Automated Software Engineering (2022) 29:37
https://doi.org/10.1007/s10515-022-00336-y

# FLASc: a formal algebra for labeled property graph schema

Chandan Sharma[1,2] · Roopak Sinha[2]

Received: 31 March 2021 / Accepted: 3 March 2022 / Published online: 2 April 2022

**Abstract**
Contemporary labeled property graph databases are either schema-less or schema-optional to support frequent changes in the structure of data found in domains requiring high flexibility. However, the lack of structure impacts data transformation and loading operations from heterogeneous sources into graph databases. We present a formal algebra FLASc for specifying and generating graph schema for labeled property graph databases. We formally define FLASc and demonstrate the use of FLASc generated graph schemas to systematically transform and load data-sets related to domains of cyber-physical systems, big data analytics and tourism. Findings from three disparate case studies show that FLASc-generated schemas assist in enforcing integrity constraints that reduce the chance of data corruption, hence assuring data consistency and integrity.

**Keywords** Graph schema · Labeled property graph databases · ETL · Data transformation and loading · Neo4j · Cypher

## 1 Introduction

Labeled property graph database henceforth graph database are storage systems that allow modeling of real-world entities as nodes and relationships between entities as edges Angles et al. (2018). Nodes and edges in a graph database have associated labels. Data is stored inside nodes and edges as properties that exist in the form of key-value pairs Angles et al. (2017); Angles and Gutierrez (2008). Graph databases

✉ Chandan Sharma
chandan.sharma@ddn.upes.ac.in; chandan202.alive@gmail.com

Roopak Sinha
roopak.sinha@aut.ac.nz

[1] Cybernetics Cluster, School of Computer Science, University of Petroleum and Energy Studies, Dehra Dun, Uttrakhand, India

[2] Department of Computer Science and Software Engineering, Auckland University of Technology, Auckland, New Zealand

are efficient in storing and managing highly interconnected data-sets related to domains such as transportation networks, social media, bioinformatics, chemistry and astronomy (Angles and Gutierrez 2008; Angles 2012; Angles et al. 2017; Bell et al. 2009; Tetko et al. 2016). Graph databases suit big data applications as they provide a better alternative for modeling and handling complex information (Rodriguez and Neubauer 2010, 2012). Graph databases are more efficient than relational databases for extracting information from highly interconnected data-sets (Sharma et al. 2019; Sharma and Sinha 2019; Sharma 2020; Sharma et al. 2021).

The interconnections between data represent the underlying meaning of a graph data-set. Therefore, maintaining data consistency and integrity is vital in graph databases (Angles and Gutierrez 2008; Kunii 1987). Obtaining a database that is sound and consistent requires embracing good database modeling principles (Badia and Lemire 2011). In contrast to relational databases, modeling principles for graph databases are ad-hoc and not well-grounded (Park et al. 2014). Contemporary graph databases lack mechanisms to ensure data consistency and integrity, especially when the data being stored comes from multiple heterogeneous sources (Reina et al. 2020). A primary reason is that graph databases are either schema-less or schema-optional (Reina et al. 2020). A schema represents the overall structure of a data-set and assists in understanding data semantics (Pokornỳ 2016). Furthermore, schemas aid in defining integrity constraints that are sets of rules for ensuring consistency and integrity in the database that conforms to the schema (Codd 2002; Ghrab et al. 2014). The lack of schema and integrity constraints poses significant challenges in ensuring data consistency and integrity (Khan et al. 2012), in performing advanced analytics (Sharma 2021) and achieving data interoperability (Sciore et al. 1994), and for data integration, query optimization and processing (Frozza et al. 2020).

Traditional database modeling consists of three stages *conceptual, logical* and *physical modeling* (Badia and Lemire 2011). In graph databases, the conceptual modeling stage represents gathering requirements of a given problem domain that are then used for defining entities and relationships between them. The logical modeling stage represents the enforcement of integrity constraints, including mandatory, optional and unique properties associated with entities and relationships defined in the conceptual modeling stage. The physical modeling stage represents the realization of graph schema formulated at the conceptual and logical modeling stage into database creation scripts.

An open problem in graph database design is that practitioners do not have proper guidelines for designing conceptual models (Pokornỳ 2016; Badia and Lemire 2011) that can facilitate systematic transformation and loading of data from heterogeneous sources into graph databases. Conceptual modeling stage is not used in the majority of graph database solutions (Fitzgerald et al. 1999; Brodie and Liu 2010). Graph databases lack abstraction tools Angles and Gutierrez (2008) and most current research is primarily focused on logical and physical modeling (Reina et al. 2020; Pokornỳ et al. 2017; Pokorny 2017). These observations lead us to the following research questions:

RQ1 What are the key strengths and limitations of existing approaches used for modeling graph databases?

RQ2 What mechanisms can be designed to formulate conceptual and logical graph schemas for labeled property graph databases?

RQ3 In order to ensure data consistency, how can the graph schema generated by RQ2 be used to systematically import data from heterogeneous sources into a labeled property graph database?

RQ3.1 How can the Extract-Transfrom-Load design pattern be extended in order to support loading data-sets for heterogeneous sources into graph database?

We answered these research questions using a mixed-methods research methodology (Johnson et al. 2007). Firstly, for addressing RQ1 a literature review was carried out to identify existing evidence and gaps in the literature related to the research question. We addressed RQ2 by proposing an algebra `FLASc` which is based on conceptual graphs introduced by (Sowa 2008, 1992, 1999). The three operators of `JOIN`, `DETACH` and `DELETE_NODE` provided by `FLASc` serve as mechanisms for formulating conceptual graph schemas which are further extended to logical graph schemas. The three `FLASc` operators presented in this research paper can be used for designing schema generation and manipulation algorithms. Hence a major utility of `FLASc` is that it serves as a formal basis for designing future data definition languages for graph databases. For addressing RQ3 and RQ3.1, we illustrate the integration of `FLASc` with the well known Extract-Transform-Load (ETL) design pattern. The graph schemas generated by `FLASc` can be used to enforce integrity constraints and assist in the systematic generation of database creation scripts hence ensuring data consistency. To demonstrate the utility of our approach we consider three distinct case studies related to industrial cyber-physical systems (Sharma et al. 2019), big data analytics (Khalajzadeh et al. 2019, 2020) and tourism (Airbnb 2018; Sharma and Sinha 2019). We generate graph schemas for the heterogeneous data-sets provided in the three case studies and produce database creation scripts in `Cypher` using the `FLASc` integrated ETL design pattern.

The critical contributions of this work include:

1. We formulate `FLASc` a formal algebra for constructing a labeled property graph schema that can capture data semantics of any given problem domain. We define operators of `FLASc` that assist in constructing a graph schema.
2. We demonstrate the use of graph schemas formulated via `FLASc` to enforce integrity constraints that ensure data consistency in contemporary labeled property graph databases such as Neo4j.
3. We illustrate how `FLASc` can be integrated with the Extract-Transform-Load design pattern for loading data-sets from heterogeneous sources into Neo4j.

Two case studies related to tourism and cyber physical systems, presented in Sects. 5.2 and 5.4 , have been adopted from our previously published research (Sharma et al. 2019; Sharma and Sinha 2019 and Sharma et al. 2021) respectively.

The formalism for labeled property graph schemas presented in Sharma and Sinha (2019) and Sharma et al. (2021) is foundational for designing our algebra FLASc. The work presented in this research paper empowers users of FLASc to design robust graph schemas for labeled property graph databases.

The rest of this article is organized as follows. Section 2 presents background information and related work. The gaps identified in Sect. 2 are used to build FLASc which is presented in Sect. 3. In Sect. 4 we illustrate how the conceptual and logical graph schema formulated using FLASc can be used to enforce several integrity constraints in Neo4j graph database. In Sect. 5 we present the integration of FLASc with ETL design pattern and experimentally demonstrate its use for data transformation and loading of heterogeneous data-sets into Neo4j graph database. Finally, in Sect. 6 we summarize our major findings, key contributions and future directions of this work.

## 2 Background and related work

This section enables us to address RQ1. We present a brief survey of the existing approaches that have been proposed for modeling graph databases.

### 2.1 Graph database design and modeling

Graph databases use graphs consisting of nodes and edges as elementary data structures for modeling any problem domain (Angles 2012; Angles et al. 2017; Angles and Gutierrez 2008). All graph databases use slight variations of the basic graph data structure. For example, graph databases proposed in academia such as GOOD (Gyssens et al. 1994), Gram (Amann and Scholl 1993), GraphDB (Güting 1994), GDM (Hidders 2003; Paredaens et al. 1995) and (Graves et al. 1995) use directed labeled graphs. Graph database such as hyper log (Levene and Poulovassilis 1990; Levene and Loizou 1995) use hyper node and hyper edge based graphs. Resource Description Framework (RDF) proposed by W3C (W3C 2021) use directed labeled graphs while Neo4j (2021), Oracle (2021) use directed, labeled and attributed graphs which are also known as property graphs (Angles 2018). There are three main stages of modeling a graph database: *conceptual, logical* and *physical.*

#### 2.1.1 Conceptual modeling

Conceptual modeling represents the initial stage in which knowledge is collected in the form of requirements and specifications related to a problem domain. Using graphs for representing knowledge was first proposed by Sowa (2008, 1992, 1976, 1999). Subsequent works (Kunii 1987; Chein and Mugnier 2008; Mugnier and Chein 1992) also propose the use of graphs to represent knowledge at the conceptual modeling stage. Graphs provide a natural and intuitive interface for understanding the semantics of data (Sowa 2008; Badia and Lemire 2011). Knowing the semantics of data is vital for understanding the overall structure of the database

(Pokornỳ 2016) that aids in creating, modifying and retrieving data. Schemas created at the conceptual modeling stage provide a level of abstraction that aids in the natural modeling of data (Angles 2012). Conceptual graph schemas are used to define entities that belong to the database and relationships between those entities (Badia and Lemire 2011). Moreover, determining nodes, edges, and the direction of edges are vital for conceptual modeling (Griffith 1982).

### 2.1.2 Logical modeling

Logical modeling is used to define integrity constraints on entities and relations of conceptual graph schema. Integrity constraints serve as mechanisms to ensure data consistency and integrity. They are broadly classified into two categories: *graph entity integrity* and *semantic constraints* (Ghrab et al. 2016). Graph entity integrity constraints are related to basic database design principles. These include constraints such as node/edge property uniqueness (Angles and Gutierrez 2008; Pokornỳ et al. 2017; Angles 2012; Ghrab et al. 2016; Barik et al. 2016), label uniqueness (Angles and Gutierrez 2008; Pokornỳ 2016; Angles 2012; Ghrab et al. 2016; Pokornỳ et al. 2017), property data type (Pokornỳ 2016; Barik et al. 2016) and mandatory property constraints (Ghrab et al. 2014; Pokornỳ 2016). Enforcing semantic constraints require knowledge of the problem domain captured in the conceptual graph schema. These constraints are used to guarantee the conformity of graph database with domain specific rules and require intervention from end users. These include edge pattern (Barik et al. 2016; Ghrab et al. 2016, 2014; Reina et al. 2020; Pokornỳ et al. 2017), graph pattern (Barik et al. 2016; Ghrab et al. 2016; Angles 2012; Ghrab et al. 2014) and path pattern constraints (Barik et al. 2016). Other constraints discussed in literature include type checking (Angles and Gutierrez 2008; Angles 2012; Ghrab et al. 2014), node/edge property value constraints (Reina et al. 2020), cardinality constraints (Pokornỳ 2016; Barik et al. 2016; Angles 2012; Ghrab et al. 2016; Reina et al. 2020; Šestak et al. 2021, 2016) and functional dependencies (Angles and Gutierrez 2008; Pokornỳ 2016; Angles 2012; Levene and Poulovassilis 1991; Yu and Heflin 2011; Megid et al. 2018).

### 2.1.3 Physical modeling

Physical modeling represents the realization of the graph schema designed during conceptual and logical modeling into actual database (Finkelstein et al. 1988). There are two approaches discussed in literature for physical modeling: *integrated* and *layered* (Šestak et al. 2016). In the integrated approach, mechanisms to support the enforcement of integrity constraints are directly deployed on the database. These mechanisms are developed by altering and/or modifying the source code of a database system. In the layered approach, APIs specific to the database platform are used to create an additional layer that communicates with the database. This consist of wrappers written in programming languages such as Java, Python that contains database creation scripts and logic to enforce the integrity constraints.

### 2.1.4 Integration of logical and physical modeling

There exist many studies to support the integration of logical and physical modeling aspects of graph databases. For instance, Ghrab et al. (2016) follow a layered approach and propose the construction of a wrapper that can be used to enforce integrity constraints, including graph and path pattern constraints over Neo4j graph database. An integrated approach to extend the source code of OrientDB to support the enforcement of integrity constraints, including uniqueness, key, cardinality, and edge degree constraints, has been studied in Reina et al. (2020). Similarly, the extension of Cypher query language to support additional integrity constraints such as uniqueness, node property, edges pattern, and mandatory properties is presented in Pokornŷ et al. (2017), de Sousa and Cura (2018). A layered approach to demonstrate the enforcement of uniqueness integrity constraint on two different graph databases Neo4j and Apache Tinkerpop, is proposed in Šestak et al. (2016). The use of integrated and layered approach together to perform graph database manipulation operations on Neo4j graph database is proposed in Barik et al. (2016). Authors in Daniel et al. (2016) propose the model-driven engineering based approach for converting and loading of UML diagrams into tinkerpop blueprints.[1]

## 2.2 Gaps in current literature

Several studies have been proposed in the last decade that address the problem of modeling graph databases. These studies mainly focus on the integration of logical and physical modeling aspects. A primary reason of this due to the emergence of several graph data models such as resource description framework (RDF) (Lassila et al. 1998; Pérez et al. 2006), labeled property graphs (LPG) (Angles 2018; Sharma et al. 2019; Sharma and Sinha 2019; Sharma 2020, 2021) and creation of query languages such as SPARQL (2013), Cypher (Neo4j) 2021, Gremlin (Apache) (2021), PGQL (Oracle) (2021) and GSQL (TigerGraph) (2020) to support data modeling and retrieval. More recently, projects such as ISO/IEC 39,075,[2] openCypher (2018) and Linked Data Benchmark Council (LDBC) Alex and Norbert (2013) have been proposed for developing a standard query language for the labeled property graph data model. Most of these studies focus on extending the existing query languages to support logical and physical modeling while conceptual modeling is done in an ad-hoc manner. Authors in Ghrab et al. (2016), Roy-Hubara et al. (2017), Hartig and Hidders (2019) present a formal approach for logical modeling of graph databases. However, physical modeling in these research papers are not discussed in detail

[1] https://github.com/tinkerpop/blueprints.

[2] https://www.iso.org/standard/76120.html.

(Šestak et al. 2021) and application of the proposed formalisms on real-world datasets are considered as future work.

To obtain a robust graph database that captures semantics of the problem domain conceptual modeling stage is vital. A sound conceptual graph schema ensures that logical and physical modeling stages are also robust (Mior et al. 2017). The graph data modeling approaches proposed so far do not provide the means to create robust conceptual graph schemas. Authors in Park et al. (2014), Roy-Hubara et al. (2017), Daniel et al. (2016) propose the use of existing visual modeling tools such as entity relationships diagrams (ERD) and unified modeling language (UML) for creating conceptual and logical graph schemas. The schemas generated by visual models such as UML diagrams are based on node-labeled graphs (Sharma and Sinha 2019) where only the nodes can have properties associated with them. According to Chen (1976), ERDs are based on node and edge labeled graphs where edges are also attributed. However, in order to support the creation of relational databases, attributed edges in ERDs have to be represented as strong and weak entities (or attributed nodes)[3]. Modeling tools such as ERD and UML are generic and while they can be used to model LPG schema, they do not capture subtleties like edge labels and attributes without carefully considered extensions. Our algebra `FLASc` directly supports LPG schemas that have labels and properties associated with nodes and edges (Sharma and Sinha 2019; Sharma et al. 2021). Both UML and ERD are semi-formal modeling tools whereas `FLASc` provides a formal basis for LPG schemas. This opens up the opportunity to define a `FLASc`-driven schema-generation language based on formal languages such as conjuntive queries and first order logic Sharma (2021). However, such extensions of `FLASc` are not in the scope of this research paper.

In this research, we present `FLASc` a formal tool that assists in the formulation of robust conceptual and logical graph schemas which is an advancement over existing studies in graph database modeling. The majority of integrity constraints presented in the existing studies can be specified in graph schemas generated by `FLASc`. Furthermore, syntax and semantics of `FLASc` presented in this study assist in its implementation at the physical modeling stage. `FLASc` assists in the integration of conceptual, logical and physical modeling stages which currently is lacking in graph database research.

## 3 `FLASc`: formal algebra for conceptual and logical graph schema

This section addresses RQ2, we present the formal algebra `FLASc` that assists in formulating conceptual and logical graph schemas for labeled property graph databases. We use the concepts from Sowa's conceptual graphs identified in Sect. 2.1.1 to propose the operators of `FLASc`. We use a formal approach for constructing `FLASc` which assures the robustness of its design (Marciniak 1994; Clarke and Wing 1996). `FLASc` has sound mathematical basis that enables a user to precisely

[3] Interested readers can refer to Chen's research paper Chen (1976) for further clarification.

define: *(i)* connections between entities of a graph database (intensional information) and *(ii)* properties associated with entities and relations in a graph database (extensional information) (Sowa 1976, 1992, 1999, 2008).

