduce abstract types to encapsulate shared attributes, thereby allowing for a structured hierarchy where new types can inherit these common characteristics. To facilitate the discovery of such abstract types, our method includes an optional mechanism for their identification. This process involves computing the similarity between all detected types using the metric defined in Section 4.3.1. If the most similar pair of types exceeds the predefined similarity threshold τ_{abs}:, a new abstract type is inferred based on their shared attributes.

Formally, if an abstract type T^3 is created from two types T^1 and T^2, then:

$$\begin{aligned} L^3_{\text{mand}} &= L^1_{\text{mand}} \cap L^2_{\text{mand}}, \\ L^3_{\text{opt}} &= L^1_{\text{opt}} \cap L^2_{\text{opt}}, \\ P^3_{\text{mand}} &= P^1_{\text{mand}} \cap P^2_{\text{mand}}, \\ P^3_{\text{opt}} &= P^1_{\text{opt}} \cap P^2_{\text{opt}}, \\ S^1 &= S^1 \cup T^3, \\ S^2 &= S^2 \cup T^3. \end{aligned}$$

The abstract type T^3 is then added to the schema as a higher-level representation that generalizes T^1 and T^2. This process is repeated until no pair of types is found whose similarity exceeds the threshold.

4.4 Schema Merging

In scenarios where multiple instances of property graphs exist, such as different snapshots of the same graph over time, it becomes essential to derive a general schema that accommodates all instances. Our method addresses this challenge by allowing an existing schema to be provided as input to the discovery process.

The method first extracts a schema from the given graph instance and then merges it with the provided input schema. By applying this process iteratively to multiple graph instances, a schema that fits all instances can be constructed. When an additional schema (referred to as the "original schema") is supplied, it is merged with the newly extracted schema (referred to as the "new schema"). Both schemas are represented as lists of node types and edge types. The merging process identifies corresponding types across schemas and combines them while ensuring structural validity and hierarchical consistency.

For each type T^1 in the original schema, the most similar type T^2 in the new schema is identified based on a similarity score $S(T^1, T^2)$, as defined in Section 4.3.1. If the similarity score is greater or equal than a user defined threshold τ_{sim2}:

$$S(T^1, T^2) \geq \tau_{sim2},$$

then the types are merged into a new type T^3 according to the merging rules detailed in Section 4.3.2. If no sufficiently similar type is found, the type from the original schema is directly added to the merged schema without modification. Likewise, types in the new schema that do not have a corresponding match in the original schema are also included in the final schema unchanged. This process is depicted in Algorithm 4.2

Schemas may include types with attributes specifying whether their labels and properties are **open** or **closed**. Since these attributes cannot be automatically inferred in an extracted schema, they must be specified by the user. During merging, the following rule applies:

Algorithm 4.2: Merge Types List

Input : $\mathcal{T}^1, \mathcal{T}^2$ - lists of types from two schemas,
τ_{sim2} - type similarity threshold.
Output : $\mathcal{T}^3$ - merged list of types.

$\mathcal{T}^3 \leftarrow \emptyset$;
foreach $T^1 \in \mathcal{T}^1$ **do**
  $T^2 \leftarrow \arg\max_{T \in \mathcal{T}^2} S(T^1, T)$ `// Find most similar type`
  if $S(T^1, T^2) \geq \tau_{sim2}$ **then**
    $T^3 \leftarrow \text{MergeTypes}(T^1, T^2)$ `// Merge according to rules in Section 4.3.2`
    $\mathcal{T}^3 \leftarrow \mathcal{T}^3 \cup \{T^3\}$
    $\mathcal{T}^2 \leftarrow \mathcal{T}^2 \setminus \{T^2\}$
  end
  else
    $\mathcal{T}^3 \leftarrow \mathcal{T}^3 \cup \{T^1\}$ `// No match, keep original type`
  end
end
$\mathcal{T}^3 \leftarrow \mathcal{T}^3 \cup \mathcal{T}^2$ `// Include unmatched types`
return $\mathcal{T}^3$

- If both types T^1 and T^2 have **closed** labels or properties, the merged type T^3 also has closed labels/properties.
- Otherwise, T^3 adopts **open** labels/properties to ensure schema generality.

For example, assume:

- T^1 has **closed labels** and a mandatory label X,
- T^2 has **open labels** and also contains X.

If T^3 were to adopt **closed labels**, a node with labels $\{X, Y\}$ would conform to T^2 but not to T^3. To ensure that the merged schema remains **more general** and maintains compatibility, T^3 adopts **open labels**.

At the conclusion of the merging process, the hierarchical structure of the merged schema must be updated to ensure consistency. Specifically, some supertype relations may become invalid due to the merging process, as discussed in Section 4.3.2, in the case of merging unrelated types. During the schema merging process, we initially set the supertype set of the merged type as follows:

$$S^3 = S^1 \cup S^2.$$

Subsequently, each supertype relation is verified to ensure that it satisfies the necessary conditions for a valid supertype relationship (see Definition 4.1.4). If any relation violates these conditions, it is removed to maintain schema consistency.

In addition to verifying existing supertype relations, we also check for potential new supertype relationships that may emerge through the merging process. Consider the following scenario. Suppose we are merging schemas derived from two property graphs. The first graph results in a schema that includes a type `PersonType`, which has multiple subtypes, such as `EmployeeType` and `ManagerType`. The second graph does not explicitly contain a `PersonType`, but it introduces a new type `InternType`, which possesses all the labels and properties of `PersonType`, along with some additional ones. Ideally, when merging the schemas, we want to detect that `InternType` is in fact a subtype of `PersonType` and establish this relationship in the merged schema. To achieve this, we systematically evaluate all pairs of types in the merged schema to determine whether they satisfy the necessary conditions for a valid supertype-subtype relationship, as defined in Definition 4.1.4. If these conditions are met, the corresponding relation is added to the schema. This additional step ensures that the hierarchical structure remains well-defined and accurately reflects the relationships present in the merged schema. The complete schema merging process is formally outlined in Algorithm 4.3.

Algorithm 4.3: Schema Merging

Input : $\mathcal{NT}^1, \mathcal{ET}^1$ - node and edge types from the original schema,
$\mathcal{NT}^2, \mathcal{ET}^2$ - node and edge types from the new schema,
τ_{sim2} - type similarity threshold.
Output : $\mathcal{NT}^3, \mathcal{ET}^3$ - merged node and edge types.

// **Merge Node Types**
$\mathcal{NT}^3 \leftarrow$ MergeTypesList$(\mathcal{NT}^1, \mathcal{NT}^2, \tau_{sim2})$;
// **Merge Edge Types**
$\mathcal{ET}^3 \leftarrow$ MergeTypesList$(\mathcal{ET}^1, \mathcal{ET}^2, \tau_{sim2})$;
// **Update Type Hierarchy**
foreach $T^3 \in (\mathcal{NT}^3 \cup \mathcal{ET}^3$ **do**
  foreach $S \in S^3$ **do**
    if *!IsValidSupertypeRelation*(S, T^3) **then**
      $S^3 \leftarrow S^3 \setminus \{S\}$ // Remove invalid supertypes
    end
  end
end
// **Infer New Supertype Relationships**
foreach $T^a, T^b \in \mathcal{NT}^3, T^a \neq T^b$ **do**
  if *IsValidSupertypeRelation*(T^a, T^b) **then**
    AddSupertypeRelation(T^a, T^b) ;
  end
end
foreach $T^a, T^b \in \mathcal{ET}^3, T^a \neq T^b$ **do**
  if *IsValidSupertypeRelation*(T^a, T^b) **then**
    AddSupertypeRelation(T^a, T^b) ;
  end
end
return $\mathcal{NT}^3, \mathcal{ET}^3$;

CHAPTER 5

Implementation

This section outlines the relevant implementation details of our schema discovery approach. It begins by specifying the inputs to the method, including supported formats for property graphs and the configurable parameters that guide its operation. Next, we outline the functionality of our graph generator, which constructs synthetic graphs from a provided schema, enabling us to test custom schemas and further evaluate our method. Then we explain how we deal with data type inconsistencies in the values of properties. Next, we describe the FCA library to uncover the topological and type-related structures within the graph. Furthermore, we discuss the validation phase, where the input graph is checked for conformance against the extracted schema. Lastly, we describe all the outputs our method produces.

5.1 Input

This section begins by discussing the property graph formats supported by the method. Next, we describe the functionality of our graph generator, which is designed to create synthetic graphs based on a given schema. Following this, the parameters of the method are specified in detail, highlighting their roles, configurations, and how they influence the functionality and output of the method.

5.1.1 Input Property Graph

The property graph model is a versatile framework for organizing graph data, implemented in various systems across different database paradigms. One prominent implementation of the property graph model is Neo4j, which is widely used for research in property graphs. Since the datasets used to evaluate existing approaches (as discussed in Section 3.1) are based on Neo4j, our method supports Neo4j as an input format.

While Neo4j model restricts edges to a single label, our method supports multiple labels on edges. Importantly, the method is designed to be easily adaptable to other input formats. To achieve this, the Neo4j data graph is queried to extract all nodes, edges, and their associated labels and properties, which are then converted into an internal data graph structure. For extending the method to new formats, only this translation step needs to be adapted to feed data into the internal structure. Once integrated, the core processing pipeline remains unchanged, enabling compatibility with various property graph implementations.

5.1.2 Graph Generator

The input to our graph generator is a schema, as outlined in Section 2.2. Accordingly, the graph generator requires a parser to interpret the schema. The implemented parser is designed solely for schemas adhering to the format described in Section 2.2. It is intended for testing purposes only and does not fully validate the schema or provide detailed error messages if the schema is invalid.

For each type defined in the input schema, the graph generator produces a number of nodes or edges ranging between `graph_generator_min_entities` and `graph_generator_max_entities`. Nodes and edges are assigned all mandatory labels of their respective type definition, while optional labels are assigned based on a 50:50 random selection. Properties assigned in a similar way and receive random values consistent with their respective data types.

The generation process proceeds in two steps: First, nodes for each node type are created. Then the edges are created and based on the specified end-node types in the schema, suitable source and target nodes are selected randomly from the pool of previously generated nodes. The result is a graph that conforms to the provided schema and represents the defined structure of the schema.

5.1.3 Parameters

This section provides a detailed description of the configuration file fields used in the method. Each field's purpose, allowed values, and behavior are explained below.

Data Source

- `data_source`[**str**]: Specifies the source of the data. Currently, only `neo4j` is supported.
- `neo4j.uri`[**str**]: URI for connecting to the Neo4j database.
- `neo4j.username`[**str**]: Username for authentication.
- `neo4j.password`[**str**]: Password for authentication.