We consider a data-set from Airbnb Sharma and Sinha (2019); Sharma et al. (2021) as our first case study related to the tourism domain that assists in illustrating various definitions and concepts of `FLASc`. This data-set consists of three CSV files that contain information related to property listings, reviews and calendar data. This data-set is highly interconnected, making it a prime candidate for graph database design and implementation (Sharma et al. 2021; Sharma 2021).

### 3.1 Basic terminology

**Definition 1** (Directed Multigraph) A directed multigraph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \mathcal{S}, \mathcal{T})$ is a tuple where $\mathcal{N}$ is a set of nodes and $\mathcal{E}$ is a set of edges. Two associated functions, $\mathcal{S} : \mathcal{E} \rightarrow \mathcal{N}$ and $\mathcal{T} : \mathcal{E} \rightarrow \mathcal{N}$, map each edge to its *source* and *target* nodes, respectively.

Each edge in a *directed* multigraph has unique source and target nodes. Edges with same source and target nodes are allowed (hence the term *multi*graph. We use the short hand $n_i \rightarrow n_j$ to represent an edge $e_k$ where $\mathcal{S}(e_k) = n_i$ and $\mathcal{T}(e_k) = n_j$.

Graph can contain labels over nodes and edges. Given a set of node labels $L_\mathcal{N}$ and a set of edge labels $L_\mathcal{E}$ such that $L_\mathcal{N} \cap L_\mathcal{E} = \emptyset$. A labeling is simply a map $f : S_1 \rightarrow S_2$ such that for every element $a \in S_1$, there is a unique element $f(a) \in S_2$. We can define an *edge-* labeled graph as follows.

**Definition 2** (Edge-Labeled Graph) A graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \xi, \mathcal{S}, \mathcal{T})$ is called an edge-labeled graph if there exists a labeling $\xi : \mathcal{E} \rightarrow L_\mathcal{E}$ which maps all edges to labels in a set of edge labels $L_\mathcal{E}$. We use the short-hand $e_k = n_i \xrightarrow{l} n_j$ for any $e_k \in \mathcal{E}$ and $\xi(e_k) = l$.

Similarly, we can define a node labeled graph.

**Definition 3** (Node-Labeled Graph) A graph $\mathcal{G} = (\mathcal{N}, \mathcal{E}, \eta, \mathcal{S}, \mathcal{T})$ is called a node-labeled graph if there exists a labeling $\eta : \mathcal{N} \rightarrow L_\mathcal{N}$ which maps all nodes to labels in a set of node labels $L_\mathcal{N}$ for any $n_i \in \mathcal{N}$ and $l \in L_\mathcal{N}$ if $l$ is mapped to $n_i$ then $\eta(n_i) = l$.

### 3.2 Conceptual graph schema

A conceptual graph schema is used to capture intensional information. Conceptual modeling is easier for the user to understand and contribute. Therefore, a conceptual graph schema must be closer to the semantics of natural languages like English. It must reflect real-world entities, and relations that are not directly represented by the conceptual graph schema must be accessible to infer (Sowa 1992; Mugnier and Chein 1992). As discussed in Sharma and Sinha (2019) to define relationships, we

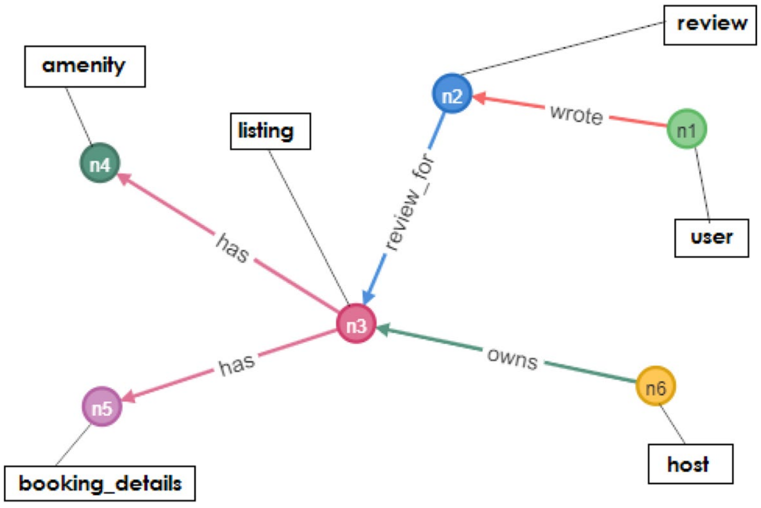

**Fig. 1** Conceptual graph schema generated for Airbnb case study

use the `(subject,predicate,object)` format from semantics web (Berners-Lee et al. 2001) where the subject can be a noun, the predicate can be a verb, and an object can also be a noun.

**Definition 4** (Conceptual graph schema)
Given a set of node labels $L_{\mathcal{N}}$ and a set of edge labels $L_{\mathcal{E}}$, conceptual graph schema $\mathcal{G}_s$ is a tuple $(\mathcal{N}_s, \mathcal{E}_s, \eta_s, \xi_s, L_{\mathcal{N}}, L_{\mathcal{E}}, \mathcal{S}_s, \mathcal{T}_s)$ where,

- $\mathcal{N}_s$ is a finite set of nodes and $\mathcal{E}_s$ is a finite set of edges of the graph schema.
- $(\mathcal{N}_s, \mathcal{E}_s, \mathcal{S}_s, \mathcal{T}_s)$ is a directed multigraph.
- $\eta_s : \mathcal{N}_s \rightarrow L_{\mathcal{N}}$ is a node labeling function and $\xi_s : \mathcal{E}_s \rightarrow L_{\mathcal{E}}$ is an edge labeling function.

We use the shorthand notation $\mathcal{G}_s = (\mathcal{N}_s, \mathcal{E}_s, \eta_s, \xi_s, \mathcal{S}_s, \mathcal{T}_s)$ to represent the conceptual graph schema.

***Example 1*** The conceptual graph schema generated for Airbnb case study as discussed in Sharma and Sinha (2019) is presented in Fig. 1. The graph schema consists of six labels including `review`, `user`, `host` and `listing` and four edge labels `wrote`, `review_for`, `has` and `owns`. In the Airbnb data-set (2018) a person using Airbnb service can write a review for a listing that was recently visited by him or her. A conceptual graph schema in such a scenario consists of entities such as `user` and `review`. Relationships can be of the form `(users,wrote,review)` where `users` is the subject, `wrote` is the verb and `review` is the object.

### 3.2.1 Basic conceptual graph schema

Basic conceptual graph schemas are restricted form of conceptual graph schemas. They serve as building blocks for formulating conceptual graph schemas. Formally basic conceptual graph schemas are defined as follows.

**Definition 5** (Basic conceptual graph schema) Given sets of node and edge labels $L_{\mathcal{N}}$ and $L_{\mathcal{E}}$, a basic conceptual graph schema $\mathcal{G}_b$ is a tuple $(\mathcal{N}_b, \mathcal{E}_b, \eta_b, \xi_b, \mathcal{S}_b, \mathcal{T}_b)$ where

- $\mathcal{N}_b = \{n_i, n_j\}$ is a set of two nodes.
- $\mathcal{E}_b = \{e_k\} \cup \emptyset$ can either be a singleton set or an empty set.
- $(\mathcal{N}_b, \mathcal{E}_b, \mathcal{S}_b, \mathcal{T}_b)$ is a restricted from of directed multigraph supporting only one directed edge between two nodes.
- $\eta_b : \mathcal{N}_b \to L_{\mathcal{N}}$ is a node labeling function and $\xi_b : \mathcal{E}_b \to L_{\mathcal{E}}$ is an edge labeling function.

***Example 2*** The Airbnb data-set consists of several basic conceptual graph schemas including $\mathcal{G}_{b1} = \left(\{n_1, n_2\}, \{n_1 \xrightarrow{\texttt{wrote}} n_2\}, \eta_1, \xi_1\right)$ such that $\eta_{b1}(n_1) = \texttt{user}$, $\eta_1(n_2) = \texttt{review}$ and $\xi_1(n_1 \xrightarrow{\texttt{wrote}} n_2) = \texttt{wrote}$. Similarly $\mathcal{G}_{b2} = \left(\{n_2, n_3\}, \{n_2 \xrightarrow{\texttt{review\_for}} n_3\}, \eta_2, \xi_2\right)$ such that $\eta_2(n_2) = \texttt{review}$, $\eta_2(n_3) = \texttt{listing}$ and $\xi_2(n_2 \xrightarrow{review\_for} n_3)$. The basic conceptual graph schema is used to represent the intensional information that a review was written by a user and review was written for a listing.

Basic conceptual graph schemas serve as a starting point for a database designer and assist in conceptual modeling. A basic conceptual graph schema can contain nodes that are not connected to one another by an edge. A designer can create separate basic conceptual graph schemas for each requirement and/or use case. We now present our algebra FLASc for creating robust conceptual graph schemas from basic conceptual graph schemas.

### 3.2.2 Syntax and semantics of FLASc

An algebra consists of sets, constants that belong to the sets and some functions or operators that are used to manipulate data stored inside the sets (Tucker and Stephenson 2003). Our algebra FLASc is defined as follows:

**Definition 6** ( FLASc ) An algebra defined over a finite set of basic conceptual graph schemas $\mathcal{G}_B$, is a tuple $\langle \mathcal{G}_B, \mathcal{G}, \mathcal{F} \rangle$ where:

- $\mathcal{G}$ is the set of all conceptual graph schemas over $\mathcal{G}_B$, with $\mathcal{G}_B \subset \mathcal{G}$.
- $\mathcal{F}$ is a set containing three operators:

1. `JOIN`: $\mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ is a binary operator such that if $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}$ then `JOIN` $(\mathcal{G}_1, \mathcal{G}_2)$ is a conceptual graph schema formed by the *union* of two conceptual graph schemas. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1, \mathcal{S}_1, \mathcal{T}_1)$ where $L_{\mathcal{N}_1}$ is a set of node labels and $L_{\mathcal{E}_1}$ is a set of edge labels associated with $\mathcal{G}_1$. Let $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2, \mathcal{S}_2, \mathcal{T}_2)$ where $L_{\mathcal{N}_2}$ is a set of node labels and $L_{\mathcal{E}_2}$ is a set of edge labels associated with $\mathcal{G}_2$. Then $\texttt{JOIN}(\mathcal{G}_1, \mathcal{G}_2) = \mathcal{G}_3 = \left(\mathcal{N}_3, \mathcal{E}_3, \eta_3, \xi_3, \mathcal{S}_3, \mathcal{T}_3\right)$ such that

   - $\mathcal{N}_3 = \mathcal{N}_1 \cup \mathcal{N}_2$ and $\mathcal{E}_3 = \mathcal{E}_1 \cup \mathcal{E}_2$.
   - $\eta_3 = \eta_1 \cup \eta_2$ where $\eta_3 : (\mathcal{N}_1 \cup \mathcal{N}_2) \rightarrow (L_{\mathcal{N}_1} \cup L_{\mathcal{N}_2})$ such that

     If $n_i \in \mathcal{N}_1$ then $\eta_3(n_i) = \eta_1(n_i) = l_{n_1}$ and $l_{n_1} \in L_{\mathcal{N}_1}$.
     If $n_i \in \mathcal{N}_2$ then $\eta_3(n_i) = \eta_2(n_i) = l_{n_1}$ and $l_{n_1} \in L_{\mathcal{N}_2}$.
     If $n_i \in (\mathcal{N}_1 \cap \mathcal{N}_2)$ then $\eta_3(n_i) = \eta_1(n_i) = \eta_2(n_i) = l_{n_1}$ and $l_{n_1} \in (L_{\mathcal{N}_1} \cap L_{\mathcal{N}_2})$.

   - $\xi_3 = \xi_1 \cup \xi_2$ where $\xi_3 : (\mathcal{E}_1 \cup \mathcal{E}_2) \rightarrow (L_{\mathcal{E}_1} \cup L_{\mathcal{E}_2})$ such that that

     If $e_i \in \mathcal{E}_1$ then $\xi_3(e_i) = \xi_1(e_i) = l_{e_1}$ and $l_{e_1} \in L_{\mathcal{E}_1}$.
     If $e_i \in \mathcal{E}_2$ then $\xi_3(e_i) = \xi_2(e_i) = l_{e_1}$ and $l_{e_1} \in L_{\mathcal{E}_2}$.
     If $e_i \in (\mathcal{E}_1 \cap \mathcal{E}_2)$ then $\xi_3(e_i) = \xi_1(e_i) = \xi_2(e_i) = l_{e_1}$ and $l_{e_1} \in (L_{\mathcal{E}_1} \cap L_{\mathcal{E}_2})$.

   - $\mathcal{S}_3 = \mathcal{S}_1 \cup \mathcal{S}_2$ where $\mathcal{S}_3 : (\mathcal{E}_1 \cup \mathcal{E}_2) \rightarrow (\mathcal{N}_1 \cup \mathcal{N}_2)$ such that

     If $e_i \in \mathcal{E}_1$ then $\mathcal{S}_3(e_i) = \mathcal{S}_1(e_i) = n_i$ and $n_i \in \mathcal{N}_1$.
     If $e_i \in \mathcal{E}_2$ then $\mathcal{S}_3(e_i) = \mathcal{S}_2(e_i) = n_i$ and $n_i \in \mathcal{N}_2$.
     If $e_i \in (\mathcal{E}_1 \cap \mathcal{E}_2)$ then $\mathcal{S}_3(e_i) = \mathcal{S}_1(e_i) = \mathcal{S}_2(e_i) = n_i$ and $n_i \in (\mathcal{N}_1 \cap \mathcal{N}_2)$.

   - $\mathcal{T}_3 = \mathcal{T}_1 \cup \mathcal{T}_2$ where $\mathcal{T}_3 : (\mathcal{E}_1 \cup \mathcal{E}_2) \rightarrow (\mathcal{N}_1 \cup \mathcal{N}_2)$ such that

     If $e_i \in \mathcal{E}_1$ then $\mathcal{T}_3(e_i) = \mathcal{T}_1(e_i) = n_j$ and $n_j \in \mathcal{N}_1$.
     If $e_i \in \mathcal{E}_2$ then $\mathcal{T}_3(e_i) = \mathcal{T}_2(e_i) = n_j$ and $n_j \in \mathcal{N}_2$.
     If $e_i \in (\mathcal{E}_1 \cap \mathcal{E}_2)$ then $\mathcal{T}_3(e_i) = \mathcal{T}_1(e_i) = \mathcal{T}_2(e_i) = n_j$ and $n_j \in (\mathcal{N}_1 \cap \mathcal{N}_2)$.

2. `DETACH`: $\mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ is a binary operator such that if $\mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}$ then `DETACH` $(\mathcal{G}_1, \mathcal{G}_2)$ is a conceptual graph schema formed by applying ring sum over the edge sets of $\mathcal{G}_1$ and $\mathcal{G}_2$. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1, \mathcal{S}_1, \mathcal{T}_1)$ where $L_{\mathcal{N}_1}$ is a set of node labels and $L_{\mathcal{E}_1}$ is a set of edge labels associated with $\mathcal{G}_1$. Let $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2, \mathcal{S}_2, \mathcal{T}_2)$ where $L_{\mathcal{N}_2}$ is a set of node labels and $L_{\mathcal{E}_2}$ is a set of edge labels associated with $\mathcal{G}_2$. The resultant conceptual graph schema consists of all the nodes present in graphs $\mathcal{G}_1$ and $\mathcal{G}_2$ that is $(\mathcal{N}_1 \cup \mathcal{N}_2)$. While the ring sum operator is only applied over the edge sets of two graphs that is $(\mathcal{E}_1 \oplus \mathcal{E}_2) = (\mathcal{E}_1 \cup \mathcal{E}_2) - (\mathcal{E}_1 \cap \mathcal{E}_2)$. $\texttt{DETACH}(\mathcal{G}_1, \mathcal{G}_2) = \mathcal{G}_3 = \left(\mathcal{N}_3, \mathcal{E}_3, \eta_3, \xi_3, \mathcal{S}_3, \mathcal{T}_3\right)$ such that

   - $\mathcal{N}_3 = \mathcal{N}_1 \cup \mathcal{N}_2$ and $\mathcal{E}_3 = \mathcal{E}_1 \oplus \mathcal{E}_2$ if $\mathcal{E}_1 \cap \mathcal{E}_2 = \emptyset$ then $\mathcal{E}_3 = \mathcal{E}_1 \cup \mathcal{E}_2$
   - $\eta_3 = \eta_1 \cup \eta_2$ where $\eta_3 : (\mathcal{N}_1 \cup \mathcal{N}_2) \rightarrow (L_{\mathcal{N}_1} \cup L_{\mathcal{N}_2})$ such that

     If $n_i \in \mathcal{N}_1$ then $\eta_3(n_i) = \eta_1(n_i) = l_{n_1}$ and $l_{n_1} \in L_{\mathcal{N}_1}$.
     If $n_i \in \mathcal{N}_2$ then $\eta_3(n_i) = \eta_2(n_i) = l_{n_1}$ and $l_{n_1} \in L_{\mathcal{N}_2}$.
     If $n_i \in (\mathcal{N}_1 \cap \mathcal{N}_2)$ then $\eta_3(n_i) = \eta_1(n_i) = \eta_2(n_i) = l_{n_1}$ and $l_{n_1} \in (L_{\mathcal{N}_1} \cap L_{\mathcal{N}_2})$.