Graph Generator Settings

- `graph_generator`[**bool**]: Enables or disables graph generation mode (`true`/`false`). When set to true, the method ignores the data source given and tries to generate a graph based on the schema given as input.
- `graph_generator_schema_path`[**str**]: Path to the schema file used for graph generation. The graph generator creates a graph based on the schema given in this file.
- `graph_generator_max_entities`[**int**]: Maximum number of entities to generate per type in the schema (positive integer).
- `graph_generator_min_entities`[**int**]: Minimum number of entities to generate per type in the schema (must be less than or equal to `max_entities`).

Schema Extraction Settings

- `node_type_extraction`[**str**]: Method for extracting node types. Allowed values: `label_based`, `property_based`, `label_property_based`.
- `edge_type_extraction`[**str**]: Method for extracting edge types. Allowed values: `label_based`, `property_based`, `label_property_based`.
- `optional_labels`[**bool**]: When set to true, the resulting schema includes optional labels.
- `optional_properties`[**bool**]: When set to true, the resulting schema includes optional properties.
- `type_outlier_threshold`[**int**]: Determines the minimum number of nodes or edges that must correspond to a type for it to be considered valid. If the count of nodes or edges associated with a type falls below this threshold, the type is classified as an outlier. Such types are assumed to represent false, erroneous, or insufficient data and are consequently removed from the schema.
- `label_outlier_threshold`[**int**]: Specifies the minimum absolute number of nodes or edges of a type that must contain a label for it to be considered valid. Labels appearing in fewer elements than this threshold are classified as outliers and excluded from the schema.
- `property_outlier_threshold`[**int**]: Defines the minimum absolute number of nodes or edges of a type that must include a property for it to be considered valid. Properties appearing in fewer elements than this threshold are regarded as outliers and removed from the schema.

- `endpoint_outlier_threshold`[**int**]: Specifies the minimum absolute number of edges of a specific type that must involve a source or target node type for the endpoint type to be valid. Source or target node types appearing in fewer edges than this threshold are treated as outliers and omitted from the schema.
- `merge_threshold`[**float**]: Similarity threshold for merging types during type extraction.
- `remove_empty_types`[**float**]: If true, types that have no nodes/edges assigned to them will be removed.
- `max_node_types`[**int**]: Maximum number of node types in the resulting schema.
- `max_edge_types`[**int**]: Maximum number of edge types in the resulting schema.
- `max_types`[**bool**]: Enables type merging to conform to the maximum number of allowed types specified in `max_node_types` and `max_edge_types`.
- `abstract_type_threshold`[**float**]: Threshold for the similarity score of two types such that an abstract type of their shared attributes will be created.
- `abstract_type_lookup`[**bool**]: Enables lookup for abstract types during node type extraction.
- `graph_type_name`[**str**]: Name of the graph type (any valid string).
- `out_dir`[**str**]: Directory to save the results to.

Validation Settings

- `validate_graph`[**bool**]: Enables graph validation against the schema.
- `open_labels`[**bool**]: Our method lets the user define if labels should be open for the types in the resulting schema. If a type has open labels it permits labels not defined in the schema during validation (see 5.4). Consequently all types in the resulting schema will have a open label declaration.
- `open_properties`[**bool**]: Our method lets the user define if properties should be open for the types in the resulting schema. If a type has open properties it permits properties not defined in the schema during validation (see 5.4). Consequently all types in the resulting schema will have a open properties declaration.

Schema Merging Settings

- `merge_schema`[**bool**]: Enables schema merging functionality.
- `schema_to_merge_path`[**str**]: Path to the schema file to merge with the extracted schema.

- `schema_merge_threshold`[**float**]: Similarity threshold for merging entities during schema merging.

Schema that are provided as input, for example for schema merging or for the graph generator, have to be defined in the PG-Schema language, as described in Section 2.2.

5.2 Data Type Inference

When data is successfully retrieved from the data source, the first step in this process is to infer the datatypes of the properties. Each property can be viewed as a unique record, and its datatype should be consistent across all its occurrences.

To determine the datatype for a property, the process examines all occurrences of a given property key or name. For each instance, the value is checked to identify whether it is an `DATE`, `STRING`, `INT` or any other supported datatype. All datatypes specified in the PG-Schema are supported for inference. A count is maintained for each identified datatype, and the datatype that appears most frequently is declared as the official datatype for that property.

If the datatype of a property cannot be determined or is unsupported, it is assigned the keyword "UNKNOWN" to signify its undefined datatype. This approach ensures consistency and handles cases where data may be incomplete or ambiguous.

5.3 Formal Concept Analysis

After successfully extracting the input data source for nodes, edges and their corresponding labels and properties, our method builds a formal context as described in Section 4. To perform this task our method integrates the `fcapy` library [Dud24], which provides a robust framework for FCA. Key features of the library utilized in our method include:

- **Data structure for formal contexts:** `fcapy` offers flexible representation for formal contexts, which form the foundation for concept lattice computation.
- **Concept lattice computation:** The library supports multiple algorithms for generating concept lattices. For binary formal contexts, the `Lindig algorithm` [LG00] is employed, which is the method implemented in our method. For multi-valued formal contexts, `fcapy` also includes algorithms such as `CbO` [Kuz93] and approaches based on Random Forest models. However, these algorithms are not relevant for our binary context use case.
- **Visualization capabilities:** `fcapy` provides tools for visualizing the concept lattices, allowing the generated lattices to be saved and analyzed in a graphical format.

With help from `fcapy` a concept lattice will be computed, which is the core structure for our method.

5.4 Validation

After successfully extracting the schema from the input graph, the method performs validation of the graph against the schema. Following the validation approach proposed in [ABD+23], we verify that each node in the graph conforms to at least one type defined in the schema. Once we know the valid nodes, we validate the edges by checking if their endpoint types align with the defined types in the schema.

In Section 2.2 we discussed when a node or edge conforms to a type and when a property graph conforms to a graph type. A node is considered to conform to a given type if it satisfies the requirements of at least one of its defined formal base types. To verify this, we check whether the node contains all mandatory attributes required by the type. For types where labels or properties are defined as **closed**, any additional attributes in the node must match the optional attributes specified by the type. Conversely, if a type is **open** regarding labels or properties, the node remains valid as long as it includes all mandatory attributes, even if it possesses additional attributes not explicitly defined in the type. This behavior can be controlled using the parameters `open_labels` and `open_properties`.

In the case of edges, validation extends beyond the attributes and also involves ensuring that the endpoints of the edge conform to the specified types for the source and target nodes.

For the input graph to conform to the schema, every node and edge must conform to at least one of the defined types.

Requirements for a Valid Schema

In principle, the method generates a schema such that the input graph conforms to the schema. However, this validity is only guaranteed when the outlier threshold parameters are set to zero, specifically:

- `type_outlier_threshold`,
- `label_outlier_threshold`,
- `property_outlier_threshold`,
- `endpoint_outlier_threshold`.

These thresholds are used to exclude outliers during the type discovery process. Outliers might arise due to errors in the data or other anomalies. While these thresholds help refine type definitions by ignoring rare cases, they introduce the possibility that some

nodes or edges in the graph do not fit any type in the schema. Consequently, when validating the graph against the schema, such outliers may be marked as invalid since they lack a corresponding type definition.

Additionally, the parameters `optional_labels` and `optional_properties` must be set to `true`. These parameters allow users to specify whether optional labels and/or optional properties should be permitted in the schema. This capability serves two primary purposes:

1. To focus solely on discovering mandatory attributes of graph elements. This can be useful in scenarios where optional attributes are irrelevant to the analysis.
2. To filter out optional labels or properties as outliers or false data automatically. This is particularly helpful when the user knows that the input graph dataset should not contain any optional labels or properties. In such cases, these elements are excluded without requiring specific threshold values to be defined.

However, the same issues arise as with threshold values. Disallowing optional labels or properties may result in the generation of a rigid schema. Such a schema might not accommodate the input graph fully, leading to potential validation failures.

5.5 Output

The output of the method includes the following documents:

- **PG-Schema**: Our approach primarily delivers a schema for the input property graph, adhering to the PG-Schema definition introduced in [ABD+23]. More specifically, it generates a schema that corresponds to a fragment PG-Schema, which is further described in Section 2.2.
- **Graph-Entity to Schema-Type Mapping**: If a valid schema was produced, our method will output a JSON file containing a mapping of each node and edge to its corresponding type of the schema.
- **Invalid Elements**: If the input graph does not conform to the produced schema (for instance, due to outliers in the data), then the graph elements that do not conform to any type in the schema will be included in a separate output file. This file lists the invalid elements, enabling further analysis or refinement of the schema.
- **Concept Lattice Visualization**: The output of our method also contains a visualization of the corresponding concept lattices produced in the process.
- **Merged Schema**: If an existing schema is provided as input, the method will additionally produce a merged schema that combines the input schema with the schema extracted from the property graph instance. This merged schema serves as an integration of the predefined structure and the newly identified elements.

CHAPTER 6

Evaluation

This section begins by outlining the experimental setup used for our evaluation. Next, we present the datasets employed to assess our method against state-of-the-art approaches in property graph schema discovery, followed by a comparative evaluation. We then evaluate our method further by generating synthetic graphs based on a variety of different schemas. Additionally, we evaluate our schema merging method. Finally, we discuss the results and draw conclusions based on our findings.

6.1 Experiment Setup

In this section, we detail the hardware and software environment used to conduct the experiments. Ensuring reproducibility and consistency across runs, we document the specifications of the hardware, the versions of software and libraries. All experiments were run in a controlled environment with the configurations and software versions specified below. The codebase and datasets have been made available on GitHub [Hit25] for reproducibility.

Hardware Specifications

The experiments were conducted on a machine with the following hardware configuration:

- **Processor:** AMD Ryzen 7 5800X 8-Core Processor (3.80 GHz)
- **Memory:** 32 GB DDR4 RAM at 3200 MHz
- **Storage:** 1 TB SSD for primary data processing
- **Graphics:** Radeon RX 580X
- **Operating System:** Windows 10 Pro 22H2

Software Environment

The method is implemented using Python Version 3.10 and Neo4j Community Edition 5.3.0. In the following table are the most important libraries/packages and their version listed.

Library/Package	Version	Description
caspailleur	0.2.0	Package for mining concepts and implications in binary data with FCA framework.
fcapy	0.1.4.4	A python library to work with FCA.
graphviz	0.20.3	Python interface for Graphviz graph drawing software.
kiwisolver	1.4.7	A fast implementation of the Cassowary constraint solving algorithm.
matplotlib	3.9.2	Comprehensive library for creating static, animated, and interactive visualizations.
neo4j	5.25.0	Official driver for Neo4j graph database.
networkx	3.3	Python library for the creation, manipulation, and analysis of complex networks.
numpy	2.1.2	Fundamental package for numerical computation in Python.
pandas	2.2.3	Powerful data analysis and manipulation library.
pydot	3.0.2	Interface to Graphviz Dot language.
pyparsing	3.1.4	Parser library for defining and executing grammars.
pyroaring	1.0.0	Python bindings for Roaring bitmaps.
scikit-learn	1.5.2	Machine learning library for Python.
scikit-mine	1.0.0	Pattern mining algorithms for scikit-learn.
scipy	1.14.1	Library for scientific computing and technical computing.