   - $\xi_3$ is defined as $\xi_3 : (\mathcal{E}_1 \oplus \mathcal{E}_2) \rightarrow (L_{\mathcal{E}_1} \oplus L_{\mathcal{E}_2})$ such that

     If $\mathcal{E}_1 \cap \mathcal{E}_2 \neq \emptyset$ and $e_i \in (\mathcal{E}_1 \cap \mathcal{E}_2)$ then $\xi_3(e_i) = \emptyset$.

Otherwise if $e_i \in \mathcal{E}_1$ then $\xi_3(e_i) = \xi_1(e_i) = l_{e_1}$ and $l_{e_1} \in L_{\mathcal{E}_1}$. If $e_i \in \mathcal{E}_2$ then $\xi_3(e_i) = \xi_2(e_i) = l_{e_1}$ and $l_{e_1} \in L_{\mathcal{E}_2}$.

- $\mathcal{S}_3$ is defined as $\mathcal{S}_3 : (\mathcal{E}_1 \oplus \mathcal{E}_2) \rightarrow (\mathcal{N}_1 \cup \mathcal{N}_2)$ such that

  If $(\mathcal{E}_1 \cap \mathcal{E}_2) \neq \emptyset$ and $e_i \in (\mathcal{E}_1 \cap \mathcal{E}_2)$ then $\mathcal{S}_3(e_i) = \emptyset$.
  Otherwise if $e_i \in \mathcal{E}_1$ then $\mathcal{S}_3(e_i) = \mathcal{S}_1(e_i) = n_i$ and $n_i \in \mathcal{N}_1$. If $e_i \in \mathcal{E}_2$ then $\mathcal{S}_3(e_i) = \mathcal{S}_2(e_i) = n_i$ and $n_i \in \mathcal{N}_2$.

- $\mathcal{T}_3$ is defined as $\mathcal{T}_3 : (\mathcal{E}_1 \oplus \mathcal{E}_2) \rightarrow (\mathcal{N}_1 \cup \mathcal{N}_2)$ such that

  If $(\mathcal{E}_1 \cap \mathcal{E}_2) \neq \emptyset$ and $e_i \in (\mathcal{E}_1 \cap \mathcal{E}_2)$ then $\mathcal{T}_3(e_i) = \emptyset$.
  Otherwise if $e_i \in \mathcal{E}_1$ then $\mathcal{T}_3(e_i) = \mathcal{T}_1(e_i) = n_i$ and $n_i \in \mathcal{N}_1$. If $e_i \in \mathcal{E}_2$ then $\mathcal{T}_3(e_i) = \mathcal{T}_2(e_i) = n_i$ and $n_i \in \mathcal{N}_2$.

3. `DELETE_NODE`: $\mathcal{G} \times \mathcal{G} \rightarrow \mathcal{G}$ is a binary operator such that if $\mathcal{G}_1, \mathcal{G}_d \in \mathcal{G}$ then `DELETE_NODE`$(\mathcal{G}_1, \mathcal{G}_d)$ is a conceptual graph schema formed by applying ring sum over the node sets of $\mathcal{G}_1$ and $\mathcal{G}_d$. Let $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1, \mathcal{S}_1, \mathcal{T}_1)$ where $L_{\mathcal{N}_1}$ is a set of node labels and $L_{\mathcal{E}_1}$ is a set of edge labels associated with $\mathcal{G}_1$. Let $\mathcal{G}_d = (\mathcal{N}_d, \mathcal{E}_d, \eta_d, \mathcal{S}_d, \mathcal{T}_d)$ is a node labeled graph where $L_{\mathcal{N}_d}$ is a set of node labels associated with $\mathcal{G}_d$. Furthermore, the graph $\mathcal{G}_d$ has no edges associated with it that is $\mathcal{E}_d = \emptyset$ subsequently, $\mathcal{S}_d = \emptyset$ and $\mathcal{T}_d = \emptyset$. Then the resultant conceptual graph schema after applying the `DELETE_NODE` operator consist of nodes that are formed by applying the ring sum over the node sets of two graphs that is $(\mathcal{N}_1 \oplus \mathcal{N}_d) = (\mathcal{N}_1 \cup \mathcal{N}_d) - (\mathcal{N}_1 \cap \mathcal{N}_d)$. The set of edges in the conceptual graph schema `DELETE_NODE`$(\mathcal{G}_1, \mathcal{G}_d)$ is equal to the set of edges in $\mathcal{G}_1$ that is $\mathcal{E}_1$. `DELETE_NODE`$(\mathcal{G}_1, \mathcal{G}_d) = \mathcal{G}_3 = (\mathcal{N}_3, \mathcal{E}_3, \eta_3, \xi_3, \mathcal{S}_3, \mathcal{T}_3)$ such that

- $\mathcal{N}_3 = (\mathcal{N}_1 \oplus \mathcal{N}_d)$ if $(\mathcal{N}_1 \cap \mathcal{N}_d) = \emptyset$ then $\mathcal{N}_3 = (\mathcal{N}_1 \cup \mathcal{N}_d)$ and $\mathcal{E}_3 = \mathcal{E}_1$.
- $\eta_3$ is defined as $\eta_3 : (\mathcal{N}_1 \oplus \mathcal{N}_d) \rightarrow (L_{\mathcal{N}_1} \oplus L_{\mathcal{N}_d})$ such that

  If $(\mathcal{N}_1 \cap \mathcal{N}_d) \neq \emptyset$ and $n_i \in (\mathcal{N}_1 \cap \mathcal{N}_d)$ then $\eta_3(n_i) = \emptyset$
  Otherwise, if $n_i \in \mathcal{N}_1$ then $\eta_3(n_i) = \eta_1(n_i) = l_{n_i}$ and $l_{n_i} \in L_{\mathcal{N}_1}$. If $n_i \in \mathcal{N}_d$ then $\eta_3(n_i) = \eta_d(n_i) = l_{n_i}$ and $l_{n_i} \in L_{\mathcal{N}_d}$.

- $\xi_3 = \xi_1$ such that $\xi_3 : \mathcal{E}_1 \rightarrow L_{\mathcal{E}_1}$.
- $\mathcal{S}_3 = \mathcal{S}_1$ such that $\mathcal{S}_3 : \mathcal{E}_1 \rightarrow \mathcal{N}_1$.
- $\mathcal{T}_3 = \mathcal{T}_1$ such that $\mathcal{T}_3 : \mathcal{E}_1 \rightarrow \mathcal{N}_1$.

`FLASc` provides `JOIN`, `DETACH` and `DELETE_NODE` operators over basic conceptual graph schemas to formulate composite conceptual graph schemas. We can now discuss the semantics of these three operators and provide some examples.

`JOIN` is used to combine together two or more conceptual graph schemas. We follow the similar notion of join compatible mapping as discussed in Angles et al. (2017); Castro and Soto (2017); Pérez et al. (2006). Two conceptual graph schemas are join compatible if they share common nodes. That is $\mathcal{G}_1 = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1, \mathcal{S}_1, \mathcal{T}_1)$ and $\mathcal{G}_2 = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2, \mathcal{S}_2, \mathcal{T}_2)$ are join compatible if $\exists e_i \in \mathcal{E}_1$ and $\exists e_j \in \mathcal{E}_2$ such that *either* $\mathcal{S}_1(e_i) = \mathcal{T}_2(e_j)$ *or* $\mathcal{T}_1(e_i) = \mathcal{S}_2(e_j)$ *or*

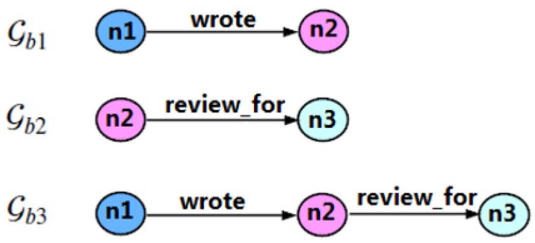

**Fig. 2** The application of `JOIN` operator to connect two conceptual graph schemas

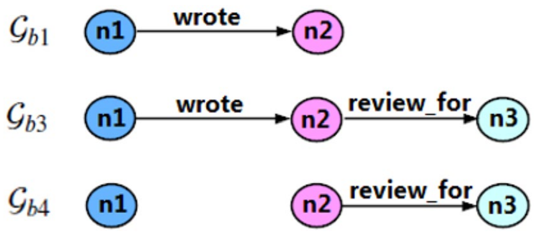

**Fig. 3** The application of `DETACH` operator to delete an edge from a conceptual graph schemas

$\mathcal{S}_1(e_i) = \mathcal{S}_2(e_j)\ or\ \mathcal{T}_1(e_i) = \mathcal{T}_2(e_j)$. Furthermore, if $\mathcal{S}_1(e_i)\ or\ \mathcal{T}_1(e_i) = n_i$ and $\mathcal{S}_2(e_j)\ or\ \mathcal{T}_2(e_j) = n_j$ then $\eta_1(n_i) = \eta_2(n_j)$.

***Example 3*** The basic conceptual graph schemas presented in Example 2 are join compatible because both graphs share a common node $n_2$ that have the node label `review`. Figure 2 shows that applying the `JOIN` operator over basic conceptual graph schemas $\mathcal{G}_{b1} = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1, \mathcal{S}_1, \mathcal{T}_1)$ and $\mathcal{G}_{b2} = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2, \mathcal{S}_2, \mathcal{T}_2)$ creates a conceptual graph schema $\mathcal{G}_{b3} =$ `JOIN` $(\mathcal{G}_{b1}, \mathcal{G}_{b2})$. Graphs $\mathcal{G}_{b1}$ and $\mathcal{G}_{b2}$ are join compatible because the target node of edge $e_1 \in \mathcal{E}_1$ that is $\mathcal{T}_1(e_1)$ and source node of edge $e_2 \in \mathcal{E}_2$ that is $\mathcal{S}_2(e_2)$ are same. Moreover the node labels associated with these two nodes are also same that is $\eta_1(\mathcal{T}_1(e_1)) = \eta_2(\mathcal{S}_2(e_2)) =$ `review`.

Two join compatible conceptual graphs share common nodes. This assists in connecting smaller graphs. When two conceptual graph schemas are not join compatible, then application of the `JOIN` operator creates a union of two disconnected conceptual graph schemas.

`DETACH` is used to delete edges from a conceptual graph schema. This operator is useful if a database designer wishes to delete existing relationships in a conceptual graph schema. The graph produced after applying a `DETACH` operator over two conceptual graph schemas contain nodes from both the graphs. While edges of the new conceptual graph schema are calculated by applying the ring sum operator over the edges of conceptual graph schemas that provided as input to the `DETACH` operator. Applying the `DETACH` operator over two conceptual graph schemas $\mathcal{G}_{b1} = (\mathcal{N}_1, \mathcal{E}_1, \eta_1, \xi_1, \mathcal{S}_1, \mathcal{T}_1)$ and $\mathcal{G}_{b2} = (\mathcal{N}_2, \mathcal{E}_2, \eta_2, \xi_2, \mathcal{S}_2, \mathcal{T}_2)$ creates a conceptual graph schema $\mathcal{G}_{b3} =$ `DETACH` $(\mathcal{G}_{b1}, \mathcal{G}_{b2})$. If one graph is a sub-graph of another conceptual graph schema then applying `DETACH` operator over such

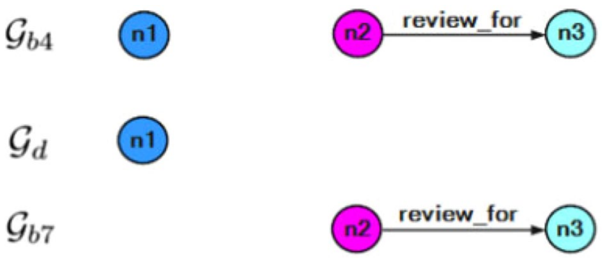

**Fig. 4** The application of `DELETE_NODE` operator to delete a node from a conceptual graph schemas

graph represents set difference of the edge set. An edge can only be deleted using DETACH if $(\mathcal{E}_1 \cap \mathcal{E}_2) \neq \emptyset$ which means that both conceptual graph schema must share some common edges. Furthermore, the labels associated with these edges must be same that is, $\exists e_1 \in \mathcal{E}_1$ and $\exists e_2 \in \mathcal{E}_2$ such that $\xi_1(e_1) = \xi_2(e_2)$. The application of DETACH removes existing edges from a conceptual graph schema. The resulting conceptual graph schemas after the application of DETACH may contain disconnected nodes.

***Example 4*** Edges can be deleted from a conceptual graph schema by using DETACH. As shown in Figure 3 applying DETACH between conceptual graph schemas $\mathcal{G}_{b1}$ and $\mathcal{G}_{b3}$ results in conceptual graph schema $\mathcal{G}_{b4}$ that only contains an edge between node $n_2$ and $n_3$. That is $\mathcal{G}_{b4} = \texttt{DETACH}\ (\mathcal{G}_{b1}, \mathcal{G}_{b3})$ such that $\eta(n_1) = \texttt{user}, \eta(n_2) = \texttt{review}$ and $\eta(n_3) = \texttt{listing}$. Furthermore, node $n_1$ is not the source and target of any edge in the conceptual graph schema.

DELETE_NODE is used to delete disconnected nodes in a conceptual graph schema. This operator is useful if a database designer wishes to delete existing nodes that are not connected to any other nodes in a conceptual graph schema. That is nodes that are neither the source nor the target of any edge in a conceptual graph schema. As mentioned in Definition 6 the set of nodes in $\mathcal{G}_3 = \texttt{DELETE\_NODE}(\mathcal{G}_1, \mathcal{G}_d)$ is $\mathcal{N}_3 = (\mathcal{N}_1 \oplus \mathcal{N}_d)$. A node $n_i \in \mathcal{N}_1$ can only be deleted by using the DELETE_NODE operator if $\forall e \in \mathcal{E}_1$ and $\mathcal{E}_1 \in \mathcal{G}_1$, $\mathcal{S}_1(e) \neq n_i, \mathcal{T}_1(e) \neq n_i$ moreover, $(\mathcal{N}_1 \cap \mathcal{N}_d) \neq \emptyset$. This means that both graph must share common nodes, furthermore $\forall n_i \in \mathcal{N}_1$ and $\forall n_d \in \mathcal{N}_d$ such that $\eta_1(n_i) = \eta_d(n_d)$ which means that both nodes must have same node label. Otherwise, all nodes in $\mathcal{N}_d$ shall be added to the conceptual graph schema resulting from $\texttt{DELETE\_NODE}(\mathcal{G}_1, \mathcal{G}_d)$.

***Example 5*** Disconnected nodes can be deleted from a conceptual graph schema by using the DELETE_NODE. As shown in Fig. 4 applying the DELETE_NODE operator between conceptual graph schemas $\mathcal{G}_{b4}$ and $\mathcal{G}_d$ results in a conceptual graph schema $\mathcal{G}_{b7}$ that only consists of nodes $n_2, n_3$ and an edge connecting nodes $n_2$ and $n_3$. The resulting graph does not contain any disconnected node. That is $\mathcal{G}_{b7} = \texttt{DELETE\_NODE}(\mathcal{G}_{b4}, \mathcal{G}_d)$ such that $\eta(n_2) = \texttt{review}$ and $\eta(n_3) = \texttt{listing}$. The

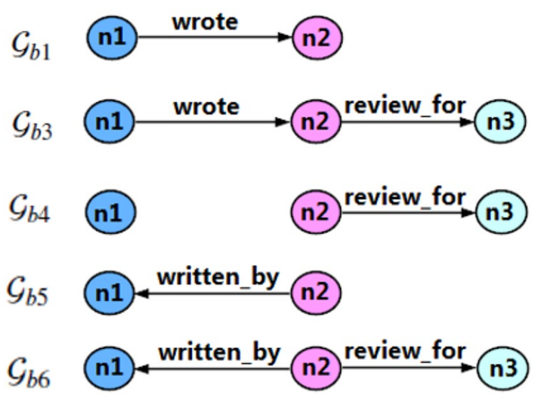

**Fig. 5** The application of `JOIN` and `DETACH` operators to alter an existing edge

graph $\mathcal{G}_d$ only consists of a node $n_1$ such that $\eta(n_1) = \texttt{user}$ and this node has been removed from the conceptual graph schema $\mathcal{G}_{b4}$.

Using `JOIN` and `DETACH` together become helpful if the label and/or direction of edges in a conceptual graph schema have to be altered or changed. These operators, when used together, enables a designer to alter intensional information stored in a conceptual graph schema.

***Example 6*** For instance if a designer wishes to alter the label and direction of an edge between node $n_1$ labeled as `user` and node $n_2$ labeled as `review` in the conceptual graph schema $\mathcal{G}_{b3}$ presented in Example 3. As shown in Fig. 5 a designer can apply `DETACH` between graphs $\mathcal{G}_{b1}$ and $\mathcal{G}_{b3}$ which results in graph $\mathcal{G}_{b4} = \texttt{DETACH}(\mathcal{G}_{b3}, \mathcal{G}_{b1})$. The designer can now define a basic conceptual graph schema $\mathcal{G}_{b5}$ where $\eta(n_1) = \texttt{user}$ and $\eta(n_2) = \texttt{review}$. Applying the `JOIN` operator between graphs $\mathcal{G}_{b4}$ and $\mathcal{G}_{b5}$ results in conceptual graph schema $\mathcal{G}_{b6} = \texttt{JOIN}(\mathcal{G}_{b4}, \mathcal{G}_{b5})$ as shown in Figure 5.

### 3.3 Logical graph schema

A logical graph schema is used to capture extensional information of the entities and relations stored in a graph database. A logical graph schema is formed by enforcing integrity constraints on conceptual graph schema. Label uniqueness constraints are automatically enforced in the logical graph schema since the node, and edge labels used in conceptual graph schema are unique. For defining property-based constraints, we first define properties that can exist in graph databases. Properties in graph databases exist as key-value pairs where property values are atomic entities and have an associated data type. Logical graph schema stores properties as key-type format. Properties can be mandatory as well as optional for instance, properties such as ids must be unique. This information must be stored in a logical graph schema.

Let $\texttt{K}_s$ be a set of infinite keys (e.g., id, name, age, etc.) and $\texttt{T}_s$ be a finite set of data types (e.g., String, Integer, etc.) We define a set of properties $\texttt{P}_s \subseteq (\texttt{K}_s \times \texttt{T}_s)$. The property set is of two types *(i) mandatory property set* ($\texttt{P}_m$) and *(ii) optional* *property set* ($\mathtt{P}_o$) such that $\mathtt{P}_s = \mathtt{P}_m \cup \mathtt{P}_o$. Mandatory property set can have some properties that have unique values associated with them. Let $\mathtt{U}$ be a set of Boolean values, we define a uniqueness function $\mathcal{U} : \mathtt{P}_m \rightarrow \mathtt{U}$ that maps elements from mandatory property set to `TRUE` or `FALSE` signifying that some values associated with a mandatory property must be unique.

Edges in a graph schema also have semantic information such as cardinality associated with them which refers to total number of edges that can exist between any two given nodes of a graph database. Cardinality of an edge represents a range where the minimum value of cardinality refers to minimum number of edges that must exist between any two nodes of a graph databases. Similarly, maximum value of cardinality refers to maximum number of edges that can exist between any two nodes in a graph database.

Let $\mathtt{MIN} \in \mathbb{W}$ represent a minimum cardinality set which belongs to a set of whole numbers. Let $\mathtt{MAX} \in \mathbb{N}$ represents a maximum cardinality set which belongs to a set of natural numbers. We define a set of cardinalities as $\mathtt{C} \subseteq (\mathtt{MIN} \times \mathtt{MAX})$ with a condition that if $\mathtt{min} \in \mathtt{MIN}$ and $\mathtt{max} \in \mathtt{MAX}$ then $\mathtt{min} \leq \mathtt{max}$. This means that minimum cardinality can never be greater than maximum cardinality. The minimum cardinality belongs to a set of whole numbers which means that minimum cardinality can be zero. On the other hand maximum cardinality belongs to a set of natural numbers therefore, the smallest value that can be associated with maximum cardinality is 1. Furthermore, in such a scenario minimum cardinality can be either 0 or 1.

A logical graph schema extends the conceptual graph schema discussed in Definition 4 by labeling the nodes and edges with mandatory and optional properties. Moreover, in a logical graph schema edges are labeled with cardinality values. Formally, a logical graph schema is defined as follows:

**Definition 7** (Logical graph schema) A logical graph schema $\mathcal{G}_l$ is a tuple $(\mathcal{N}_s, \mathcal{E}_s, \mathtt{P}_m, \mathtt{P}_o, \mathtt{C}_s, \eta_s, \xi_s, \mathcal{S}_s, \mathcal{T}_s, \Delta_m, \Delta_o, \zeta_s)$ where,

- $(\mathcal{N}_s, \mathcal{E}_s, \eta_s, \xi_s, \mathcal{S}_s, \mathcal{T}_s)$ is a conceptual graph schema as presented in Definition 4.
- $\mathtt{C}_s$ is a set of cardinalities such that $\mathtt{C}_s \subseteq (\mathtt{MIN} \times \mathtt{MAX})$ where $\mathtt{MIN} \in \mathbb{W}$ and $\mathtt{MAX} \in \mathbb{N}$.
- $\Delta_m : (\mathcal{N}_s \cup \mathcal{E}_s) \rightarrow \mathcal{P}^+(\mathtt{P}_m)$ is a mandatory property labeling function that maps all nodes and edges to the non empty subset of the mandatory property set where $\mathcal{P}^+(\mathtt{P}_m)$ represents the powerset of mandatory property set excluding the empty set.
- $\Delta_o : (\mathcal{N}_s \cup \mathcal{E}_s) \rightarrow \mathcal{P}(\mathtt{P}_o)$ is an optional property labeling function that maps all nodes and edges to the powerset, represented as $\mathcal{P}(\mathtt{P}_o)$, of the optional property set.
- $\zeta_s : \mathcal{E}_s \rightarrow \mathtt{C}_s$ is a cardinality labeling function that maps all edges to a set of cardinalities such that $\forall e \in \mathcal{E}_s$, the cardinality function $\zeta_s(e) = (\mathtt{min}, \mathtt{max})$ returns a minimum and maximum value pair such that $\mathtt{min} \leq \mathtt{max}$, $\mathtt{min} \in \mathtt{MIN}$ and $\mathtt{max} \in \mathtt{MAX}$. Given an edge $e \in \mathcal{E}_s$, let $n_i, n_j \in \mathcal{N}_s$ such that $\mathcal{S}_s(e) = n_i$ and $\mathcal{T}_s(e) = n_j$ then the following conditions hold:

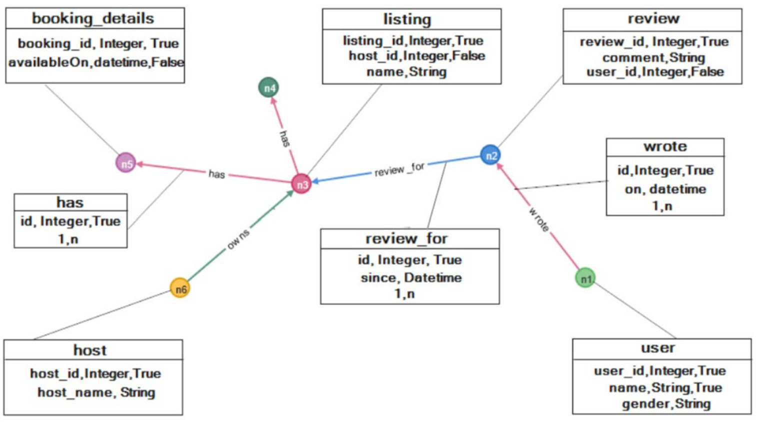

**Fig. 6** Logical graph schema generated for Airbnb case study

- The minimum number of edges belonging to the edge label $\xi_s(e)$ that can exist between nodes of label $\eta_s(n_i)$ and $\eta(n_j)$ is `min`.
- The maximum number of edges belonging to the edge label $\xi_s(e)$ that can exist between nodes of label $\eta_s(n_i)$ and $\eta(n_j)$ is `max`.
- The total number of edges belonging to edge label $\xi_s(e)$ that can exist between nodes of label $\eta_s(n_i)$ and $\eta_s(n_j)$ in a graph database must not be less than `min` and more than `max`.

***Example 7*** By using Definition 7 the logical graph schema generated for Airbnb case study is presented in Fig. 6. The logical graph schema's topology is the same as the conceptual graph schema presented in Fig. 1.

Based in Definition 7 we can observe that a logical graph schema extends the conceptual graph schema by defining the property labeling functions over the nodes and edges of conceptual graph schema. Therefore, the intensional information captured in the conceptual graph schema is maintained in the logical graph schema. Additionally, the logical graph schema consists of extensional information as unique, mandatory, optional properties and edge cardinalities (Angles et al. 2021). Furthermore, the data type associated with each property is also captured in the logical graph schema.

***Example 8*** Figure 6 shows the properties associated with nodes and edges of the logical graph schema. For instance, the node labeled as `host` consists of a mandatory and an optional property. The mandatory property `host_id` is of data type Integer and must be unique. The value associated with the Boolean flag being `TRUE` signifies the uniqueness constraint. The optional property `name` is of data type String and does not contain the uniqueness constraint. As discussed in Definition 7 edges of the logical graph schema contain information about the cardinality. For instance, the edge between node labeled as `host` and `listing` is labeled as `owns` and the cardinality associated in `(1,n)`. This means that a host can own multiple listings and a listing can be associated with a single host. In the cardinality `n` represents a place holder for a natural number that can be calculated while creating the database creation script.

In our approach, the combination of conceptual and logical graph schema modeling stages represent the four steps of database design as suggested by Chen (1976). Information such as entity set, relationship set and organization of data into entities and relationships is covered in conceptual graph schema modeling stage (Angles et al. 2021). In the logical graph schema modeling stage semantic information such as cardinality of edges and properties associated with nodes and edges are defined (Angles et al. 2021).

### 3.3.1 `FLASc` operators for designing logical graph schemas

The three operators, `JOIN`, `DETACH` and `DELETE_NODE` can also be used for designing and manipulating the logical graph schema. As mentioned in Definition 7 a logical graph schema is an extension of conceptual graph schema. Therefore, node and edge labeling functions as well as source and target function are valid in a logical graph schema. The semantics associated with these functions are also same. A logical graph schema consists of additional functions such as mandatory and optional property labeling and edge cardinality functions. The use of `FLASc` operators namely `JOIN`, `DETACH` and `DELETE_NODE` is constrained due the additional labeling functions at the logical graph schema modeling stage. We now discuss the application of `FLASc` operators for logical graph schema modeling:

`JOIN`: The application of `JOIN` on two given logical graph schemas works in the similar manner as for source, target, node and edge labeling functions as presented in Definition 6. The additional mappings are required for property and cardinality labeling functions which are discussed as follows:

**Definition 8** (`JOIN` on Logical Graph Schema) Given two logical graph schemas $\mathcal{G}_{l1} = (\mathcal{N}_{s1}, \mathcal{E}_{s1}, \mathrm{P}_{m1}, \mathrm{P}_{o1}, \mathrm{C}_{s1}, \eta_{s1}, \xi_{s1}, \mathcal{S}_{s1}, \mathcal{T}_{s1}, \Delta_{m1}, \Delta_{o1}, \zeta_{s1})$ and $\mathcal{G}_{l2} = (\mathcal{N}_{s2}, \mathcal{E}_{s2}, \mathrm{P}_{m2}, \mathrm{P}_{o2}, \mathrm{C}_{s2}, \eta_{s2}, \xi_{s2}, \mathcal{S}_{s2}, \mathcal{T}_{s2}, \Delta_{m2}, \Delta_{o2}, \zeta_{s2})$ then $\mathtt{JOIN}(\mathcal{G}_{l1}, \mathcal{G}_{l2}) = \mathcal{G}_{l3} = \left(\mathcal{N}_{s3}, \mathcal{E}_{s3}, \mathrm{P}_{m3}, \mathrm{P}_{o3}, \mathrm{C}_{s3}, \eta_{s3}, \xi_{s3}, \mathcal{S}_{s3}, \mathcal{T}_{s3}, \Delta_{m3}, \Delta_{o3}, \zeta_{s3}\right)$ where:

- $\left(\mathcal{N}_{s3}, \mathcal{E}_{s3}, \eta_{s3}, \xi_{s3}, \mathcal{S}_{s3}, \mathcal{T}_{s3}\right)$ is a conceptual graph schema as discussed in Definition 4. The node and edge labeling functions, source and target functions are defined as in Definition 6.
- $\Delta_{m3} = \Delta_{m1} \cup \Delta_{m2}$ where $\Delta_{m3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup \mathcal{E}_{s1} \cup \mathcal{E}_{s2}) \rightarrow \mathcal{P}^{+}(\mathrm{P}_{m1} \cup \mathrm{P}_{m2})$ such that
  - If $ne_i \in (\mathcal{N}_{s1} \cup \mathcal{E}_{s1})$ then $\Delta_{m3}(ne_i) = \Delta_{m1}(ne_i)$.
  - If $ne_i \in (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})$ then $\Delta_{m3}(ne_i) = \Delta_{m2}(ne_i)$.
  - If $ne_i \in \left((\mathcal{N}_{s1} \cup \mathcal{E}_{s1}) \cap (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})\right)$ then $\Delta_{m3}(ne_i) = \Delta_{m1}(ne_i) = \Delta_{m2}(ne_i)$.
- $\Delta_{o3} = \Delta_{o1} \cup \Delta_{o2}$ where $\Delta_{o3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup \mathcal{E}_{s1} \cup \mathcal{E}_{s2}) \rightarrow \mathcal{P}(\mathrm{P}_{o1} \cup \mathrm{P}_{o2})$ such that

- If $ne_i \in (\mathcal{N}_{s1} \cup \mathcal{E}_{s1})$ then $\Delta_{o3}(ne_i) = \Delta_{o1}(ne_i)$.
- If $ne_i \in (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})$ then $\Delta_{o3}(ne_i) = \Delta_{o2}(ne_i)$.
- If $ne_i \in \big((\mathcal{N}_{s1} \cup \mathcal{E}_{s1}) \cap (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})\big)$ then $\Delta_{o3}(ne_i) = \Delta_{o1}(ne_i) = \Delta_{o2}(ne_i)$.

- $\zeta_{s3} = \zeta_{s1} \cup \zeta_{s2}$ where $\zeta_{s3} : (\mathcal{E}_{s1} \cup \mathcal{E}_{s2}) \rightarrow (\mathtt{C}_{s1} \cup \mathtt{C}_{s2})$ such that
  - If $e \in \mathcal{E}_{s1}$ then $\zeta_{s3}(e) = \zeta_{s1}(e)$.
  - If $e \in \mathcal{E}_{s2}$ then $\zeta_{s3}(e) = \zeta_{s2}(e)$.
  - If $e \in \mathcal{E}_{s1} \cap \mathcal{E}_{s2}$ then $\zeta_{s3}(e) = \zeta_{s2}(e) = \zeta_{s1}(e)$.

The notion of two logical graph schemas being join compatible is same as discussed for conceptual graph schemas as discussed in Sect. 3.2.2. With respect to the properties two logical graph schemas are join compatible if nodes have same mandatory and optional properties that is, $\exists n_1 \in \mathcal{N}_{s1}$ and $\exists n_2 \in \mathcal{N}_{s2}$ such that $\Delta_{m1}(n_1) = \Delta_{m2}(n_2)$ and $\Delta_{o1}(n_1) = \Delta_{o2}(n_2)$. In such a scenario we say that nodes $n_1$ and $n_2$ of two logical graph schemas are join compatible.