Table 6.1: Software Libraries and Packages Used in the Experiments.

6.2 Evaluation Metrics

As discussed in [Lei21], the choice of a suitable schema for a database is dependent on the specific use case, and there is no universally best solution. However, to evaluate our method, it is essential to define metrics and use datasets as well as generated data graphs that include a ground truth schema. This allows us to compare our inferred results against a known solution.

The evaluation metrics we employ are precision, recall, and F1-score. We consider an inferred type as **True Positive (TP)**, if the the inferred type is also part of the ground truth schema and as **False Positive (FP)** if the type is not part of the ground truth schema. We further consider a type present in the ground truth that was not inferred as **False Negative (FN)**.

We consider an inferred type as equivalent to a ground truth type if it correctly captures most of the labels and properties of the original. This criterion, though loosely defined, reflects our focus on practical accuracy, recognizing that minor differences do not significantly affect the inferred types correctness. Since these differences often involve small deviations, we treat such cases as successful type extraction while addressing notable discrepancies in the evaluation of results.

Additionally, we measure the time required for the schema extraction process. This measurement exclusively captures the time taken for extracting the schema from the data graph. It does not include the time for querying the data graph, generating the graph, or validating the graph against the schema. Further discussion about the runtime performance of the method is provided in Section 6.6.

6.3 Evaluation on Datasets

The first part of our evaluation conducts experiments of our schema discovery method on real-world datasets and synthetic datasets generated from real-world schemas. We chose the following datasets since there exists a ground truth of their schemas, that we can compare and evaluate our results against and because they were also used by existing property graph schema discovery methods in [BDM22] and [Lei21]. This section starts with a description of the datasets and is followed by a comparison and evaluation of the results.

6.3.1 Real-world Dataset

There are relatively few property graph datasets publicly available that include an explicit schema that can be used a ground truth. Two notable datasets that have been used in the evaluation of schema discovery methods are **MB6** [TAH+17] and **Fib25** [TXL+15]. Both datasets are part of the NeuPrint project, with **MB6** providing data on mushroom bodies and **Fib25** focusing on the fruit fly brain. The schema of the **MB6** dataset is depicted in Figure 6.1. The schema of the **Fib25** dataset is almost identical and only differs in the name of some labels. Both datasets are characterized by diverse node and edge properties, as well as multi-labeled nodes and consist of 4 node types and five 5 types. Labels and properties can be optional. However, as they are modeled in a Neo4j graph database, edges are restricted to having a single label.

6.3.2 Synthetic Dataset from Real-World Schemas

For further evaluation of our method, we use our graph generator to create property graphs based on existing schemas.

The LDBC Social Network Benchmark (**LDBC SNB**) represents a synthetic property graph modeling a realistic social network. Its schema is illustrated in Figure 6.2. This property graph comprises 8 node types and 15 edge types. It is characterized by single-

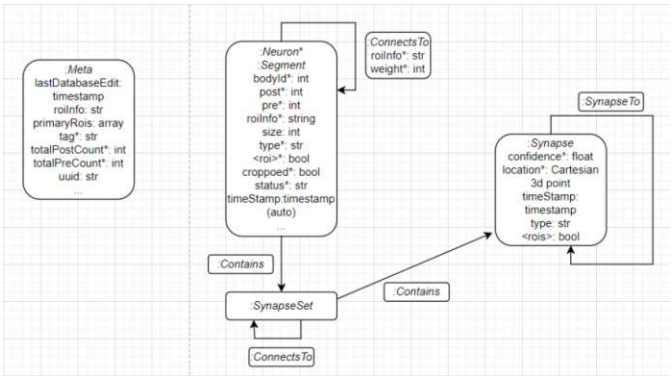

Figure 6.1: Ground-Truth Schema for MB6 dataset, from [Lei21].

labeled nodes with a diverse range of properties, as well as single-labeled edges that do not have any properties.

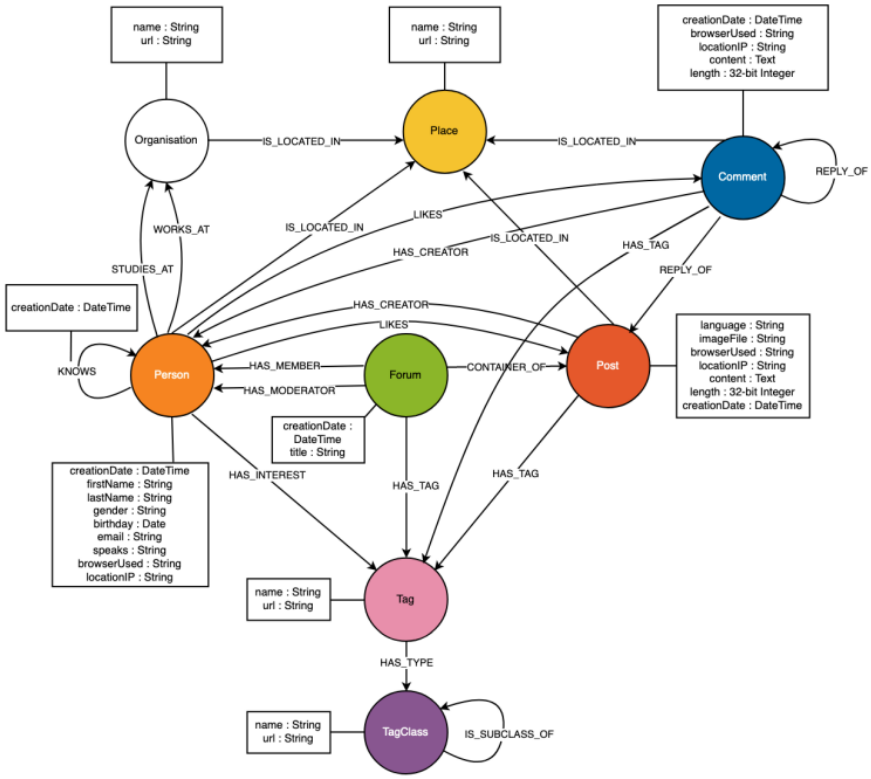

Figure 6.2: Ground-Truth Schema for LDBC dataset, from [BDM22].

The **Northwind** dataset is typically a relational database containing information about products, customers, and their orders. In [Fri16], the original schema is translated into

a property graph schema, which is further refined in [Lei21]. For our evaluation, this dataset is particularly interesting because its nodes do not have any labels. Instead, there are 13 node types distinguished just by their properties, and 10 single-labeled edge types without properties. The schema is depicted in Figure 6.3.

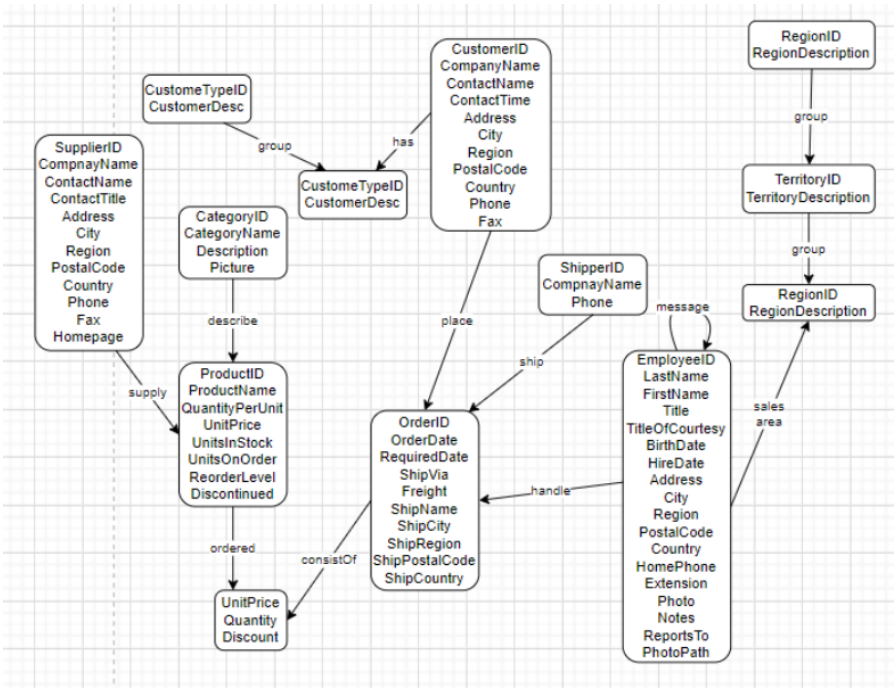

Figure 6.3: Ground-Truth Schema for Northwind dataset, from [Lei21].

6.3.3 Results and Comparison to existing Methods

For each of the four property graphs, we compare the results of our schema extraction method against two existing approaches: the work of [BDM22], referred to as **"DiscoPG"** (as per the name of their tool on GitHub), and the work of [Lei21], abbreviated as **"PGSE"** (derived from the title of their thesis).

Both **DiscoPG** and **PGSE** use the same metrics as we do: precision, recall, and F1 score (with one exception: **PGSE** does not report execution time). Additionally, **PGSE** considers the topology of types when evaluating type equality, whereas **DiscoPG** does not clearly specify the conditions under which types are considered equivalent. As such, while we compare the metrics directly, we acknowledge these differences in their approaches and account for this uncertainty.

The performance metrics achieved by our method are presented in Figure 6.2.

Dataset	Precision	Recall	F1	Time	Node Extraction	Edge Extraction
MB6	1.0	1.0	1.0	20.46s	label-based	label-based
Fib25	1.0	1.0	1.0	36.9s	label-based	label-based
LDBC	1.0	1.0	1.0	52.88s	label-based	label-based
Northwind	1.0	0.85	0.92	151.52s	property-based	label-based

Table 6.2: Performance metrics of the FCA-based method across different datasets.

Results for the MB6 Dataset

The schema extracted by our method captures all types and relationships present in the original property graph schema. But we observed several differences between the schema and the actual dataset. Each label includes an additional identifier indicating the corresponding `neuprint` dataset (e.g., `"mushroombody"` for the **MB6** dataset), which was not described in the original schema. Additionally, some properties defined in the **MB6** schema are missing from the dataset, while others present in the dataset are absent from the schema. After manual inspection in the Neo4j instance, we conclude that the schema description is not fully accurate. Nevertheless, the types are generally consistent, and the resulting schema effectively captures the structure of the original. Thus, we consider the types equivalent.

```
CREATE GRAPH TYPE ResultGraphType STRICT {
(NodeType1: Synapse & mushroombody_Synapse {confidence FLOAT, location POINT, type
  STRING, OPTIONAL alpha3 INTEGER, OPTIONAL alpha1 INTEGER, OPTIONAL alpha2 INTEGER}),
(NodeType2: SynapseSet & mushroombody_SynapseSet),
(NodeType3: Segment & mushroombody_Segment & mushroombody_Neuron? & Neuron? {roiInfo
  STRING, size INTEGER, bodyId INTEGER, post INTEGER, pre INTEGER, OPTIONAL alpha3
  INTEGER, OPTIONAL alpha2 INTEGER, OPTIONAL alpha1 INTEGER, OPTIONAL instance STRING,
  OPTIONAL statusLabel STRING, OPTIONAL status STRING, OPTIONAL cropped INTEGER,
  OPTIONAL type STRING}),
(NodeType5: Meta & medulla7column_Meta {primaryRois LIST, roiInfo STRING, roiHierarchy
  STRING, neuroglancerInfo STRING, totalPreCount INTEGER, lastDatabaseEdit STRING, uuid
  STRING, statusDefinitions STRING, preHPThreshold FLOAT, totalPostCount INTEGER,
  superLevelRois LIST, latestMutationId INTEGER, logo STRING, postHighAccuracyThreshold
  FLOAT, meshHost STRING, dataset STRING, postHPThreshold FLOAT, neuroglancerMeta
  STRING}),
(:NodeType2|NodeType3) - [EdgeType1 : Contains ] → (:NodeType2|NodeType1),
(:NodeType1) - [EdgeType2 : SynapsesTo ] → (:NodeType1),
(:NodeType2|NodeType3) - [EdgeType3 : ConnectsTo {OPTIONAL roiInfo STRING, OPTIONAL
  weight INTEGER, OPTIONAL weightHP INTEGER}] → (:NodeType2|NodeType3)
}
```

Figure 6.4: Resulting schema for MB6 dataset.

PGSE achieves a precision of 0.8, recall of 1.0, and an F1 score of 0.89 for node types and 0.56 precision, recall of 1.0 and an F1 score of 0.81 for edge types. In this schema, the topology of types should not influence the results. We hypothesize that **PGSE** failed to recognize the "Neuron/Segment" type as a single type, instead splitting it into two separate types. Based on this, we conclude that our method outperforms **PGSE** in this case.

DiscoPG reports perfect scores for precision, recall, and F1. However, they claim to have identified 19 node types and 27 edge types. It is unclear whether they further merge

some of these types or consider subtype relationships within these types. **DiscoPG** does, however, outperform our method in terms of execution time, completing the extraction in just 4.53 seconds compared to our 20.46 seconds.

Results for the Fib25 Dataset

In the **Fib25** property graph, we also observed differences between the schema specification and the actual data graph. Similar to the **MB6** dataset, we conducted a manual verification of labels and properties in the Neo4j instance. Therefore we consider our resulting schema to accurately reflect both the structure of the original schema and the actual data in the graph.

```
CREATE GRAPH TYPE ResultGraphType STRICT {
(NodeType1: medulla7column_Synapse & Synapse {confidence FLOAT, location POINT, type
  STRING, OPTIONAL proximal INTEGER, OPTIONAL distal INTEGER}),
(NodeType2: medulla7column_SynapseSet & SynapseSet),
(NodeType3: medulla7column_Segment & Segment & medulla7column_Neuron? & Neuron?
  {roiInfo STRING, bodyId INTEGER, size INTEGER, pre INTEGER, post INTEGER, OPTIONAL
  proximal INTEGER, OPTIONAL distal INTEGER, OPTIONAL cropped INTEGER, OPTIONAL
  statusLabel STRING, OPTIONAL status STRING, OPTIONAL instance STRING, OPTIONAL type
  STRING}),
(NodeType5: Meta & medulla7column_Meta {primaryRois LIST, roiInfo STRING, roiHierarchy
  STRING, neuroglancerInfo STRING, totalPreCount INTEGER, lastDatabaseEdit STRING, uuid
  STRING, statusDefinitions STRING, preHPThreshold FLOAT, totalPostCount INTEGER,
  superLevelRois LIST, latestMutationId INTEGER, logo STRING, postHighAccuracyThreshold
  FLOAT, meshHost STRING, dataset STRING, postHPThreshold FLOAT, neuroglancerMeta
  STRING}),
(:NodeType3|NodeType2) - [EdgeType1 : Contains ] → (:NodeType2|NodeType1),
(:NodeType3|NodeType2) - [EdgeType2 : ConnectsTo {OPTIONAL roiInfo STRING, OPTIONAL
  weight INTEGER, OPTIONAL weightHP INTEGER}] → (:NodeType3|NodeType2),
(:NodeType1) - [EdgeType3 : SynapsesTo ] → (:NodeType1)
}
```

Figure 6.5: Resulting schema for Fib25 dataset.

PGSE does not provide results for this dataset. **DiscoPG**, as with the **MB6** data graph, achieves perfect scores on all metrics and surpasses our method in execution time. However, they report identifying 26 node types and 106 edge types. Similar to the MB6 evaluation the are some issues in the reported of the results. Moreover, it is unclear why **DiscoPG** extracts significantly more types for the **Fib25** dataset than for the **MB6**, given that their original schemas are nearly identical and contain the same number of types.

Results for the LDBC-SNB Dataset

We synthesized the **LDBC-SNB** property graph with 417,169 nodes and 831,818 edges. Our method successfully captures the original schema with complete accuracy. The types are clearly identifiable by their labels resulting in a accurate extraction of the schema.

The **PGSE** method achieves a precision of 1.0, recall of 0.79 and an F1 score of 0.88 for node types and a precision of 0.52, recall of 0.65 and an F1 score of 0.58 for edge types. Our method clearly devlivers better results compared to **PGSE** on this property graph.

```
CREATE GRAPH TYPE ResultGraphType STRICT {
(NodeType1: Place {name STRING, url STRING, type STRING}),
(NodeType2: Organisation {name STRING, url STRING, type STRING}),
(NodeType3: Comment {creationDate DATE, browserUsed STRING, locationIP STRING, content
  STRING, length INTEGER}),
(NodeType4: TagClass {name STRING, url STRING}),
(NodeType5: Post {creationDate DATE, browserUsed STRING, locationIP STRING, content
  STRING, length INTEGER, language STRING, imageFile STRING}),
(NodeType6: Tag {name STRING, url STRING}),
(NodeType7: Forum {title STRING, creationDate DATE}),
(NodeType8: Person {firstName STRING, lastName STRING, gender STRING, birthday DATE,
  creationDate DATE, locationIP STRING, email STRING, speaks STRING, browserUsed
  STRING}),
(:NodeType6) - [EdgeType1 : HasType ] → (:NodeType4),
(:NodeType8) - [EdgeType2 : StudyAt {classYear INTEGER}] → (:NodeType2),
(:NodeType8) - [EdgeType3 : IsLocatedIn ] → (:NodeType1),
(:NodeType3|NodeType5|NodeType7) - [EdgeType4 : HasTag ] → (:NodeType6),
(:NodeType8) - [EdgeType5 : Knows {creationDate DATE}] → (:NodeType8),
(:NodeType4) - [EdgeType6 : IsSubClassOf ] → (:NodeType4),
(:NodeType8) - [EdgeType7 : HasInterest ] → (:NodeType6),
(:NodeType1) - [EdgeType8 : isPartOf {workFrom INTEGER}] → (:NodeType1),
(:NodeType7) - [EdgeType9 : HasModerator ] → (:NodeType5),
(:NodeType7) - [EdgeType10 : HasMember {creationDate DATE}] → (:NodeType8),
(:NodeType3|NodeType5) - [EdgeType11 : HasCreator ] → (:NodeType8),
(:NodeType8) - [EdgeType12 : WorkAt {workFrom INTEGER}] → (:NodeType2),
(:NodeType3) - [EdgeType13 : ReplyOf ] → (:NodeType3|NodeType5),
(:NodeType8) - [EdgeType14 : Likes {creationDate DATE}] → (:NodeType3|NodeType5)
}
```

Figure 6.6: Resulting schema for LDBC dataset.

The **DiscoPG** method achieves perfect metrics but again contradicts itself with the number of types extracted. With an execution time of 47.44 seconds, it slightly outperforms our method. An interesting observation in their graph visualization of their extracted schema is the discovery of two subtypes for the Post type. Although these types are not present in the ground truth schema, they are not entirely incorrect. The UML specification of the **LDBC-SNB** dataset reveals that Post and Comment share a supertype called Message. While their method identifies similarities suggestive of a super/subtype relationship, it incorrectly applies this only to the Post type.

We attempted different configurations of our method to uncover such relationships. The absence of instances for the Message type, along with its lack of a label and high property similarity across schema types, makes detecting these relations impractical while preserving a coherent and useful schema. Using the abstract type lookup we were able to detect the message type as an abstract type of both Post and Comment, but with this setting we also retrieved several other abstract types that do not serve as a useful schema. The high similarity between other types, like Tag and TagClass or Place and Organisation lead to unwanted abstract types in the process.

Results for the Northwind Dataset

We synthesized the **Northwind** property graph with 731818 nodes and 454667 edges. Using the property-based extraction for node types, our method achieves nearly perfect metric score. Since there are two types for customer descriptions and region descriptions that have the exact same attributes, our method cannot distinguish these and only

extracts one type for both. Looking at the UML specification of the dataset, the property graph translation we used regarding these 4 types can be questioned.

```
CREATE GRAPH TYPE ResultGraphType STRICT {
(NodeType2: {CustomerDesc INTEGER, CustomerTypeID INTEGER}),
(NodeType3: {RegionDescription STRING, RegionID INTEGER}),
(NodeType6: {CategoryID INTEGER, CategoryName STRING, Description STRING, Picture
  STRING}),
(NodeType9: {CompanyName STRING, Phone STRING, ShipperID INTEGER}),
(NodeType10: {Discount INTEGER, Quantity INTEGER, UnitPrice INTEGER}),
(NodeType11: {Address STRING, City STRING, CompanyName STRING, ContactName STRING,
  ContactTime DATE, Country STRING, CustomerDesc INTEGER, Fax STRING, Homepage STRING,
  Phone STRING, PostalCode INTEGER, Region STRING, SupplierID INTEGER}),
(NodeType12: {Address STRING, BirthDate DATE, City STRING, Country STRING, EmployeeID
  INTEGER,  Extension STRING, FirstName STRING, HireDate DATE, HomePhone STRING,
  LastName  STRING, Notes  STRING, Photo STRING, PhotoPath STRING, PostalCode INTEGER,
  Region STRING, ReportsTo STRING, Title STRING, TitleOfCourtesy STRING}),
(NodeType13: {Freight INTEGER, OrderDate DATE, OrderID INTEGER, RequiredDate DATE,
  ShipCity STRING, ShipCountry INTEGER, ShipName STRING, ShipPostalCode STRING,
  ShipRegion STRING, ShipVia INTEGER}),
(NodeType14: {Address STRING, City STRING, CompanyName STRING, ContactName STRING,
  ContactTime DATE, Country STRING, CustomerID INTEGER, Fax STRING, Phone STRING,
  PostalCode INTEGER, Region STRING}),
(NodeType15: {TerritoryDescription STRING, TerritoryID INTEGER}),
(NodeType16: {Discontinued INTEGER, ProductID INTEGER, ProductName STRING,
  QuantityPerUnit STRING, ReorderLevel INTEGER, UnitPrice INTEGER, UnitsInStock
  INTEGER, UnitsOnOrder INTEGER}),
(:NodeType12) - [EdgeType1 : Message ] → (:NodeType12),
(:NodeType11) - [EdgeType2 : Supply ] → (:NodeType16),
(:NodeType14) - [EdgeType3 : Place ] → (:NodeType13),
(:NodeType2|NodeType15|NodeType3) - [EdgeType4 : Group ] →
(:NodeType2|NodeType15|NodeType3),
(:NodeType9) - [EdgeType5 : Ship ] → (:NodeType13),
(:NodeType16) - [EdgeType6 : Ordered ] → (:NodeType10),
(:NodeType12) - [EdgeType7 : Handle ] → (:NodeType13),
(:NodeType13) - [EdgeType8 : ConsistsOf ] → (:NodeType10),
(:NodeType6) - [EdgeType9 : Describes ] → (:NodeType16),
(:NodeType14) - [EdgeType10 : Has ] → (:NodeType2),
(:NodeType12) - [EdgeType11 : SalesArea ] → (:NodeType3)
}
```

Figure 6.7: Resulting schema for Northwind dataset.