`DETACH`: The `DETACH` operator can be utilized by a database designer to delete an existing edge from a logical graph schema. Deleting an existing edge from a logical graph schema requires checking that the two conceptual graphs share some common edge with same labels as discussed in Sect. 3.2.2. Additionally, deleting edges in logical graph schemas also requires that the edge properties and cardinalities must be same. In order to formalize the notion of `DETACH` operator at the logical schema level we further divide the set of mandatory and optional properties into node and edge properties. Let $\mathtt{NP}_m$ and $\mathtt{EP}_m$ be two sets containing mandatory properties specific to nodes and edge respectively such that $\mathtt{P}_m = \mathtt{NP}_m \cup \mathtt{EP}_m$. Similarly, let $\mathtt{NP}_o$ and $\mathtt{EP}_o$ be two sets containing optional properties specific to nodes and edge respectively then $\mathtt{P}_o = \mathtt{NP}_o \cup \mathtt{EP}_o$

**Definition 9** (`DETACH` on logical graph schema) Given two logical graph schema $\mathcal{G}_{l1} = (\mathcal{N}_{s1}, \mathcal{E}_{s1}, (\mathtt{NP}_{m1} \cup \mathtt{EP}_{m1}), (\mathtt{NP}_{o1} \cup \mathtt{EP}_{o1}), \mathtt{C}_{s1}, \eta_{s1}, \xi_{s1}, \mathcal{S}_{s1}, \mathcal{T}_{s1}, \Delta_{m1}, \Delta_{o1}, \zeta_{s1})$ and $\mathcal{G}_{l2} = (\mathcal{N}_{s2}, \mathcal{E}_{s2}, (\mathtt{NP}_{m2} \cup \mathtt{EP}_{m2}), (\mathtt{NP}_{o2} \cup \mathtt{EP}_{o2}), \mathtt{C}_{s2}, \eta_{s2}, \xi_{s2}, \mathcal{S}_{s2}, \mathcal{T}_{s2}, \Delta_{m2}, \Delta_{o2}, \zeta_{s2})$ then $\mathtt{DETACH}(\mathcal{G}_{l1}, \mathcal{G}_{l2}) = \mathcal{G}_{l3} = (\mathcal{N}_{s3}, \mathcal{E}_{s3}, (\mathtt{NP}_{m3} \cup \mathtt{EP}_{m3}, (\mathtt{NP}_{o3} \cup \mathtt{EP}_{o3}), \mathtt{C}_{s3}, \eta_{s3}, \xi_{s3}, \mathcal{S}_{s3}, \mathcal{T}_{s3}, \Delta_{m3}, \Delta_{o3}, \zeta_{s3})$ where:

- $\big(\mathcal{N}_{s3}, \mathcal{E}_{s3}, \eta_{s3}, \xi_{s3}, \mathcal{S}_{s3}, \mathcal{T}_{s3}\big)$ is a conceptual graph schema as discussed in Definition 4. The node and edge labeling functions, source and target functions are defined as in Definition 6.
- $\mathtt{NP}_{m3} \cup \mathtt{EP}_{m3} = \big(\mathtt{NP}_{m1} \cup \mathtt{NP}_{m2} \cup (\mathtt{EP}_{m1} \oplus \mathtt{EP}_{m2})\big)$.
- $\mathtt{NP}_{o3} \cup \mathtt{EP}_{o3} = \big(\mathtt{NP}_{o1} \cup \mathtt{NP}_{o2} \cup (\mathtt{EP}_{o1} \oplus \mathtt{EP}_{o2})\big)$.
- $\Delta_{m3}$ is defined as $\Delta_{m3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2})) \rightarrow \mathcal{P}^{+}(\mathtt{NP}_{m3} \cup \mathtt{EP}_{m3})$ such that
  - If $(\mathcal{E}_{s1} \cap \mathcal{E}_{s2}) \neq \emptyset$ and $ne_i \in (\mathcal{E}_{s1} \cap \mathcal{E}_{s2})$ then $\Delta_{m3}(ne_i) = \emptyset$.
  - Otherwise

    If $ne_i \in (\mathcal{N}_{s1} \cup \mathcal{E}_{s1})$ then $\Delta_{m3}(ne_i) = \Delta_{m1}(ne_i)$.
    If $ne_i \in (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})$ then $\Delta_{m3}(ne_i) = \Delta_{m2}(ne_i)$.
    If $ne_i \in \big((\mathcal{N}_{s1} \cup \mathcal{E}_{s1}) \cap (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})\big)$ then $\Delta_{m3}(ne_i) = \Delta_{m2}(ne_i) = \Delta_{m1}(ne_i)$.

- $\Delta_{o3}$ is defined as $\Delta_{o3} : (\mathcal{N}_{s1} \cup \mathcal{N}_{s2} \cup (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2})) \rightarrow \mathcal{P}(\mathtt{NP}_{o3} \cup \mathtt{EP}_{o3})$ such that

- If $(\mathcal{E}_{s1} \cap \mathcal{E}_{s2}) \neq \emptyset$ and $ne_i \in (\mathcal{E}_{s1} \cap \mathcal{E}_{s2})$ then $\Delta_{o3}(ne_i) = \emptyset$.
- Otherwise

  If $ne_i \in (\mathcal{N}_{s1} \cup \mathcal{E}_{s1})$ then $\Delta_{o3}(ne_i) = \Delta_{o1}(ne_i)$.
  If $ne_i \in (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})$ then $\Delta_{o3}(ne_i) = \Delta_{o2}(ne_i)$.
  If $ne_i \in \big((\mathcal{N}_{s1} \cup \mathcal{E}_{s1}) \cap (\mathcal{N}_{s2} \cup \mathcal{E}_{s2})\big)$ then $\Delta_{o3}(ne_i) = \Delta_{o2}(ne_i) = \Delta_{o1}(ne_i)$.

- $\zeta_{s3}$ is defined as $\zeta_{s3} : (\mathcal{E}_{s1} \oplus \mathcal{E}_{s2}) \rightarrow (\mathtt{C}_{s1} \oplus \mathtt{C}_{s2})$ such that
  - If $e \in (\mathcal{E}_{s1} \cap \mathcal{E}_{s2})$ and $(\mathcal{E}_{s1} \cap \mathcal{E}_{s2}) \neq \emptyset$ then $\zeta_{s3}(e) = \emptyset$.
  - Otherwise if $e \in \mathcal{E}_{s1}$ then $\zeta_{s3}(e) = \zeta_{s1}(e)$. If $e \in \mathcal{E}_{s2}$ then $\zeta_{s3}(e) = \zeta_{s2}(e)$.

In order to delete existing edges by using the `DETACH` operator there must exist some edges that are common between two logical graph schemas that is $(\mathcal{E}_{s1} \cap \mathcal{E}_{s2}) \neq \emptyset$. This means that labels for both edges must be the same. Additionally, the properties and cardinalities associated with the common edges must be same as well that is $\exists e_1 \in \mathcal{E}_{s1}$ and $\exists e_2 \in \mathcal{E}_{s2}$ such that $\Delta_{m1}(e_1) = \Delta_{m2}(e_2), \Delta_{o1}(e_1) = \Delta_{o2}(e_2)$ and $\zeta_{s1}(e_1) = \zeta_{s2}(e_2)$.

`DELETE_NODE`: The `DELETE_NODE` operator can be utilized by a database designer to delete disconnected nodes from a logical graph schema. As discussed in Sect. 3.2.2 in order to delete an existing disconnected node the two logical graph schemas must contain common nodes. As mentioned in Definition 6 the node labeling must be same. Additionally the mandatory and optional properties must be the same as well.

**Definition 10** (`DELETE_NODE` on logical graph schema) Given two logical graph schemas $\mathcal{G}_{l1} = (\mathcal{N}_{s1}, \mathcal{E}_{s1}, (\mathtt{NP}_{m1} \cup \mathtt{EP}_{m1}), (\mathtt{NP}_{o1} \cup \mathtt{EP}_{o1}), \mathtt{C}_{s1}, \eta_{s1}, \xi_{s1}, \mathcal{S}_{s1}, \mathcal{T}_{s1}, \Delta_{m1}, \Delta_{o1}, \zeta_{s1})$ and $\mathcal{G}_{l2} = (\mathcal{N}_{s2}, \mathcal{E}_{s2}, \mathtt{NP}_{m2}, \mathtt{NP}_{o2}, \eta_{s2}, \mathcal{S}_{s2}, \mathcal{T}_{s2}, \Delta_{m2}, \Delta_{o2})$ is a node labeled property graph such that $\mathcal{E}_{s2} = \emptyset$ and subsequently $\mathcal{S}_{s2} = \emptyset$ and $\mathcal{T}_{s2} = \emptyset$. Then $\mathtt{DELETE\_NODE}(\mathcal{G}_{l1}, \mathcal{G}_{l2}) = \mathcal{G}_{l3} = \big(\mathcal{N}_{s3}, \mathcal{E}_{s3}, (\mathtt{NP}_{m3} \cup \mathtt{EP}_{m3}), (\mathtt{NP}_{o3} \cup \mathtt{EP}_{o3}), \mathtt{C}_{s1}, \eta_{s3}, \xi_{s3}, \mathcal{S}_{s3}, \mathcal{T}_{s3}, \Delta_{m3}, \Delta_{o3}, \zeta_{s3}\big)$ where:

- $\big(\mathcal{N}_{s3}, \mathcal{E}_{s3}, \eta_{s3}, \xi_{s3}, \mathcal{S}_{s3}, \mathcal{T}_{s3}\big)$ is a conceptual graph schema as discussed in Definition 4. The node and edge labeling functions, source and target functions are defined as in Definition 6.
- $\mathtt{NP}_{m3} \cup \mathtt{EP}_{m3} = \big((\mathtt{NP}_{m1} \oplus \mathtt{NP}_{m2}) \cup \mathtt{EP}_{m1}\big)$.
- $\mathtt{NP}_{o3} \cup \mathtt{EP}_{o3} = \big((\mathtt{NP}_{o1} \oplus \mathtt{NP}_{o2}) \cup \mathtt{EP}_{o1}\big)$.
- $\Delta_{m3}$ is defined as $\Delta_{m3} : \big((\mathcal{N}_{s1} \oplus \mathcal{N}_{s2}) \cup \mathcal{E}_{s1}\big) \rightarrow \mathcal{P}^{+}(\mathtt{NP}_{m3} \cup \mathtt{EP}_{m3})$ such that
  - If $(\mathcal{N}_{s1} \cap \mathcal{N}_{s2}) \neq \emptyset$ and $ne_i \in \mathcal{N}_{s1} \cap \mathcal{N}_{s2}$ then $\Delta_{m3}(ne_i) = \emptyset$.
  - Otherwise if $ne_i \in (\mathcal{N}_{s1} \cup \mathcal{E}_{s1})$ then $\Delta_{m3}(ne_i) = \Delta_{m1}(ne_i)$. If $ne_i \in \mathcal{N}_{s2}$ then $\Delta_{m3}(ne_i) = \Delta_{m2}(ne_i)$.
- $\Delta_{o3}$ is defined as $\Delta_{o3} : \big((\mathcal{N}_{s1} \oplus \mathcal{N}_{s2}) \cup \mathcal{E}_{s1}\big) \rightarrow \mathcal{P}(\mathtt{NP}_{o3} \cup \mathtt{EP}_{o3})$ such that
  - If $(\mathcal{N}_{s1} \cap \mathcal{N}_{s2}) \neq \emptyset$ and $ne_i \in \mathcal{N}_{s1} \cap \mathcal{N}_{s2}$ then $\Delta_{o3}(ne_i) = \emptyset$.
  - Otherwise if $ne_i \in (\mathcal{N}_{s1} \cup \mathcal{E}_{s1})$ then $\Delta_{o3}(ne_i) = \Delta_{o1}(ne_i)$. If $ne_i \in \mathcal{N}_{s2}$ then $\Delta_{o3}(ne_i) = \Delta_{o2}(ne_i)$.
- $\zeta_{s3} = \zeta_{s1}$ such that $\zeta_{s3} : \mathcal{E}_{s1} \rightarrow \mathtt{C}_{s1}$.

**Table 1** Axiomatic specifications of operators in FLASc

| Axioms | JOIN | DETACH | DELETE_NODE |
|---|---|---|---|
| Associativity | $\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}$ $[(\mathcal{G}_1 \sqcup \mathcal{G}_2) \sqcup \mathcal{G}_3 = \mathcal{G}_1 \sqcup (\mathcal{G}_2 \sqcup \mathcal{G}_3)]$ | $\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}$ $[(\mathcal{G}_1 \Diamond \mathcal{G}_2) \Diamond \mathcal{G}_3 = \mathcal{G}_1 \Diamond (\mathcal{G}_2 \Diamond \mathcal{G}_3)]$ | $\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G}$ $[(\mathcal{G}_1 \nabla \mathcal{G}_2) \nabla \mathcal{G}_3 = \mathcal{G}_1 \nabla (\mathcal{G}_2 \nabla \mathcal{G}_3)]$ |
| Commutativity | $\forall \mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}$ $[\mathcal{G}_1 \sqcup \mathcal{G}_2 = \mathcal{G}_2 \sqcup \mathcal{G}_1]$ | $\forall \mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}$ $[\mathcal{G}_1 \Diamond \mathcal{G}_2 = \mathcal{G}_2 \Diamond \mathcal{G}_1]$ | $\forall \mathcal{G}_1, \mathcal{G}_2 \in \mathcal{G}$ $[\mathcal{G}_1 \nabla \mathcal{G}_2 = \mathcal{G}_2 \nabla \mathcal{G}_1]$ |
| Identity | $\forall \mathcal{G}_1 \in \mathcal{G} [\mathcal{G}_1 \sqcup I_{\mathcal{G}} = \mathcal{G}_1]$ | $\forall \mathcal{G}_1 \in \mathcal{G} [\mathcal{G}_1 \Diamond I_{\mathcal{G}} = \mathcal{G}_1]$ | $\forall \mathcal{G}_1 \in \mathcal{G} [\mathcal{G}_1 \nabla I_{\mathcal{G}} = \mathcal{G}_1]$ |
| Idempotent | $\forall \mathcal{G}_1 \in \mathcal{G} [\mathcal{G}_1 \sqcup \mathcal{G}_1 = \mathcal{G}_1]$ | $\forall \mathcal{G}_1 \in \mathcal{G} [\mathcal{G}_1 \Diamond \mathcal{G}_1 = \mathcal{G}_1]$ | $\forall \mathcal{G}_1 \in \mathcal{G} [\mathcal{G}_1 \nabla \mathcal{G}_1 = \mathcal{G}_1]$ |

$\sqcup$ = JOIN operator

$\Diamond$ = DETACH operator

$\nabla$ = DELETE_NODE operator

**Table 2** Distributive axiom of FLASc operators

| FLASc operators | Axiomatic Specification |
|---|---|
| JOIN and DETACH | $\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G} [\mathcal{G}_1 \sqcup (\mathcal{G}_2 \Diamond \mathcal{G}_3) = (\mathcal{G}_1 \sqcup \mathcal{G}_2) \Diamond (\mathcal{G}_1 \sqcup \mathcal{G}_3)]$ |
| JOIN and DELETE_NODE | $\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G} [\mathcal{G}_1 \sqcup (\mathcal{G}_2 \nabla \mathcal{G}_3) = (\mathcal{G}_1 \sqcup \mathcal{G}_2) \nabla (\mathcal{G}_1 \sqcup \mathcal{G}_3)]$ |
| DETACH and DELETE_NODE | $\forall \mathcal{G}_1, \mathcal{G}_2, \mathcal{G}_3 \in \mathcal{G} [\mathcal{G}_1 \Diamond (\mathcal{G}_2 \nabla \mathcal{G}_3) = (\mathcal{G}_1 \Diamond \mathcal{G}_2) \nabla (\mathcal{G}_1 \Diamond \mathcal{G}_3)]$ |

$\sqcup$ = JOIN operator

$\Diamond$ = DETACH operator

$\nabla$ = DELETE_NODE operator

In order to delete existing nodes by using the DELETE_NODE operator there must exist some nodes that are common between two logical graph schemas that is ($\mathcal{N}_{s1} \cap \mathcal{N}_{s2} \neq \emptyset$). This means that labels for both nodes must be the same. Additionally, the mandatory and optional properties associated with the common nodes must be same as well that is $\exists n_1 \in \mathcal{N}_{s1}$ and $\exists n_2 \in \mathcal{N}_{s2}$ such that $\Delta_{m1}(n_1) = \Delta_{m2}(n_2)$ and $\Delta_{o1}(n_1) = \Delta_{o2}(n_2)$.

### 3.3.2 Axiomatic specifications of FLASc operators

The axiomatic specifications of any algebra enable us to check its completeness (Tucker and Stephenson 2003). In order to show the axiomatic specification we use infix notation for the operators in FLASc. As such we use the ($\sqcup$) notation for the JOIN operator, ($\Diamond$) notation for the DETACH operator and ($\nabla$) notation for the DELETE_NODE operator.

The axiomatic specification of FLASc operators is presented in Table 1. For defining the identity axiom, we define an identity graph $I_{\mathcal{G}} = (\emptyset, \emptyset)$ which means that the identity graph does not contain any nodes and edges. We can observe that JOIN, DETACH and DELETE_NODE operators follow associativity, commutativity, idempotent and identity axioms.

The distributive axioms for the JOIN, DETACH and DELETE_NODE operators is presented in Table 2. The axiomatic specification of FLASc operators enable us to use `FLASc` for generating new graph schemas from existing logical and conceptual graph schemas.

The integrity constraints that can be enforced by a logical graph schema presented in Definition 7 include graph entity integrity constraints such as property uniqueness, label uniqueness, property data type and mandatory property constraints. The enforcement of these constraints and semantics constraints such as edge pattern, graph pattern, and path pattern constraints can be done at the physical modeling stage by using database-specific query languages. Following the graph schema to generate database creation scripts at the physical modeling stage ensures data consistency.