The **PGSE** method reports only F1 scores for this dataset under various parameter settings, with a maximum of 0.88 for node types and 0.82 for edge types. Based on these results, we conclude that our method outperforms **PGSE**.

DiscoPG did not include this property graph in their evaluation.

6.4 Evaluation on Self Generated Graphs

Since there are not many publicly available datasets with a ground truth schema, we have created several schemas to test our method in detail, which have different features and test different capabilities of the method. We use our graph generator (see 5.1.2) to populate graph instances based on these schemas. In this section we explain what the schemas should test, what features they have and what results our method has achieved.

All experiments are conducted using a synthesized graph containing 10,000 to 100,000 nodes and edges per type. The experiments including the test schemas, parameter settings

and results can be found in the repository [Hit25] in the experiments folder.

The **BigUniversityGraphType** schema is a schema where types are easily identified by its labels but has a lot of different types, complex subtype hierarchies and diverse properties. The goal of this example is to see if the number of types and the size of the schema play any role in the quality of the schema discovery method. The extracted schema accurately represents the original schema. The only difference is that our method does not resolve subtypes in endpoint nodes but instead explicitly enumerates all permissible types. This distinction is purely syntactical and does not affect the schema's correctness or functionality.

The concept lattice in FCA provides a hierarchy of concepts derived from graph data, making it a compelling tool for exploring complex type hierarchies. To evaluate this feature of our method, we tested it on several schemas with complex type hierarchies.

The first type hierarchy graph, referred to as **TypeHierarchyGraph1**, features multiple inheritance and has a hierarchy depth of 3. Most types are identifiable by their label but the `ManagerType` only by a single additional property. Using the label-based extraction method for node types, the original schema could not be reconstructed correctly, as the `ManagerType` was not identified as a distinct type. However, with the property-based extraction method, we achieved a perfect result. This outcome can be attributed to the fact that each type has a unique set of properties that allows for precise identification.

We evaluated our method with several different schemas and observed that as long as types are clearly identifiable by a unique label or a distinct set of properties, our method consistently extracted the original schema without any issues. To push the limits of our approach, we tested it on a modified version of the previous schema. In **TypeHierarchyGraph2**, we removed the labels for `Person` and `Worker`, eliminated the `username` property from the `UserType`, and made the `age` property of the `PersonType` optional. Although this schema is unintuitive and impractical for structuring real data, it serves as a challenging test case. The best result was obtained using the label-property-based extraction method for node types, combined with the max-type merging feature, which limited the output to a maximum of five node types and one edge type. While the overall structure of the schema was partially extracted, it is evident that our method struggled to correctly identify all types in this challenging scenario.

To further evaluate the capabilities of our method in extracting complex type hierarchies, we present the **TypeHierarchyGraph3** schema featuring a 3-1-3 structure with multiple inheritance, extending to a depth of three. Using the property-based node type extraction our method extracts the original schema perfectly. This setup demonstrates the ability of our approach to handle intricate relationships and layered dependencies between types, showcasing its effectiveness in reconstructing sophisticated schemas accurately.

To evaluate the extraction of type hierarchies comprehensively, we also tested whether our method can successfully extract hierarchies involving edge types on the schema **EdgeTypeHierarchyGraph**. The following schema exemplifies a scenario with multiple inheritance among edge types. Using property-based edge type extraction, our

method successfully reconstructs the original schema with complete accuracy. This result highlights our methods capability to handle complex edge type hierarchies effectively.

The **AbstractGraphType** schema illustrates a scenario where an abstract type combining the properties of both `ManagerType` and `EmployeeType` could be beneficial. This abstract type, referred to as `PersonType`, would include common properties such as `name`, `birthdate`, `email`, and `phone_number`. The result demonstrates that when the `abstract_type_lookup` parameter is set to true, our method successfully identifies such structures and derives a useful abstract type. Furthermore, the rest of the schema is extracted with 100 percent accuracy.

The **LibraryGraphType** schema illustrates a complex structure with type hierarchies and optional attributes. During our experiments, we observed that subtype relationships with optional attributes can pose challenges in schema extraction. Specifically, it can be difficult to determine whether a particular value belongs to a subtype or is merely an optional attribute of its parent type. For instance, when examining `BookType` and its subtype `PrintedBookType`, it is possible that `PrintedBookType` might not be identified as a distinct type. Instead, the additional labels and properties specific to the subtype may be incorrectly interpreted as optional attributes of `BookType`. The label-based or property-based approach fails in such cases. However, the label-property-based method, when combined with the maximal number of types feature, successfully reconstructs the schema perfectly.

In addition to the predefined schemas, we evaluated our method on various characteristics of input property graphs. These included unconnected graphs, cyclic graphs, and graphs where either only labels or only properties were present. Such characteristics did not impact the method, and schemas were successfully extracted.

We also tested for outliers, such as single or small sets of nodes or edges where labels or properties were misspelled, or where nodes and edges were entirely different from the rest of the graph, representing erroneous data insertion. By appropriately setting the parameters `type_outlier_threshold`, `label_outlier_threshold`, and `property_outlier_threshold`, the method produced suitable results that excluded redundant types for outliers. However, in such cases, the resulting schema did not conform to the input property graph, as the outliers did not belong to any type.

Additionally, we evaluated property graphs where some properties did not match, i.e. cases where the property key was the same, but the associated values were of different datatypes. Since our method determines the most frequent datatype for a property in advance, it handled such inconsistencies effectively. However, the resulting schema will not conform to the input graph in these cases, as instances with incorrect datatypes for a property would deviate from the schema.

6.5 Schema Merging

In this section, we present a series of test cases for our schema merging approach. Each test involves an original schema provided as input and a new schema, representing scenarios of an evolving graph. The graph generator creates a graph based on the new schema, which is subsequently processed using our FCA-based method to extract its schema. Finally, the extracted schema is merged with the original schema to evaluate the merging process under various settings.The schemas of the experiments and the results can be found in the repository [Hit25] in the experiments folder.

In the **SchemaMergingGraphType1** test case, both the original and new schemas include multiple node and edge types with notable similarities. The original schema defines a type named `PersonType` with two subtypes, while the new schema introduces an additional subtype for `PersonType`. This test case reflects a scenario where new data in the graph leads to the creation of a new subtype. The results demonstrate that the schema was successfully extracted, and the merging process effectively integrated the new subtype into the original schema as a subtype of `PersonType`, ensuring seamless schema consistency.

The original schema of the **SchemaMergingGraphType2** test case again includes a `PersonType` with two subtypes. In contrast, the new schema in this test case does not explicitly include the `PersonType`. Instead, it defines three types that are semantically subtypes of `PersonType`. Two of these types are similar to the subtypes in the original schema, while the third is a completely new type not present in the original schema. Our method extracts the new schema exactly as it was defined for the graph generator, without explicitly detecting the `PersonType`. During the schema merging process, the two identical subtypes are merged and remain subtypes of `PersonType`. The new type is identified as a subtype of `PersonType` and is appropriately added as such, resulting in a satisfactory and coherent merged schema.

SchemaMergingGraphType3 tests whether two schemas, the original and the new, can still be correctly merged when they do not share any similar types. Our method successfully handled this scenario and merged the schemas as expected.

In the **SchemaMergingGraphType4** test case, the original schema contains several node types that are also present in the new schema. However, the new schema extends these types with additional labels and properties not found in the original schema. This tests scenarios where new data or additional information is added to the graph. Similarly, structural differences in the edge types are also tested. Notably, the original schema includes an edge type called `TeachesType`, while the new schema introduces `TutorsType`, which differs in name and labels but shares the same properties. This scenario evaluates situations where newly added data uses a different name for what should be the same kind of data. The result shows that both the additional information is correctly added and merged, and that `TeachesType` and `TutorsType` are identified as the same type and merged accordingly.

The **SchemaMergingGraphType5** test case, tests similarrly to **SchemaMerging-GraphType1** if a type in the new schema not present in the original one, that is semantically a subtype of a type in the original one, will be merged as such, but for edge types. Additionally it tests if similar types in the new schema with additional labels than the corrsponding type in the orignal schema will be merged correctly. Our methods also merges these scenarios correctly for edge types.

The **SchemaMergingGraphType6** test case involves two highly complex schemas. Both the original and the new schema feature multiple inheritance, combined with a hierarchy depth of 4. The type structure is organized as follows: in hierarchy layer one, there is one type; in layer two, two types that are subtypes of layer one; in layer three, one type that inherits from all types in layer two; and in layer four, two types that are subtypes of layer three. The original and new schemas differ in layers two and four, sharing one of the two types in each layer, but each also has one additional type not present in the other. This should result in a 1-3-1-3 hierarchy. While this scenario is not intuitive for database modeling, it serves to test the limitations of our approach. For this test case, our method was unable to perform the merging process as intended. Although the new schema is extracted correctly, and the merging of layers one and two proceeds as intended, the subsequent layers (three and four) fail to identify and merge the types correctly. Instead, many properties are mistakenly identified as optional ones, rather than being correctly assigned to their respective types.

6.6 Runtime Analysis

To evaluate the runtime of our method, we conducted experiments under various settings, scaling different characteristics of the input property graph as well as the schema from which the graph generator populates the graph. The default configuration for all experiments consists of a schema containing 5 node types and 5 edge types, each with one unique label and one unique property. For each type, 10,000 instances are generated. Each setting is evaluated across the three main approaches of our method: label-based, property-based, and label-property-based. The results for each setting compare the overall runtime of the three methods and provide a breakdown of the runtime for each stage: lattice computation, type extraction, which includes outlier removal, type merging, type enrichment, type validity checks and endpoints computation. The experiments including the test schemas can be found in the repository [Hit25] in the experiments folder.

First we scaled the total number of unique labels and properties in the graph, ranging from ten to one hundred. As shown in Figure 6.8, the lattice computation account for the most time-consuming part of the entire process.

Overall, the number of labels and properties has little impact on the runtime. Even in the maximum setting of 100 labels and 100 properties, the longest recorded execution time was 14.01 seconds. Figure 6.9 illustrates that the label-property-based method takes the longest, while the label-based and property-based methods exhibit similar runtimes.

Attributes	10	25	50	75	100
Node Lattice	0.67s	1.27s	2.08s	2.95s	3.80s
NodeType Extraction	0.04s	0.04s	0.06s	0.07s	0.08s
Edge Lattice	0.63s	1.10s	1.99s	2.80s	3.67s
EdgeType Extraction	0.11s	0.12s	0.14s	0.15s	0.15s
Overall	1.54s	2.63s	4.36s	6.06s	7.79s

(a) Label-Based Method

Attributes	10	25	50	75	100
Node Lattice	0.65s	1.20s	1.98s	2.80s	3.72s
NodeType Extraction	0.03s	0.03s	0.04s	0.05s	0.05s
Edge Lattice	0.61s	1.04s	1.92s	2.65s	3.60s
EdgeType Extraction	0.10s	0.09s	0.11s	0.11s	0.12s
Overall	1.47s	2.47s	4.13s	5.69s	7.59s

(b) Property-Based Method

Attributes	10	25	50	75	100
Node Lattice	0.99s	2.03s	3.49s	5.43s	6.96s
NodeType Extraction	0.03s	0.03s	0.03s	0.05s	0.04s
Edge Lattice	0.94s	1.82s	3.46s	5.24s	6.83s
EdgeType Extraction	0.09s	0.09s	0.09s	0.10s	0.10s
Overall	2.16s	4.04s	7.16s	10.90s	14.01s

(c) Label-Property-Based Method

Figure 6.8: Comparison of different Methods on scaling Attribute Size(Labels + Properties).

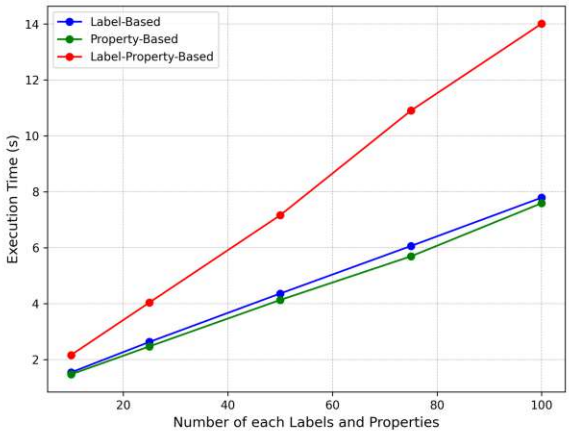

Figure 6.9: Runtime for increasing Number of Attributes.

In the next set of experiments, we aimed to evaluate the impact of the input graph size on runtime. We scaled the total number of instances (nodes and edges) in the graph from 100 thousand to 10 million. As depicted in Figure 6.10c, the lattice computation remains the most time-consuming part of the process.

Overall, the size of the input property graph has a significant effect on runtime (see Figure 6.11). The longest recorded execution time for a graph with 10 million instances was 228.34 seconds. Furthermore, the label-property-based method consistently exhibited the highest runtime among the three approaches.

In the final set of experiments, our objective was to stress-test our method by significantly increasing the complexity of the schema used by the graph generator. We scaled the

PG Size	100k	250k	500k	1M	2.5M	5M	10M
Node Lattice	0.67s	1.42s	2.95s	5.94s	15.23s	31.56s	64.65s
NodeType Extraction	0.04s	0.11s	0.22s	0.45s	1.23s	2.48s	5.24s
Edge Lattice	0.63s	1.44s	2.97s	5.60s	15.23s	31.00s	63.54s
EdgeType Extraction	0.11s	0.32s	0.71s	1.90s	4.01s	8.65s	18.28s
Overall	1.54s	3.52s	7.31s	14.83s	38.05s	78.57s	161.68s

(a) Label-Based Method

PG Size	100k	250k	500k	1M	2.5M	5M	10M
Node Lattice	0.65s	1.39s	3.10s	5.65s	14.58s	29.59s	64.15s
NodeType Extraction	0.03s	0.08s	0.18s	0.39s	1.09s	2.35s	4.99s
Edge Lattice	0.61s	1.40s	2.73s	5.25s	14.27s	29.49s	62.07s
EdgeType Extraction	0.10s	0.30s	0.59s	1.77s	3.69s	8.03s	16.50s
Overall	1.47s	3.40s	7.05s	14.00s	35.99s	74.19s	157.19s

(b) Property-Based Method

PG Size	100k	250k	500k	1M	2.5M	5M	10M
Node Lattice	0.99s	2.13s	4.67s	8.94s	23.43s	51.54s	100.76s
NodeType Extraction	0.03s	0.07s	0.13s	0.26s	0.66s	1.41s	2.82s
Edge Lattice	0.94s	2.21s	4.45s	8.65s	23.24s	49.77s	99.70s
EdgeType Extraction	0.09s	0.30s	0.53s	1.55s	3.20s	7.60s	15.37s
Overall	2.16s	4.93s	10.23s	20.32s	52.86s	115.12s	228.34s

(c) Label-Property-Based Method

Figure 6.10: Comparison of different Methods on scaling Property Graph Size (Nodes + Edges).

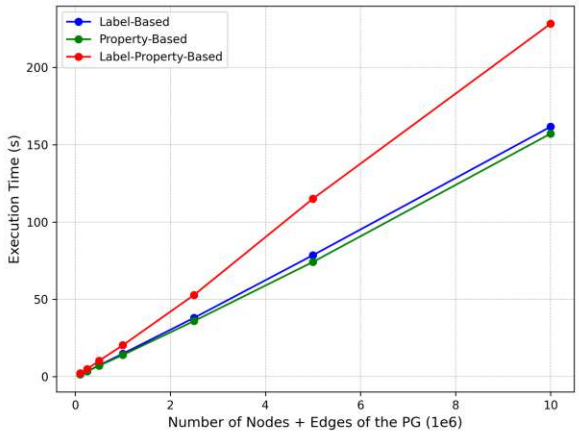

Figure 6.11: Runtime for increasing Property Graph Size.

number of types in the schema from 10 to 100, with each type having 3 labels and 3 properties. However, in this setting, attributes were not unique, instead, types shared overlapping attributes. The intention behind this setup was to create concept lattices with a high number of distinct concepts as well as numerous sub- and super-concept relationships. While these schemas do not necessarily reflect realistic data models, they were designed to generate large and intricate concept lattices. Additionally, we restricted the maximum number of node and edge types in the extracted schema to 10, further testing the efficiency of the type merging process during extraction. As shown in Figure 6.12c, even with extensive type merging, lattice computation remains the most time-consuming step.

Furthermore, with increasingly complex schemas and, consequently, highly intricate concept lattices, the overall runtime rises significantly, as can be seen in Figure 6.13.

Types	10	25	50	75	100
Node Lattice	0.66s	10.47s	35.77s	80.51s	139.93s
NodeType Extraction	0.04s	0.21s	0.51s	0.90s	1.25s
Edge Lattice	0.61s	8.83s	37.83s	77.87s	127.97s
EdgeType Extraction	0.10s	0.38s	1.04s	1.93s	2.23s
Overall	1.49s	20.11s	75.63s	161.91s	272.32s

(a) Label-Based Method

Types	10	25	50	75	100
Node Lattice	0.65s	10.29s	36.28s	82.60s	139.26s
NodeType Extraction	0.03s	0.18s	0.45s	0.83s	1.16s
Edge Lattice	0.61s	7.70s	45.86s	121.09s	270.89s
EdgeType Extraction	0.10s	0.37s	0.98s	1.75s	2.14s
Overall	1.47s	18.77s	84.05s	206.96s	414.40s

(b) Property-Based Method

Types	10	25	50	75	100
Node Lattice	0.99s	20.89s	71.08s	169.26s	277.02s
NodeType Extraction	0.03s	0.44s	1.46s	3.45s	5.67s
Edge Lattice	0.94s	29.75s	182.80s	506.79s	1119.92s
EdgeType Extraction	0.09s	1.02s	4.10s	9.08s	15.39s
Overall	2.16s	52.33s	259.92s	689.28s	1418.93s

(c) Label-Property-Based Method

Figure 6.12: Comparison of different Methods on Number of Types.

Similar to previous experiments, the label-property-based method exhibited the highest runtime. The significant difference in runtime between the node lattice computation and the edge lattice computation can be attributed to the differing patterns of attribute sharing among edges. The attribute relationships between edges appear to be more complex, leading to an increased computational overhead for constructing the corresponding concept lattice.

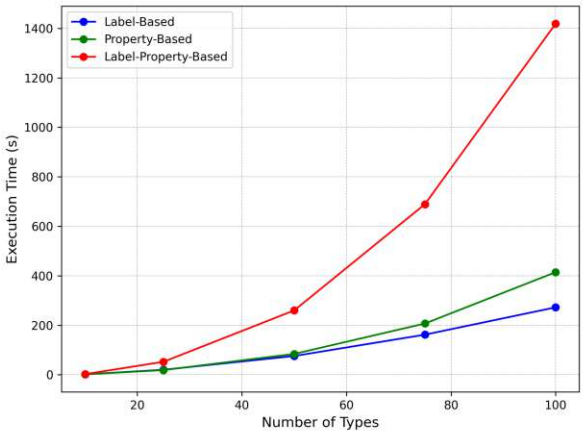

Figure 6.13: Runtime for increasing Number of Types.

As anticipated, the label-property-based method exhibited the highest runtimes overall, as this approach must account for both labels and properties during the computation of the concept lattice. Another consistent trend observed throughout the experiments is that the concept lattice computation constitutes the most time-consuming part of the entire process. Notably, variations in the experimental settings had minimal impact on the type extraction phase, even when extensively testing the merging procedures.

Overall, the runtime of our method can be considered acceptable. Even for the most complex schema tested, designed to be significantly more intricate than any real-world schema, the method completed execution in under seven minutes. Given that schema discovery is not an operation performed frequently but rather a one-time or occasional process for a property graph, and that time constraints are typically not a critical factor in such scenarios, we consider these results to be satisfactory.

6.7 Discussion

We draw the following conclusions from our evaluations. Based on the results from both real-world datasets and experiments on synthesized property graphs, our method demonstrates its ability to discover accurate and useful schemas. In comparison with state-of-the-art approaches, we have shown that our method competes effectively. **PGSE**, the method from the masters thesis of Lei [Lei21], was clearly outperformed across all three datasets we compared it to. The approach from Bonifati et al. [BDM22] yielded results similar to ours, although it was slightly faster in the schema extraction process. However, the way they present their results is unclear to us, and further investigation or clarification is necessary to ensure the correctness of their reported outcomes.