### 3.3.3 Schema instance consistency

The schema instance consistency is used to ensure that the labeled property graph database constructed at the physical modeling stage adheres to the logical graph schema generated by using `FLASc`. A labeled property graph database uses a graph structure for storing and managing data, allowing the modeling of real world entities as nodes and edges (Angles et al. 2018; Sharma and Sinha 2019). Nodes are used to store data and relationships or interactions between nodes are stored as edges (Angles et al. 2017; Sharma et al. 2019). Nodes and edges in a graph database can have properties associated with them. Let $\mathtt{K}_d$ be a set of infinite keys (e.g., id, name, age, etc.), $\mathtt{V}_d$ be a set of infinite values (e.g., 345, `James`, 33, etc.) and $\mathtt{T}_d$ be a set of finite data types (e.g., String, Integer etc.) we define a function $\Upsilon : \mathtt{V}_d \rightarrow \mathtt{T}_d$ that maps values in set $\mathtt{V}_d$ to their respective data types in the set $\mathtt{T}_d$. The set of properties associated with the nodes and edges of a graph database are defined as $\mathtt{P}_d \subseteq (\mathtt{K}_d \times \mathtt{V}_d)$ such that each $p_d \in \mathtt{P}_d$ is a key-value pair where each value has a data type. To accommodate the existence of mandatory and optional properties the set of properties can be further written as $\mathtt{P}_d = \mathtt{P}_{dm} \cup \mathtt{P}_{do}$. Formally a labeled property graph database is defined as follows:

**Definition 11** (`Labeled Property Graph Database`) A labeled property graph database $\mathcal{G}_d$ is a tuple $(\mathcal{N}_d, \mathcal{E}_d, \mathtt{P}_{dm}, \mathtt{P}_{do}, \eta_d, \xi_d, \mathcal{S}_d, \mathcal{T}_d, \Delta_{dm}, \Delta_{do})$ where,

- $\mathcal{N}_d$ is a finite set of nodes and $\mathcal{E}_d$ is a finite set of edges of the graph database
- $(\mathcal{N}_d, \mathcal{E}_d, \mathcal{S}_d, \mathcal{T}_d)$ a directed multigraph as discussed in Definition 1.
- $\mathtt{P}_{dm}$ and $\mathtt{P}_{do}$ are mandatory and optional property sets associated with the graph database.
- $\eta_d : \mathcal{N}_d \rightarrow L_{\mathcal{N}}$ is a node labeling function which maps all nodes to labels in the set of node labels $L_{\mathcal{N}}$.
- $\xi_d : \mathcal{E}_d \rightarrow L_{\mathcal{E}}$ is an edge labeling function which maps all edges to labels in the set of edge labels $L_{\mathcal{E}}$.
- $\Delta_{dm} : (\mathcal{N}_d \cup \mathcal{E}_d) \rightarrow \mathcal{P}^{+}(\mathtt{P}_{dm})$ is a property labeling function which maps all nodes and/or edges to all subsets (excluding the empty set) of the mandatory property set $\mathtt{P}_{dm}$.

- $\Delta_{do} : (\mathcal{N}_d \cup \mathcal{E}_d) \rightarrow \mathcal{P}(\mathtt{P}_{do})$ is a property labeling function which maps all nodes and/or edges to all subsets (including the empty set) of the optional property set $\mathtt{P}_{do}$.

The notion of schema instance consistency implies that a labeled property graph database adheres to the structural restrictions established by a labeled property graph schema (2018). Such a notion can be formally defined as follows:

**Definition 12** (`Schema Instance Consistency`) Given a labeled property graph database $\mathcal{G}_d = (\mathcal{N}_d, \mathcal{E}_d, \mathtt{P}_{dm}, \mathtt{P}_{do}, \eta_d, \xi_d, \mathcal{S}_d, \mathcal{T}_d, \Delta_{dm}, \Delta_{do})$ as defined in Definition 11 and a labeled property graph schema $\mathcal{G}_l = (\mathcal{N}_s, \mathcal{E}_s, \mathtt{P}_{sm}, \mathtt{P}_{so}, \mathtt{C}_s, \eta_s, \xi_s, \mathcal{S}_s, \mathcal{T}_s, \Delta_{sm}, \Delta_{so}, \zeta_s)$ as defined in Definition 7. We say that $\mathcal{G}_d$ is consistent with $\mathcal{G}_l$ when:

- For each node $n \in \mathcal{N}_d$, there must exist a corresponding node in graph schema where $n' \in \mathcal{N}_s$ such that $\eta_d(n) = \eta_s(n')$.
- For each edge $e_i \in \mathcal{G}_d$ there must exist a corresponding edge in graph schema that is $e'_i \in \mathcal{G}_l$ such that $\eta_d(\mathcal{S}_d(e_i)) = \eta_s(\mathcal{S}_s(e'_i))$, $\eta_d(\mathcal{T}_d(e_i)) = \eta_s(\mathcal{T}_s(e'_i))$ and $\xi_d(e_i) = \xi_s(e'_i)$.
- For each $n_i \in \mathcal{N}_d$ (or $e_i \in \mathcal{E}_d$), there exists $n'_i \in \mathcal{N}_s$ (or $e'_i \in \mathcal{E}_s$) such that
  - If $\Delta_{dm}(n_i) = k_{dm} \times v_d$ where $k_{dm} \in \mathtt{K}_d$ and $v_d \in \mathtt{V}_d$.
  - If $\Delta_{sm}(n'_i) = k_{sm} \times t_s$ where $k_{sm} \in \mathtt{K}_s$ and $t_s \in \mathtt{T}_s$.
  - Then, $k_{sm} \times t_s = k_{dm} \times \Upsilon(v_d)$ that is, the key and data type of value stored in node (or edge) of graph database is same as the key and data type of node (or edge) in the graph schema.
- For each $n_i \in \mathcal{N}_d$ (or $e_i \in \mathcal{E}_d$), there exists $n'_i \in \mathcal{N}_s$ (or $e'_i \in \mathcal{E}_s$) such that
  - If $\Delta_{do}(n_i) = k_{do} \times v_d$ where $k_{do} \in \mathtt{K}_d$ and $v_d \in \mathtt{V}_d$.
  - If $\Delta_{so}(n'_i) = k_{so} \times t_s$ where $k_{so} \in \mathtt{K}_s$ and $t_s \in \mathtt{T}_s$.
  - Then, $k_{so} \times t_s = k_{do} \times \Upsilon(v_d)$ that is, the key and data type of value stored in node (or edge) of graph database is same as the key and data type of node (or edge) in the graph schema.
- The total number of edges of a certain label generated in the labeled property graph database must be between the minimum and maximum cardinality values associated with edges of same label in the graph schema.

Cardinality can be enforced programatically at the physical modeling stage by using the logical graph schema generated by `FLASc`. Similarly, the adherence to node and edge labeling, property (optional and mandatory) labeling can be enforced at the physical modeling stage. The logical graph schema is independent of the underlying implementations. Moreover, the graph schema can be used in both integrated and layered physical modeling approaches. To support our claim in the following two sections, we experimentally demonstrate the use of graph schema to transform and load data-sets by using both approaches for physical modeling for graph databases.

However, while demonstrating the integrated approach we do not make any changes to the source code of graph database system and consider this as future work.

## 4 Using `FLASc` to enforce integrity constraints

In this section, we demonstrate the use of graph schema generated by `FLASc` for enforcing integrity constraints, which are essential for ensuring data consistency in graph databases. We illustrate the manual integration of conceptual, logical and physical modeling stages. We design the database creation scripts using the logical graph schema generated by `FLASc` for Airbnb data-set as shown in Fig. 6. We do not make any changes to the source code of Neo4j; however, the formulation of database creation scripts in Cypher is driven by the logical graph schema. We then execute these scripts directly over the Neo4j graph database.

As discussed in Sharma and Sinha (2019) Airbnb data-set consists of three CSV files containing information related to listings, review and calendar data. The listing file contains information, such as hosts that own the listings, amenities provided in the listings, location of the listing etc. The reviews file contains information related to the users who have stayed in the listings and provided feedback in reviews. The calendar file contains information related to booking details such as pricing and occupancy. These files contain multiple lines (rows) of data, where each row contains a comma-separated list of values. For instance, a CSV file containing information related to listings from Airbnb's data is shown in Table 3.

### 4.1 Manual generation of database creation scripts

The logical graph schema generated by `FLASc` for Airbnb data-set contains intensional and extensional information that assists a database designer for enforcing integrity constraints in the database scripts.

#### 4.1.1 Enforcement of graph entity integrity constraints

Graph entity integrity constraints are used to enforce restrictions on properties associated with nodes and edges in a graph database. The extensional information captured in the logical graph schema as discussed in Definition 7 is used to enforce such constraints. We discuss the enforcement of graph entity integrity constraints for transforming and loading Airbnb data-set into Neo4j graph database by using Cypher query language.

*Node property uniqueness constraint* The sample listing file as shown in Table 3 has `Listing ID` associated with each listing. Furthermore, in the logical graph schema shown in Fig. 6 `listing_id` field the uniqueness flag is set to be `True` which means that the `listing_id` must be unique. Therefore, before creating the listing nodes in the Neo4j graph database, the uniqueness constraint must be established to reduce data corruption chances. This is achieved by running Query 1

**Table 3** Sample data from listing.csv in the Airbnb data-set

| Host name | Listing ID | Listing name | Room type | Street | Host ID |
|---|---|---|---|---|---|
| Manju | 9835 | Beautiful room & House | Private room | Bulleen, VIC, Australia | 33057 |
| Lindsay | 10803 | Room in Cool Deco Apartment in Brunswick East | Private room | Brunswick East, VIC, Australia | 38901 |
| Eleni | 15246 | Large private room-close to city | Private room | Thornbury, VIC, Australia | 59786 |
| Eleni | 68482 | Charming house inner Melbourne | Entire home/apt | Thornbury, VIC, Australia | 59786 |

in Cypher. The mechanism to enforce uniqueness constraint is predefined in Neo4j graph database.

**Query 1** Cypher query to enforce node property uniqueness constraint

```
CREATE CONSTRAINT unique_listing_id IF NOT EXISTS ON (list:listing)
ASSERT list.listing_id IS UNIQUE
```

The uniqueness constraint specified in Query 1 ensures that multiple nodes with same `listing_id` are not created in the Neo4j graph database. The use of `IF NOT EXISTS` clause is used to ensure that the constraint is enforced at most once. The next constraints to be enforced are the mandatory node and edge property constraints.

*Mandatory node property constraint* The sample listing file also contains information about the `host_id` and in the logical graph schema as shown in Fig. 6, the `host_id` is a mandatory field. Therefore, additional constraints must be enforced on the listing nodes. This can be achieved by running the following query in Cypher.

**Query 2** Cypher query to enforce mandatory node property constraint

```
CREATE CONSTRAINT listing_host_id IF NOT EXISTS ON (list:listing)
ASSERT EXISTS list.host_id
```

The node property existence constraint specified in Query 2 ensures that listing nodes must always have a value assigned to the property `host_id` the `ASSERT EXISTS` clause is used to enforce such a condition.

*Mandatory edge property constraint* The mandatory property constraints can also be specified on the edges that have to be created in the graph database. The logical graph schema as discussed in Definition 7 helps in enforcing this constraint in two ways; first, it provides details about the edge labels. Second, it also provides details about mandatory, unique and optional properties associated with the edges. For example, as shown in Fig. 6 the edge labeled as `owns` has a mandatory property `since` which can be enforced by running the following Cypher query.

**Query 3** Cypher query to enforce mandatory edge property constraint

```
1. CREATE CONSTRAINT owns_edge_id IF NOT EXISTS ON ()-[owns:OWNS]->()
2. ASSERT EXISTS owns.id
```

The mandatory edge property constraint shown in Query 3 is used to ensure that their is always a value assigned to `id` of every edge labeled as `OWNS` in the graph database.

*Node key constraint* This constraint can be applied over a set of node properties. This constraint combines the functionality provided by uniqueness and mandatory property constraints. For example, the node labeled as `host` has two mandatory and unique properties `user_id` and `name`. This constraint can be enforced in the Neo4j graph database by using Query 4.

**Query 4** Cypher query to enforce node key property constraint

```
CREATE CONSTRAINT ON (u:user)
ASSERT u.user_id, u.name IS NODE KEY
```

As shown in Query 4 the use of `IS NODE KEY` keywords along with the `ASSERT` clause is used to enforce that the properties `user_id` and `name` are unique and must have a value associated with them in the graph database.

*Property data type constraint* Logical graph schema is used to enforce property data type constraint over the node and edge properties. As discussed in Definition 7 a logical graph schema contains properties that have a data type associated with them. Therefore, database creation scripts are designed by utilizing this information. For instance, in the logical graph schema shown in Fig. 6 `listing_id` and `host_id` are of Integer data type the Cypher query to enforce this constraint is presented as Query 5.

**Query 5** Cypher query to enforce property data type and edge pattern constraint

```
1. LOAD CSV WITH HEADERS FROM "http://data.insideairbnb.com/australia/
vic/melbourne/2021-01-10/visualisations/listings.csv" AS row
2. WITH DISTINCT row.id AS listing_id,
3. row.host_id AS host_id
4. row.name AS listing_name,
5. row.host_name AS host_name,
6. row AS row
7. MERGE(list:listing{listing_id:toInteger(listing_id),
host_id:toInteger(host_id),name: CASE WHEN listing_name IS NOT NULL
8. THEN listing_name
9. ELSE 'System' END})
10. MERGE(host:host{host_id:toInteger(host_id), name: CASE WHEN host_name
IS NOT NULL
11. THEN host_name
12. ELSE 'System' END})
13. WITH DISTINCT list AS l, host AS h, row AS row
14. WHERE l.host_id = h.host_id
15. AND l.listing_id = toInteger(row.id)
16. AND h.host_id = toInteger(row.host_id)
17. CREATE (h)-[:owns{date:datetime(row.last_review)}]->(l)
```

The property data type constraint is enforced by using the inbuilt `toInteger()` function in Cypher, as shown in lines 7 and 10 of Query 5. The use of this function is due to the specification in logical graph schema that the data type associated with `listing_id` and `host_id` must of Integer data type. In Query 5 the use of

Cypher's `MERGE` clauses in lines 7 and 10, represents the creation of two nodes that is a listing node and a host node. This also illustrates the combination of conceptual and logical modeling stages where a basic conceptual graph schema containing two disconnected nodes as discussed in Definition 5 is further labeled with node properties, representing the use of node labeling function ($\eta$) as discussed in Definition 7. Additionally, Cypher also supports the use of `CASE` statements as illustrated in lines 7–12 of Query 5. The `CASE` statements are used to ensure that if there exists some missing value in the csv files, then those values are loaded as a user defined values such as 'System' in our case.

Other graph entity integrity constraints such as *node and edge label uniqueness* are by default maintained by the logical graph schema generated using `FLASc`. By Definition 7 a node/edge can only have one label associated with it. On the other hand, Neo4j allows a node to be associated with more than one label (Bonifati et al. 2018; Neo4j 2021). `FLASc` does not support this for the sake of simplicity. Such features are not present in all graph database systems and tend to make the definitions of graph schema and graph databases complex (Angles et al. 2020, 2019). Constraints such as *edge property uniqueness* can be specified in `FLASc` however, such constraints cannot be enforced in Neo4j.

### 4.1.2 Enforcement of semantic integrity constraints

Semantic integrity constraints are used to enforce a topological restriction on the graph database. The intensional information captured in the graph schema during the conceptual modeling stage becomes useful to enforce semantic integrity constraints.

*Edge pattern constraint* To enforce edge pattern constraint the topological information stored in the logical graph schema is used while creating the database creation scripts. For instance, Query 5 is also used to create edges between nodes of label `host` and `listing`. Each edge created by using Query 5 is labeled as `owns` and represents a valid edge in the logical graph schema shown in Fig. 6. According to the Neo4j Cypher manual [4] `MERGE` clause serves as a combination of `MATCH` and `CREATE` clauses. Therefore, in Query 5 the `MERGE` clause in lines 7 and 10 is used to first create and then match the `host` and `listing` nodes. The `WITH` clause as presented in line 13 of Query 5 allows query parts to be chained together,[5] therefore, the `host` and `listing` nodes created in lines 7-12 are passed by using the `WITH` clause to facilitate the creation of edges between `host` and `listing` node types, that is in lines 13-17 of Query 5. The `DISTINCT` clause along with the `WITH` clause is used to ensure the removal of duplicate nodes in Query 5. The `WHERE` clause in line 14-16 at is used to define some constraints to filter results based on the values obtained from the csv files. The `CREATE` clause at line 17 in Query 5 represents the creation of a graph containing two nodes and an edge connecting them as discussed in Definition 5. The edge of the graph is further labeled with edge properties further representing the use of edge labeling function ($\xi$) as discussed in Definition 7.

[4] https://neo4j.com/docs/cypher-manual/current/clauses/merge/.

[5] https://neo4j.com/docs/cypher-manual/current/clauses/with/.

*Graph pattern constraint* Enforcing graph pattern constraints require knowledge about the topology of the data-set, which is captured by logical graph schema. These constraints check for the existence of certain graph structure in the database before any new node or edge can be created. Graph pattern constraint in Cypher is presented as Query 6 which ensures that `listing` nodes that have been `reviewed` by a `user` are attached to `booking_detail` nodes by edges that are labeled as `has`.

**Query 6** Cypher query to enforce graph pattern constraint

```
1. :auto USING PERIODIC COMMIT
2. LOAD CSV WITH HEADERS FROM "http://data.insideairbnb.com/australia/
vic/melbourne/2021-01-10/visualisations/calendar.csv" AS row
3. MATCH (u:user)-[:wrote]->(r:review), (r)-[:review_for]->(l:listing)
4. WHERE l.listing_id = toInteger(row.listing_id)
5. WITH DISTINCT row AS row, l AS l
6. CREATE (l)-[:has{id:toInteger(row.id)}]->(b:booking_detail)
```

In Query 6 the `MATCH` clause in line 3 is used to check if graph pattern exists or not. This graph pattern (Angles et al. 2017) is built by using the intensional information in the logical graph schema presented in Fig. 6 that assists in formulating valid graph patterns for enforcing such constraints. The `MATCH` clause in this query connects two graph patterns which are join compatible (Sharma et al. 2021). The `CREATE` clause in line 6 is used to combine the graph obtained from the `MATCH` clause with a logical graph schema specified in the `CREATE` clause. This represents the use of `JOIN` operator. The two logical graph schemas are join compatible since they share the node `l` labeled as `listing`. Query 6 also illustrates the use of `:auto USING PERIODIC COMMIT` clause in line 1, which is used to handle the large amount of data being processed.