Given that the datasets used for comparison with state-of-the-art methods are relatively simple schemas, we also evaluated our method on more complex schemas and settings to assess its capabilities and limitations. A key reason for considering FCA as a promising method for schema extraction was its ability to use the concept lattice to detect type hierarchies. Our results confirm that such hierarchies can be successfully identified by our method, but only to a certain extent. Our method is able to deal with multiple inheritance and deep hierarchies (inheritance spanning more than one layer). However, when inheritance is combined with optional attributes, our method occasionally struggles to determine whether attributes should be optional or belong to their own type.

Moreover, our method can identify abstract types for node types, but this capability heavily depends on the values of the parameters `abstract_type_threshold` and `merge_threshold`. Since the similarity computation is shared between both the merging process and abstract type detection, if the `merge_threshold` is lower than the `abstract_type_threshold`, types will be merged before they are considered to have an abstract type from which they inherit. This can limit the ability to identify abstract types accurately.

An interesting observation is that when the types of a schema are easily identifiable by either their labels or properties, our method successfully extracts the correct schema. In practice, when designing schemas for databases or modeling data for graphs, it is intuitive to use either labels to define type identities or properties to specify the type. Our method takes advantage of both labels and properties independently as defining features for types, yielding good results in cases like the **LDBC** database (for labels) and the **Northwind** database (for properties). Even when the data is not optimally modeled,

we provide the label-property-based method that considers both attribute categories for type definition.

Another significant aspect is our methods capability to handle optional attributes. For instance, in the **MB6** and **Fib25** databases, the segment type feature the `Segment` label as mandatory but also allow the `Neuron` label. In this case, the concept lattice identifies them as distinct concepts, but our similarity computation and type merging process successfully detects them as the same type, marking `Neuron` as an optional label. We believe that **PGSE** is less effective in detecting these optional attributes, as it appears to extract one extra type, likely due to the `Neuron` label.

Regarding our schema merging process, we found that although it is relatively simple, the method can successfully merge schemas and detect hierarchies between the types of both schemas. This capability is valuable for scenarios where we have multiple graphs or dynamic datasets that evolve over time.

Our method offers a range of parameters that must be carefully tuned to achieve optimal results. On the one hand, this flexibility makes our tool highly adaptable to various property graph structures, but on the other hand, it requires a clear understanding of how to adjust the parameters correctly, which may necessitate several attempts before extracting a useful schema. This should be considered when interpreting the evaluation results, as we only tested our method against cases where a ground-truth schema was available, making the parameter adjustment process easier.

Validity and Conformance: We validated all our results against the PG-Schema parser [ABD+23]. Our schemas are valid according to the schema language definition of PG-Schema, with only minor inconsistencies observed (e.g., the use of "type" as a property key, which is not allowed according to the PG-Schema parser). We also checked the conformity of the input graphs with the extracted schemas. As previously mentioned, this was always the case, except for three situations: (i) when outliers were filtered out, (ii) when optional attributes were not allowed in the parameters, and (iii) when properties had the same key but different data types.

Finally, we categorize our method according to the features discussed in Section 2.3:

- **Scalability**: Our methods primary focus is on providing useful schemas rather than achieving the fastest processing times. The runtime analysis shows that for complex property graphs, our method can take several minutes. Considering that very large graph databases can contain more than a billion nodes and edges [SMS+20], the runtime of our method may become impractically long. Based on the trend observed in our experiments regarding the impact of graph size on runtime, processing a billion instances would likely exceed five hours. Furthermore, since our experiments were conducted on synthesized data and did not account for the time required to query an actual graph database, the overall runtime in a real-world setting could be even longer. Therefore, we conclude that our method cannot be considered scalable for extremely large graphs.

- **Incrementality**: By providing the option to merge the extracted schema with a user-provided schema, our method fulfills the incrementality requirement.
- **Stability**: As outlined in Section 4, our method always produces the same schema for a given input property graph.
- **Online Capability**: Our method is not able to process remote data sources. Although it could easily be extend to have this feature.
- **Hybrid**: Our method is not hybrid, as it only considers the structure of the input graph for schema discovery. However, this feature could be explored in future work. For instance, in the **LDBC** dataset, we were unable to detect an abstract type that could have been beneficial. Considering additional schema-related information could potentially improve the detection of such types.

CHAPTER 7

Conclusion

7.1 Summary

In this thesis, we explored the use of Formal Concept Analysis for schema extraction in the property graph data model. We proposed a fully automatic schema discovery method based on the FCA framework. Our approach includes multiple strategies for constructing a formal context and deriving types from the resulting concept lattice.

Throughout the course of this work, we not only investigated the applicability of FCA but also implemented the entire methodology, including additional processes for evaluating and validating the results. This ensures that our method is ready for practical use by other researchers[1].

Guided by the following research questions, we developed, implemented, and evaluated our approach:

RQ 1: *How can FCA be effectively applied to discover schemas, including types and type hierarchies, in property graphs?*

RQ 2: *How does the performance of the FCA-based schema extraction framework compare to existing schema discovery methods in terms of accuracy and scalability?*

RQ 3: *What are the limitations of using FCA for schema discovery in property graphs?*

Section 4 and Section 5 address ***RQ 1***, where we investigated and described in detail how data from a property graph can be utilized to construct a formal context and, subsequently, how the concept lattice can be leveraged to identify types for a meaningful schema. Our analysis revealed multiple viable approaches, which were therefore incorporated into our methodology. Labels and properties serve as key attributes for characterizing types in property graphs and were accordingly used to construct various formal context variations.

[1] https://github.com/JakobRH/FCA_Schema_Discovery

The set of concepts and the corresponding concept lattice generated by FCA proved to be a valuable foundation for identifying types and supertype relations. However, since the concept lattice alone cannot be directly mapped to a schema to produce a practically useful structure, we demonstrated that several additional steps, such as type merging, enrichment, outlier removal, and abstract type identification, are essential to achieving satisfactory results.

To address ***RQ 2***, we compared our method against two state-of-the-art schema discovery approaches for property graphs, using the datasets employed in the respective studies. The results indicate that our approach not only competes with these methods in terms of schema quality and scalability but also, in some cases, outperforms them.

Although the FCA framework proved to be highly effective for schema extraction, it also has certain drawbacks and limitations, which we discuss in relation to ***RQ 3***. First, the concept lattice does not provide information about all aspects of a schema. For example, types and labels combined with the |-operator of the PG-Schema cannot be directly extracted. As discussed earlier, the concept lattice groups nodes and edges based on shared attributes. However, if a type were to theoretically inherit from two distinct types combined using the |-operator, the FCA-framework would generate two separate concepts without indicating that they could represent a single type inheriting from both. When considering the combination of labels using the |-operator, the detectability of such types depends on how the concept lattice is constructed. If the lattice is based on shared label attributes, these types would appear as separate entities. On the other hand, if the lattice is built solely on shared properties, one would need to check, across all instances in the graph, whether the two labels (or label combinations) occur in a mutually exclusive manner. Even in cases where this holds true, the original schema might have defined them as optional labels. As a result, inferring schemas that incorporate the |-operator is only partially feasible and inherently ambiguous.

Furthermore, certain schema constraints, such as range constraints for property values or cardinality constraints, are desirable for a comprehensive schema but cannot be derived directly from FCA and must be extracted separately using additional techniques.

Additionally, our runtime analysis revealed that lattice computation is the most time-consuming step in the schema discovery process. As a result, the scalability of our approach is limited for very large graphs, presenting a challenge for practical applications in large-scale datasets. An alternative approach could involve partitioning large graphs into smaller subgraphs and processing them iteratively using our schema merging method. We leave the exploration of this idea for future work.

7.2 Future Work

The method presented in this thesis is already practically applicable for schema discovery. However, there are several areas for future work that could enhance its usability and extend its capabilities in schema extraction.

Visualization: Providing users with a visual representation of the extracted schema, as well as relevant elements of the input graph, could greatly improve interpretability. A visualization approach similar to that of Bonifati et al. [BDM22] may be beneficial.

Similarity Measures: Our method currently employs the Jaccard similarity metric to compare types, based on sets of labels and properties. Future work could explore alternative similarity metrics or incorporate additional characteristics of types, such as incident edges/nodes or, for edge types, the types of their endpoints.

Type-Defining Elements: In this thesis, we defined types based on labels and properties, constructing the formal context accordingly. However, a more intricate approach could be considered, such as incorporating datatype values and partitioning them into ranges or leveraging graph topology, as proposed by Lei [Lei21].

Schema Constraints: Enhancing our method, independent of the FCA framework, with techniques to detect and infer constraints over the extracted schema would be valuable. This would require novel techniques for analyzing the input graph and extracting constraint-related information.

Naming of Types: Currently, our method assigns type names based on the concept ID from the concept lattice, ensuring uniqueness but lacking semantic meaning. A simple approach could involve using the first label or property as a placeholder for the name, but this is highly dependent on the input graph and may result in misleading or duplicate names. An interesting direction for future work is leveraging Large Language Models to generate meaningful type names, considering labels, properties, and type hierarchies.

Overview of Generative AI Tools Used

ChatGPT (Version 4o)[2] was utilized to enhance the grammar and language quality of this thesis. However, it is explicitly stated that it did not contribute to the content in any way beyond language improvement. All suggestions provided by the tool were carefully reviewed for consistency and errors.

[2]https://chatgpt.com/

List of Figures

List of Tables

List of Algorithms

Bibliography

[AAB+17] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)*, 50(5):1–40, 2017.

[ABD+23] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data*, 1(2):1–25, 2023.

[AFB24] Renzo Angles, Sebastián Ferrada, and Ignacio Burgos. The Property Graph Data Format (PGDF). *IEEE Access*, 12:159267–159279, 2024.

[Ang18] Renzo Angles. The property graph database model. In *AMW*, 2018.

[BDM22] Angela Bonifati, Stefania Dumbrava, and Nicolas Mir. Hierarchical clustering for property graph schema discovery. In *EDBT 2022: 25th International Conference on Extending Database Technology*, pages 449–453. OpenProceedings. org, 2022.

[BKL20] Redouane Bouhamoum, Zoubida Kedad, and Stéphane Lopes. Scalable schema discovery for RDF data. *Transactions on Large-Scale Data-and Knowledge-Centered Systems XLVI*, pages 91–120, 2020.

[BKMLK18] Redouane Bouhamoum, Kenza Kellou-Menouer, Stephane Lopes, and Zoubida Kedad. Scaling up schema discovery for RDF datasets. In *2018 IEEE 34th International Conference on Data Engineering Workshops (ICDEW)*, pages 84–89. IEEE, 2018.