*Path pattern constraint* These constraints check for the existence of certain paths in a graph database before a new node or edge can be created. Query languages for graph databases use the formalism of conjunctive two-way regular path queries `(C2RPQs)` and nested regular expressions `(NREs)` to express and then search for path patterns (Florescu et al. 1998; Wood 2012; Angles et al. 2014; Bagan et al. 2015; Barceló et al. 2011, 2012, 2016; Reutter 2013; Barceló et al. 2012). Furthermore, other expressive formalism such as conjunctive queries and union of conjunctive queries extended with Tarski's relation algebra `(CQT/UCQT)` proposed in Sharma et al. (2021) can also be used to enforce path constraints. In these formalisms regular expressions defined over the edge labels of the graph database are used to describe path patterns (Angles et al. 2017). The intensional information captured in logical graph schema assists in creating valid path patterns. Query 7 illustrates the enforcement of path pattern constraint in Cypher. Very similar to Query 6 the use of `CREATE` clause in the query represents the use of `JOIN` operator to combine the graph obtained from the `MATCH` clause at line 3 with the logical graph schema specified in the `CREATE` clause in line 6.

**Query 7** Cypher query to enforce path pattern constraint

```
1. :auto USING PERIODIC COMMIT
2. LOAD CSV WITH HEADERS FROM "http://data.insideairbnb.com/australia/
vic/melbourne /2021-01-10/visualisations/calendar.csv" AS row
3. MATCH (u:user)-[:wrote]->()-[review_for]->(l:listing)
4. WHERE l.listing_id = toInteger(row.listing_id)
5. WITH row AS row, l AS l
6. CREATE (l)-[:HAS{id:toInteger(row.id)}]->
(a:amenity{amenity_type:row.amenity_type})
```

In Query 7 the path pattern constraint is specified in the `MATCH` clause, which represents the regular expression `(wrote.review_for)` formed by applying concatenation operator over the edge labels `wrote`, `review_for` and `has`. Other regular expressions operators such as union and Kleene star can also be used to form more expressions. However, Cypher only provides limited support for regular expressions as the Kleene star operator's use over the concatenation of two more edge labels is not allowed in Cypher (Angles et al. 2017; Sharma et al. 2021). Further modifications can be done to the query language by using formalism such as Tarski's algebra instead of regular expressions for increasing their expressiveness (Sharma et al. 2021).

Other Constraints such as schema instance consistency are ensured since the generation of database creation scripts is driven by the logical graph schema. Constraints such as functional dependencies are not easy to enforce in graph databases (Angles and Gutierrez 2008); however, in order to enforce functional dependencies while modeling graph databases, a designer can follow the approach proposed in Park et al. (2014). This approach states that every non-key property must only provide information about the associated nodes and edges. Constraints such as edge identify uniqueness and cardinality constraints cannot be directly enforced in Neo4j. However, enforcing such constraints can be done by writing a wrapper in programming languages such as Java, Python that can be used to ensure that edge ids must be unique.

The logical graph schema generated by `FLASc` enables us to enforce several practical integrity constraints. `FLASc` assists in the generation of robust conceptual and logical graph schemas. `FLASc` can be integrated with the existing Extract-Transform-load process for ensuring data consistency when data from heterogeneous sources is being loaded into a graph database such as Neo4j. The manual approach presented in this section has limitations. Firstly this approach requires a database designer to possess knowledge of graph database query language such as Cypher. Secondly, creating the database creation scripts manually can be cumbersome and error-prone, making the process less maintainable, scalable and manageable. Finally, Cypher does not support loading data from heterogeneous sources into the Neo4j graph database. Therefore, to mitigate such limitations in the next section, we present our layered approach.

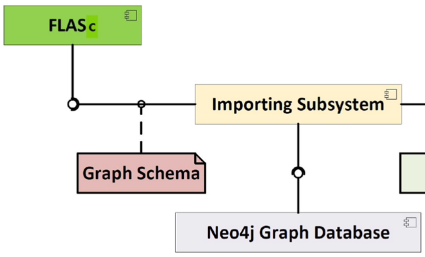

**Fig. 7** Physical view of Schema driven layered approach

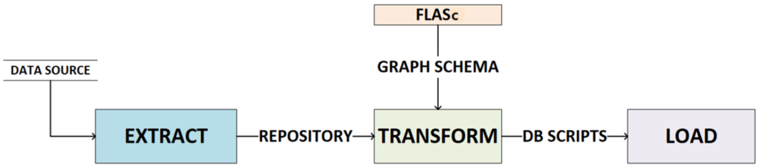

**Fig. 8** Process view of Schema driven layered approach

## 5 A layered approach for data transformation and loading using `FLASc`

Graph databases are schema-less or schema optional; therefore, maintaining data consistency and integrity is not easy. A graph database can be easily altered unless the database's underlying source code is not amended to support the enforcement of all integrity constraints. Hence in this section, we propose a *layered approach* that incorporates the development of an additional wrapper to ensure data consistency. While following the layered approach, we use the APIs provided by Neo4j to access the graph database. We illustrate how `FLASc` can be used to assist the transformation and loading of data from heterogeneous sources into graph databases hence addresses RQ3 and RQ3.1.

### 5.1 Schema driven layered approach

*Overview* The overall physical view of our layered approach is presented in Fig. 7, that consists of three main components *(i)* `FLASc` *which serves as a graph schema generator*, *(ii) an importing subsystem* and *(iii) a graph database such as Neo4j*.

The importing subsystem takes source files and a graph schema generated by `FLASc` as inputs. The subsystem then creates database creation scripts in `Cypher` by following the intensional and extensional information captured in the graph schema. The subsystem then interacts with the Neo4j graph database by using the APIs and executes the database creation scripts on the graph database.

*Importing subsystem design* The importing subsystem is based on the Extract-Transform-Load (ETL) design pattern. As shown in Fig. 8 the *Extract* stage is used to fetch data from a source and consolidated it into a repository. The *transform* stage is used to apply appropriate transformation rules over the repository data. The transform stage uses the graph schema generated by `FLASc` to apply the transformation rules and create the database creation scripts. The *load* stage is finally used to execute the scripts on the database. In the load stage, database is accessed by using the specific API calls.

*Technology stack* The subsystem is developed as a Java Maven project where the front end is designed using Java Swing library.[6] The subsystem uses Neo4j libraries for establishing a connection with the Neo4j graph database. Maven is used for handling API specific external dependencies. Neo4j's Cypher language is used for querying and creating the database.

### 5.2 Airbnb case study

Transforming and loading data in CSV format is straight forward in Neo4j and `Cypher`. Furthermore, the Airbnb data-set exists in the form of denormalized relational tables as such connection between nodes can be established based on primary key foreign key relationships. As shown in Query 6, the clause `LOAD CSV WITH HEADERS FROM` represents the *extract* stage. In Query 6 the data is being fetched from the Airbnb website as shown in line 2. The data is stored in a repository represented by the "row" variable in the query. The *transform* stage in Query 6 is represented in lines 3–6 where the `MATCH` clause is used to search for the existence of patterns, `WHERE` clause is used to restrict the result set based on some conditions and `WITH` clause serves as a medium to deliver the data (*listings and row*) to the `CREATE` clause. Finally the `CREATE` clause is used to create the edge between node labeled as `listing` and `booking_details`. The transform stage is also responsible for ensuring that the integrity constraints are enforced, which is done by using the graph schema. In a layered approach, the *load* stage is responsible for creating a connection with the Neo4j graph database by making appropriate API calls. The additional wrapper written in Java is used to execute the entire query on Neo4j finally.

The main advantage of using the layered approach is that additional logic can be written to ensure data consistency. For instance, `Cypher` does not provide inbuilt mechanisms to enforce the uniqueness constraints on edges. A layered approach is beneficial in such scenarios as additional logic can be written in programming languages to generate unique values for a particular edge property. The layered approach's advantage is evident when data in formats other than CSV are to be loaded into the Neo4j graph database. To illustrate this, we present the use of our layered approach to transform and load data-set related to big data analytics case study.

[6] The source code is available for download at https://github.com/emsoftaut/FLASc_ASE.

### 5.3 `BiDaML` case study

Implementing large-scale big data projects requires ongoing collaborations and monitoring by multiple stakeholders who have differing concerns. `BiDaML` (Big Data Analytics Modelling Languages) (Khalajzadeh et al. 2019) is a domain-specific language for planning, specifying, monitoring and designing big data analytics projects. `BiDaML` suite presents different graph-based diagrams with highly interrelated data. The `BiDaML` diagrams considered in this case study consists of five diagrams brainstorming, process, technique, data, and deployment that provide different levels of abstractions. These diagrams are generated for National Bowel Cancer Screening Program (NBCSP) in Australia (AGD of Health 2017).

The `BiDaML` suite currently lacks the necessary automation and tooling required to allow individual users to view customised information specific to their needs and preferences within these diagrams. Importing data-sets from highly structured tools, such as the current HTML based implementation of `BiDaML` diagrams into graph databases such as Neo4j, is a challenge. This is due to the reason that Neo4j does not provides clauses for importing HTML data. We illustrate the use of our schema driven approach for transforming and loading `BiDaML` diagrams into Neo4j.

#### 5.3.1 `BiDaML` diagrams data-set

The `BiDaML` data-set consists of five diagrams generated by the `BiDaML` suite. *Brainstorming diagram* provides an overview of a data analytics project and all the tasks and sub-tasks involved in designing the solution at a very high level. Users can include comments and extra information for the other stakeholders. *Process diagram* specifies the analytics process, which includes sequencing the tasks identified in the brainstorming diagram and relating these tasks to participants or stakeholders. *Technique diagrams* show how tasks from the brainstorming/process diagrams are elaborated further by applying specific techniques. *Data diagrams* document the data and artefacts produced in each of the above diagrams at a low level, i.e. the technical AI-based layer. They also define the outputs associated with different tasks like output information, reports, results, visualisations, and outcomes. And finally, *deployment diagrams* depicts the run-time configuration, i.e. the system hardware, the software installed on it, and the middle-ware connecting different machines for development related tasks.

The graph schema generated by using `FLASc` for `BiDaML` diagrams is presented in Fig. 9 where the node labeled as `TASK` allows edges that are available in different diagrams, including outgoing edges to other tasks allowed in brainstorming, process and technique diagrams. These edges are distinguished from each other via additional edge labels. For instance, edges between task nodes in brainstorming diagrams are labeled as `TT`. Edges between task nodes in process diagrams are labeled by `PR`. The schema also allows other node labels like `ROOT` in brainstorming diagrams, `START`, `END` and `CONDITION` in process diagrams and `INFRASTRUCTURE` node labels in deployment diagrams. In `BiDaML`, technique and data diagrams can have techniques and data artefacts that are used as nodes in deployment diagrams. For simplicity of the graph schema, we classify techniques, artefacts, etc.,

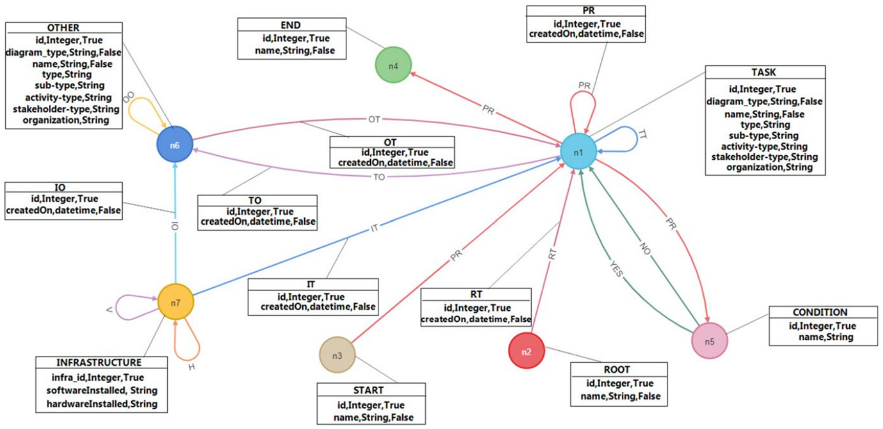

**Fig. 9** Logical graph schema for BiDaML diagrams

as nodes of label OTHER. As shown in Fig. 9 graph schema also captures the extensional information such as mandatory, unique and optional properties related to nodes and edges of BiDaML diagrams. For example, the node labeled as TASK has nine associated properties where *id,diagram_type* and *name* are mandatory properties. The *id* property must be unique and properties including *type,activity_type* and *organization* are optional.

### 5.3.2 Importing subsystem for BiDaML diagrams data-set

To transform and load BiDaML diagram data-set into Neo4j we still use the same ETL design pattern with slight modification to each stage. As shown in Fig. 10 data files in HTML format are passed to the *Extract* stage that consists of two processes: *Parse-HTML* and *Data builder*. The HTML file contains information about nodes and edges of BiDaML graphs using *map* tags as well as additional properties such as *id, name, type, sub-type, activity-type, stakeholder, comments* and *organization*. *Parse-HTML* process reads the entire HTML file by using the JSoup library Hedley (2020) and creates a repository containing all the nodes and edges, which is then passed on to the *Data Builder* for further processing.

The *Data builder* process first removes duplicate elements in the repository. The builder then converts the repository into a list of edges (and nodes) that need to be stored in the graph database. In the *Transform* stage, the *Cypher Query Builder* takes the edge list from the extract stage and graph schema generated using FLASc as inputs to generate Cypher queries for loading data into Neo4j. This stage also ensures that appropriate integrity constraints captured in the graph schema are enforced.

The final *load* stage consists of a *Database Connector* process and a Neo4j graph database interface. The *Database Connector* process establishes a connection with the Neo4j graph database using the Neo4j interface. A session is created between the

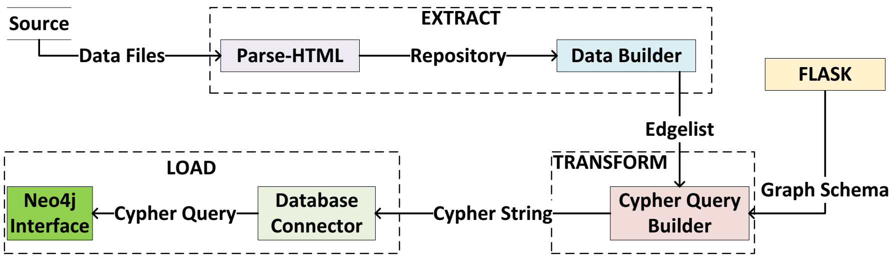

**Fig. 10** ETL stages shown as Data flow diagram to upload `BiDaML` diagram data-set into Neo4j

subsystem and the Neo4j database. The Cypher query constructed in the transform stage is packaged into a create query and then executed. This process also ensures that nodes are not duplicated, especially if some of the imported nodes were already present in the database.

The time at which each node or edge is created during the ETL operations or during subsequent editing of the diagrams, is stored as a time stamp attribute within each updated element. Additional information, such as clustering of tasks in brainstorming diagrams and mapping tasks to specific stakeholders, is all stored as attributes of the corresponding nodes.

## 5.4 P2660.1 case study

Designing robust Industrial Cyber-Physical Systems (ICPS) largely depends upon identifying industrial agents, that provide complex and harmonious control mechanisms at the software level. These industrial agents practices are used to develop more extensive and feature-rich ICPS. IEEE Standardization projects such as P2660.1 aim at identifying industrial agent practices that can suit the requirements of future ICPS. A key challenge with this project is the identification of industrial agent practices based on some user-defined criteria. This case study is based on a tool (`IASelect`[7]) developed for IEEE standardization project P2660.1 (P2660.1 2020) that assists in selecting best fit industrial agent practices for ICPS (Sharma et al. 2019).

### 5.4.1 P2660.1 data-set

The P2660.1 data-set consists of two practices *OnDevice* and *Hybrid*. Each practice is of two types *Tightly-coupled* and *loosely-coupled*. Practices have an associated set of qualities, which make these practices suitable to use in specific contexts. Hence, selecting the best-fit practices requires identifying the associated qualities. P2660.1 working group identifies four kinds of qualities *Domain,*

[7] `The source code is available for download at` https://github.com/chandanNapster/INDIN_Neo4j_Web.