[BKR23] Samira Babalou and Birgitta König-Ries. Towards building knowledge by merging multiple ontologies with comerger: A partitioning-based approach. *Applied Ontology*, 18(4):307–341, 2023.

[CHS05] Philipp Cimiano, Andreas Hotho, and Steffen Staab. Learning concept hierarchies from text corpora using formal concept analysis. *Journal of artificial intelligence research*, 24:305–339, 2005.

[CPF13] Klitos Christodoulou, Norman W Paton, and Alvaro AA Fernandes. Structure inference for linked data sources using clustering. In *Proceedings of the Joint EDBT/ICDT 2013 Workshops*, pages 60–67, 2013.

[CR14] Jesse Xi Chen and Marek Z Reformat. Learning categories from linked open data. In *Information Processing and Management of Uncertainty in Knowledge-Based Systems: 15th International Conference, IPMU 2014, Proceedings, Part III 15*, pages 396–405. Springer, 2014.

[CSS20] Diana Cristea, Christian Săcărea, and Diana Sotropa. Knowledge discovery and visualization in healthcare datasets using formal concept analysis and graph databases. In *The 5th International Workshop on Knowledge Discovery in Healthcare Data (KDH)*, 2020.

[Deh14] Razieh Mehri Dehnavi. Inferring Missing Schema from Linked Data using Formal Concept Analysis, 2014.

[DR02] Hong-Hai Do and Erhard Rahm. Coma—a system for flexible combination of schema matching approaches. In *VLDB'02: Proceedings of the 28th International Conference on Very Large Databases*, pages 610–621. Elsevier, 2002.

[Dud24] Egor Dudyrev. FCApy. `https://github.com/EgorDudyrev/FCApy`, 2024. Commit: 464467d, Accessed: 2025-11-01.

[FHK+20] Sébastien Ferré, Marianne Huchard, Mehdi Kaytoue, Sergei O Kuznetsov, and Amedeo Napoli. Formal concept analysis: from knowledge discovery to knowledge processing. *A Guided Tour of Artificial Intelligence Research: Volume II: AI Algorithms*, pages 411–445, 2020.

[Fri16] Thomas Frisendal. Graph data modeling for nosql and sql: Visualize structure and meaning. *Technics Publications*, 2016.

[Fu16] Gaihua Fu. FCA based ontology development for data integration. *Information processing & management*, 52(5):765–782, 2016.

[GEMC18] Silvio Normey Gómez, Lorena Etcheverry, Adriana Marotta, and Mariano P Consens. Findings from Two Decades of Research on Schema Discovery using a Systematic Literature Review. *AMW*, 2018.

[GSW03] Bernhard Ganter, Gerd Stumme, and R Wille. Formal concept analysis: Methods, and applications in computer science. *TU: Dresden, Germany*, 2003.

[GW12] Bernhard Ganter and Rudolf Wille. *Formal concept analysis: mathematical foundations*. Springer Science & Business Media, 2012.

[Haa04] Hele-Mai Haav. A semi-automatic method to ontology design by using fca. In *Proceedings of CLA*, pages 13–15, 2004.

[Hit25] Jakob Hitzelhammer. FCA Schema Discovery. `https://github.com/JakobRH/FCA_Schema_Discovery/`, 2025. Commit: 03233d2, Accessed: 2025-20-03.

[KLT+12] Markus Kirchberg, Erwin Leonardi, Yu Shyang Tan, Sebastian Link, Ryan KL Ko, and Bu Sung Lee. Formal concept discovery in semantic web data. In *Formal Concept Analysis: 10th International Conference, ICFCA 2012. Proceedings 10*, pages 164–179. Springer, 2012.

[KMK15] Kenza Kellou-Menouer and Zoubida Kedad. Schema discovery in rdf data sources. In *Conceptual Modeling: 34th International Conference 2015, Proceedings 34*, pages 481–495. Springer, 2015.

[KMK16] Kenza Kellou-Menouer and Zoubida Kedad. A self-adaptive and incremental approach for data profiling in the semantic web. *Transactions on Large-Scale Data-and Knowledge-Centered Systems XXIX*, pages 108–133, 2016.

[KMKT+22] Kenza Kellou-Menouer, Nikolaos Kardoulakis, Georgia Troullinou, Zoubida Kedad, Dimitris Plexousakis, and Haridimos Kondylakis. A survey on semantic schema discovery. *The VLDB Journal*, 31(4):675–710, 2022.

[Kri24] Francesco Kriegel. Efficient Axiomatization of OWL 2 EL Ontologies from Data by Means of Formal Concept Analysis. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 38, pages 10597–10606, 2024.

[KSI+21] Christos Koutras, George Siachamis, Andra Ionescu, Kyriakos Psarakis, Jerry Brons, Marios Fragkoulis, Christoph Lofi, Angela Bonifati, and Asterios Katsifodimos. Valentine: Evaluating matching techniques for dataset discovery. In *2021 IEEE 37th International Conference on Data Engineering (ICDE)*, pages 468–479. IEEE, 2021.

[Kuz93] Sergei O Kuznetsov. A fast algorithm for computing all intersections of objects from an arbitrary semilattice. *Automatic Documentation and Mathematical Linguistics*, (1):17–20, 1993.

[LBH21] Hanâ Lbath, Angela Bonifati, and Russ Harmer. Schema inference for property graphs. In *EDBT 2021-24th International Conference on Extending Database Technology*, pages 499–504, 2021.

[Lei21] Xue Lei. Property graph schema extraction, 2021.

[LG00] Christian Lindig and Gartner Gbr. Fast concept analysis. *Working with Conceptual Structures - Contributions to ICCS*, 2000.

[LPWJ20] Wen Lou, Ruofan Pi, Hui Wang, and Yuan Ju. Low-cost similarity calculation on ontology fusion in knowledge bases. *Journal of information science*, 46(6):823–836, 2020.

[LRKCM18] Artem Lutov, Soheil Roshankish, Mourad Khayati, and Philippe Cudré-Mauroux. Statix—statistical type inference on linked data. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 2253–2262. IEEE, 2018.

[MBR01] Jayant Madhavan, Philip A Bernstein, and Erhard Rahm. Generic schema matching with cupid. In *vldb*, volume 1, pages 49–58, 2001.

[MMM14] Frank Manola, Eric Miller, and Brian McBride. RDF 1.1 Primer. `https://www.w3.org/TR/rdf11-primer/`, 2014. Accessed: 2025-01-08.

[MS01] Alexander Maedche and Steffen Staab. Ontology learning for the semantic web. *IEEE Intelligent systems*, 16(2):72–79, 2001.

[neo] Neo4j. `https://neo4j.com/`. Accessed: 2024-11-27.

[NM03] Natalya F Noy and Mark A Musen. The prompt suite: interactive tools for ontology merging and mapping. *International journal of human-computer studies*, 59(6):983–1024, 2003.

[OYD21] Inès Osman, Sadok Ben Yahia, and Gayo Diallo. Ontology integration: approaches and challenging issues. *Information Fusion*, 71:38–63, 2021.

[PHP24] Jan Portisch, Michael Hladik, and Heiko Paulheim. Background knowledge in ontology matching: A survey. *Semantic Web*, 15(6):2639–2693, 2024.

[Pri06] Uta Priss. Formal concept analysis in information science. *Annu. Rev. Inf. Sci. Technol.*, 40(1):521–543, 2006.

[RAAI24] Farhana Zaman Rozony, MNA Aktar, M Ashrafuzzaman, and A Islam. A systematic review of big data integration challenges and solutions for heterogeneous data sources. *Academic Journal on Business Administration, Innovation & Sustainability*, 4(04):1–18, 2024.

[SBV+21] Sherif Sakr, Angela Bonifati, Hannes Voigt, Alexandru Iosup, Khaled Ammar, Renzo Angles, Walid Aref, Marcelo Arenas, Maciej Besta, Peter A. Boncz, Khuzaima Daudjee, Emanuele Della Valle, Stefania Dumbrava, Olaf Hartig, Bernhard Haslhofer, Tim Hegeman, Jan Hidders, Katja Hose, Adriana Iamnitchi, Vasiliki Kalavri, Hugo Kapp, Wim Martens, M. Tamer Özsu, Eric Peukert, Stefan Plantikow, Mohamed Ragab, Matei R. Ripeanu, Semih Salihoglu, Christian Schulz, Petra Selmer, Juan F. Sequeda, Joshua Shinavier, Gábor Szárnyas, Riccardo Tommasini, Antonino Tumeo, Alexandru Uta, Ana Lucia Varbanescu, Hsiang-Yun Wu, Nikolay Yakovets, Da Yan,

and Eiko Yoneki. The future is big graphs: a community view on graph processing systems. *Commun. ACM*, 64(9):62–71, 2021.

[SGSK18] Giorgos Stoilos, David Geleta, Jetendr Shamdasani, and Mohammad Khodadadi. A novel approach and practical algorithms for ontology integration. In *The Semantic Web–ISWC 2018: 17th International Semantic Web Conference, 2018, Proceedings, Part I 17*, pages 458–476. Springer, 2018.

[SM01] Gerd Stumme and Alexander Maedche. FCA-Merge: Bottom-up merging of ontologies. In *IJCAI*, volume 1, pages 225–230, 2001.

[SMS+20] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB journal*, 29:595–618, 2020.

[TAH+17] Shin-ya Takemura, Yoshinori Aso, Toshihide Hige, Allan Wong, Zhiyuan Lu, C Shan Xu, Patricia K Rivlin, Harald Hess, Ting Zhao, Toufiq Parag, et al. A connectome of a learning and memory center in the adult drosophila brain. *Elife*, 6:e26975, 2017.

[TCH06] Julien Tane, Philipp Cimiano, and Pascal Hitzler. Query-based multicontexts for knowledge base browsing: An evaluation. In *Conceptual Structures: Inspiration and Application: 14th International Conference on Conceptual Structures, ICCS 2006. Proceedings 14*, pages 413–426. Springer, 2006.

[TMA13] Amel Grissa Touzi, Hela Ben Massoud, and Alaya Ayadi. Automatic ontology generation for data mining using fca and clustering. *arXiv preprint:1311.1764*, 2013.

[TS19] Yuroti Tsuboi and Nobutaka Suzuki. An algorithm for extracting shape expression schemas from graphs. In *Proceedings of the ACM Symposium on Document Engineering 2019*, pages 1–4, 2019.

[TXL+15] Shin-ya Takemura, C Shan Xu, Zhiyuan Lu, Patricia K Rivlin, Toufiq Parag, Donald J Olbris, Stephen Plaza, Ting Zhao, William T Katz, Lowell Umayam, et al. Synaptic circuits and their variations within different columns in the visual system of drosophila. *Proceedings of the National Academy of Sciences*, 112(44):13711–13716, 2015.

[VR08] Johanna Völker and Sebastian Rudolph. Lexico-logical acquisition of OWL DL axioms: an integrated approach to ontology refinement. In *International Conference on Formal Concept Analysis*, pages 62–77. Springer, 2008.