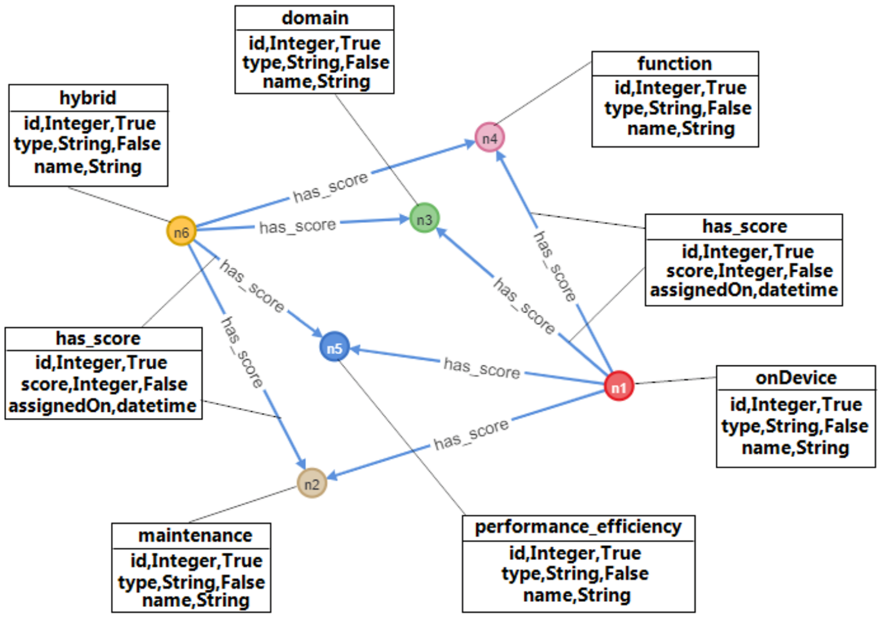

**Fig. 11** Logical graph schema for P2660.1 data-set

*Function, Maintenance* and *Performance efficiency*. Each quality has an associated type; for instance, *Domain* has three associated types, including *Factory Automation, Building Automation* and *Energy*. Similarly, quality *Function* has three associated types *Monitoring, Control* and *Simulation*. The P2660.1 data-set exists in the form of an adjacency matrix where an ICPS expert assigns a score to a combination of practice and associated quality.

The graph schema generated by using `FLASc` for P2660.1 data-set is presented in Fig. 11 which consist of two practice nodes and four quality nodes. Each practice node is connected to a quality node by an edge labeled as `has_score`. This signifies that every practice to be stored in the graph database must connect with a quality, which represents the intensional information associated with the data-set. The extensional information is captured by node and edge properties. All nodes and edges have an associated property *id* which is a mandatory property, is of Integer data type and value associated with this property must be unique. Property such as *type* is mandatory but may not be unique. All edges have a unique and mandatory property *id*. The *score* property is mandatory but is not unique, and this is because the same score value can be assigned to different practice-quality pair by an ICPS expert. All edges contain an optional property *assignedOn* with an associated data type date-time.

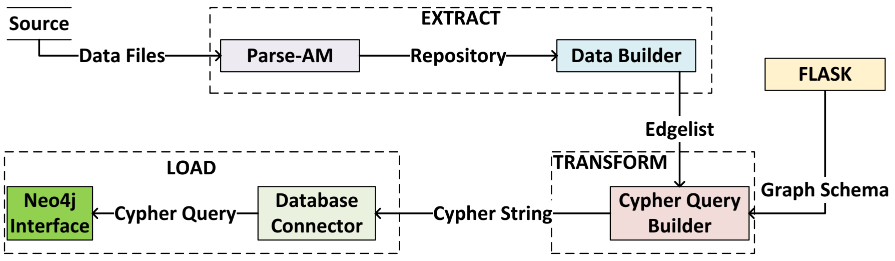

**Fig. 12** ETL stages shown as Data flow diagram to upload P2660.1 data-set into Neo4j

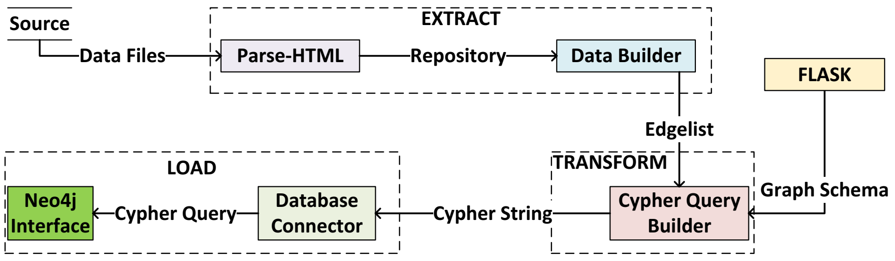

**Fig. 13** ETL stages shown as Data flow diagram to upload BiDaML diagram data-set into Neo4j

### 5.4.2 Importing subsystem for P2660.1 data-set

To transform and load the P2660.1 data-set into Neo4j, we use the ETL design pattern with slight modifications. As shown in Fig. 12 data in XLS file format containing an adjacency matrix is passed to the *Extract* stage that consists of two processes *Parse-AM* and *Data builder.*

The *Parse-AM* process is used to reads the entire XLS file by using the Apache POI library (Apache 2020) and converts it into a repository. The other process required to transform and load the P2660.1 data-set into Neo4j are similar to the processes used in the BiDaML diagram case study presented in Sect. 5.3.2.

## 5.5 Lessons learned from the case studies

The formal basis for FLASc and its integration with the ETL design pattern suggests that the data from heterogeneous sources can be transformed and loaded into several graph database by using our approach. We consider three case studies related to cyber-physical systems, big data analytics and tourism as presented in Sects. 5.2, 5.3 and 5.4 respectively. The only factor that differs in loading these three diverse data-sets is the Extract phase's parse process.

As shown in Figs. 13 and 14 the parse process uses different APIs for reading data from heterogeneous sources. All other stages for loading data into the Neo4j

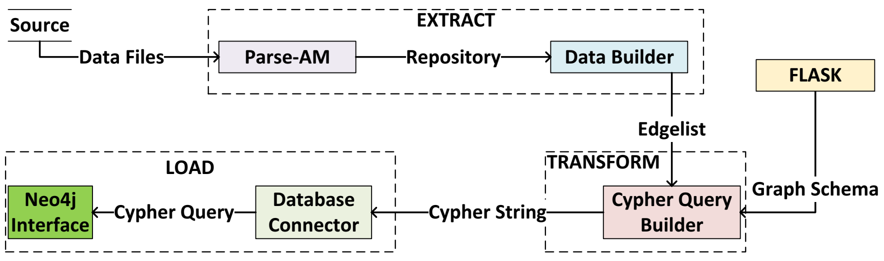

**Fig. 14** ETL stages shown as Data flow diagram to upload P2660.1 data-set into Neo4j

**Table 4** Coverage of integrity constraints

| | Integrity constraints | Integrated FLASc | Layered FLASc | Layered without FLASc |
|---|---|---|---|---|
| Graph entity | Node Property Uniqueness | ✓ | ✓ | ✓ |
| | Node/Edge Label Uniqueness | ✓ | ✓ | × |
| | Edge property uniqueness | × | ✓ | × |
| | Mandatory Node property | ✓ | ✓ | ✓ |
| | Mandatory Edge property | ✓ | ✓ | ✓ |
| | Property data type | ✓ | ✓ | × |
| Semantic | Edge pattern | ✓ | ✓ | × |
| | Graph pattern | ✓ | ✓ | × |
| | Path pattern | ✓ | ✓ | × |
| Others | Type checking | ✓ | ✓ | × |
| | Edge Cardinality | × | ✓ | × |
| | Relationship Type | × | × | × |

graph database remain the same. Similarly, suppose data has to be transformed and loaded into a database other than Neo4j. In that case, only the Load stage needs to be altered so that APIs specific to the database platform can be utilized. The transform stage in all the scenarios as mentioned above remains the same and consistent. This demonstrates the generalizability of our approach, since by using the FLASc integrated ETL design pattern can be used to load data-sets from heterogeneous sources into a graph database. Furthermore, our approach is not limited to a specific data-set format and a particular graph database.

The use of FLASc for loading data-sets from heterogeneous sources becomes more evident when using the layered approach. As shown in Table 4 only a limited number of integrity constraints can be enforced in a layered approach without using FLASc. As shown in Table 3 structured data-sets such as provided in the Airbnb case study exist in the form of CSV files and contain intensional information as primary and foreign keys. However, semi-structured data provided in BiDaML and P2660.1 data-sets require predefined structural information for systematic transformation and loading. The intensional information is facilitated by using `FLASc` hence ensuring data consistency and integrity while using the layered approach.

## 6 Discussion, conclusion and future work

In this research, we present a formal algebra `FLASc` for generating robust graph schema for labeled property graph databases. We illustrate the integration of `FLASc` with the Extract-Transform-Load design pattern that assists in systematic transformation and loading of data-sets from heterogeneous sources into graph databases such as Neo4j. Graph schemas generated by `FLASc` assist in specifying integrity constraints in the database creation scripts, ensuring data consistency and integrity.

Our approach presents the integration of conceptual, logical and physical modeling stages for graph databases. `FLASc` enables users to capture requirements of any given problem domain as basic conceptual graph schemas. The `JOIN`, `DETACH` and `DELETE_NODE` operators provided by `FLASc` can then be used to construct robust conceptual graph schemas from basic conceptual graph schemas. Properties associated with nodes and edges of graph schema are specified at the logical modeling stage. Finally, in the physical modeling stage, the enforcement of integrity constraints and design of database creation scripts are driven by `FLASc` generated graph schemas.

The integration of `FLASc` with the Extract-Transform-Load design pattern illustrates the practical application of our approach. This is demonstrated by using three diverse case studies related to cyber-physical systems, big data analytics and tourism that also illustrates the generalizability of our approach. The intensional and extensional information captured in the graph schema assists in the *transform* stage of the data loading process. This information can be used to enforce several integrity constraints on the data-sets being loaded into a graph database.

As shown in Table 4, `FLASc` facilitates the enforcement of several integrity constraints. We can observe that `FLASc` generated graph schemas are useful in enforcing semantic constraints because such constraints require knowledge of relationships between entities in data-sets. Semantic constraints such as edge, graph and path pattern constraints cannot be enforced without knowledge about relationships in the data-set. As shown in Table 4 graph entity integrity constraints such as edge property uniqueness constraint cannot be enforced in the integrated approach due to the limitations in the Neo4j graph database. Furthermore, `FLASc` generated logical graph schema also enable a database designer to specify cardinality constraints on the edges of a graph schema. However, due to the limitations in Neo4j graph database cardinality constraints cannot be enforced in the integrated approach. Such challenges can be mitigated in the layered approach by writing additional logic in programming languages such as Java, Python for specifying edge uniqueness and cardinality constraints.

The use of `FLASc` for loading data from heterogeneous sources becomes more evident while using the layered approach. As shown in Table 4 only a limited number of integrity constraints can be enforced in a layered approach without using `FLASc`. The support for integrity constraints such as node property uniqueness, mandatory node and edge property constraints are by default provided by Neo4j. Other constraints cannot be enforced without the intensional and extensional information captured in the graph schemas generated by `FLASc`. In the absence of robustly defined graph schema, the capability to enforce integrity constraints depends on the underlying engine associated with a graph database .

## 6.1 Limitations

As shown in Table 4 graph schemas generated by `FLASc` provide the ability to enforce several useful integrity constraints. However, other constraints such as relationship types is not covered in our approach. Relationship types represent the nature of relationships such as inheritance, association, composition and realisation, between nodes of a graph database. The enforcement of such constraints is not supported by `FLASc` in its current state. Furthermore, `FLASc` cannot be compared with other conceptual modeling tools such as entity-relationship diagrams (ERD) and unified modeling language (UML) diagrams as these tools support the specification of relationship types.

The main motive of `FLASc` is to assist in the design of robust conceptual graph schemas so that the soundness of logical and physical graph schemas can be ensured. `FLASc` generated conceptual graph schemas can preciously capture the intensional information. Relationship types are edge related properties (Angles 2018); hence can be classified as extensional information. These properties can be easily captured in the logical graph schema. For instance, by altering Definition 7, the logical graph schema can be enriched to support extensional information such as relationship types.

## 6.2 Conclusion and future work

The scope of our study is limited to the Neo4j graph database. Therefore, the performance evaluation of using our approach for transforming and loading data-sets into other graph databases is not discussed. We consider this as future work where `FLASc` can be utilised for evaluating the coverage of integrity constraints offered by other graph databases provided by vendors such as Oracle (2021), Apache Tinkerpop (2021) and TigerGraph (2020). We intend to work on extending `FLASc` to support other integrity constraints such as relationship types and functional dependencies. The support of such constraints can enable `FLASc` to represent visual models expressed in languages such as Entity relationship diagram (ERD), Unified Modeling Language (UML) and System Modeling Language (SysML).

Moreover, using the `FLASc` extended ETL design pattern, visual models expressed as ERD, UML or SysML diagrams related to software development projects can be imported into graph databases. Storing software development visual models in graph databases provides the additional advantages of tractability and efficient database manageability, such as automatically identifying inconsistencies across all project diagrams.

In its current state our formal algebra FLASc supports the creation of robustly defined graph schemas that captures the intensional and extensional information. A natural extension to this work is the proposal of a formal schema creation language. We intend to combine our novel query language proposed in Sharma et al. (2021) with FLASc to propose a graph schema creation language. In Sharma et al. (2021) we propose the novel formalims of conjunctive queries and union of conjunctive queries extended with Tarksi's algebra (CQT/UCQT) for extracting data stored in a graph database. This language can be further combined with FLASc for creating a novel graph schema creation language. A main advantage of such an approach is the ability to use restricted form of first-order logic (conjunctive queries) while defining a graph schema which also makes our approach compatible with object role modeling language proposed in Halpin (2005). This will further assist in the industry wide initiative of standardizing query language for graph databases.

**Funding** Open Access funding enabled and organized by CAUL and its Member Institutions.

## References

AGD of Health: National Bowel Cancer Screening Program (2017). https://www1.health.gov.au/internet/main/publishing.nsf/Content/nbcsp.htm

Airbnb: Inside Airbnb: Adding data to the debate. Accessed: 2019-02-03 (2018). http://insideairbnb.com/get-the-data.html

Alex, A., Norbert, M.: LDBC Use case analysis and choke point analysis. Accessed: 2019-03-01 (2013). http://ldbcouncil.org/sites/default/files/LDBC_D3.3.1.pdf

Amann, B., Scholl, M.: Gram: a graph data model and query languages. In: Proceedings of the ACM Conference on Hypertext, pp. 201–211 (1993)

Angles, R., Arenas, M., Barceló, P., Boncz, P., Fletcher, G., Gutierrez, C., Lindaaker, T., Paradies, M., Plantikow, S., Sequeda, J., et al.: G-core: A core for future graph query languages. In: Proceedings of the 2018 International Conference on Management of Data, pp. 1421–1432 (2018). ACM

Angles, R., Bonifati, A., Dumbrava, S., Fletcher, G., Hare, K.W., Hidders, J., Lee, V.E., Li, B., Libkin, L., Martens, W., et al.: Pg-keys: Keys for property graphs. In: Proceedings of the 2021 International Conference on Management of Data, pp. 2423–2436 (2021)

Angles, R., Thakkar, H., Tomaszuk, D.: Rdf and Property Graphs Interoperability: Status and Issues (2019)

Angles, R.: A comparison of current graph database models. In: 2012 IEEE 28th International Conference on Data Engineering Workshops, pp. 171–177 (2012). IEEE

Angles, R.: The property graph database model. In: AMW (2018)

Angles, R., Gutierrez, C.: Survey of graph database models. ACM Computing Surveys (CSUR) **40**(1), 1–39 (2008)

Angles, R., Boncz, P., Larriba-Pey, J., Fundulaki, I., Neumann, T., Erling, O., Neubauer, P., Martinez-Bazan, N., Kotsev, V. and Toma, I.: The linked data benchmark council: a graph and rdf industry benchmarking effort. SIGMOD Record 43(1): 27 (2014)

Angles, R., Arenas, M., Barceló, P., Hogan, A., Reutter, J., Vrgoč, D.: Foundations of modern query languages for graph databases. ACM Computing Surveys (CSUR) **50**(5), 1–40 (2017)

Angles, R., Thakkar, H., Tomaszuk, D.: Mapping rdf databases to property graph databases. IEEE Access 8, 86091–86110 (2020)

Apache: ”Apache java library for parsing XLS document”. Accessed: 2021-01-17 (2020). https://mvnrepository.com/artifact/org.apache.poi/poi

Apache: Apache TinkerPop. Accessed: 2021-01-02 (2021). https://tinkerpop.apache.org/

Apache: Gremlin query language Apache TinkerPop. Accessed: 2021-01-02. https://tinkerpop.apache.org/docs/current/tutorials/gremlin-language-variants/

Badia, A., Lemire, D.: A call to arms: revisiting database design. ACM SIGMOD Record 40(3), 61–69 (2011)

Bagan, G., Bonifati, A., Ciucanu, R., Fletcher, G.H., Lemay, A., Advokaat, N.: Controlling diversity in benchmarking graph databases. arXiv preprint arXiv:1511.08386 (2015)

Barceló, P., Libkin, L., Reutter, J.L.: Querying graph patterns. In: Proceedings of the Thirtieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems, pp. 199–210 (2011)

Barceló, P., Libkin, L., Lin, A.W., Wood, P.T.: Expressive languages for path queries over graph-structured data. ACM Transactions on Database Systems (TODS) **37**(4), 31 (2012)

Barceló, P., Pérez, J., Reutter, J.L.: Relative expressiveness of nested regular expressions. AMW 12, 180–195 (2012)

Barceló, P., Romero, M., Vardi, M.Y.: Semantic acyclicity on graph databases. SIAM Journal on computing 45(4), 1339–1376 (2016)

Barik, M.S., Mazumdar, C., Gupta, A.: Network vulnerability analysis using a constrained graph data model. In: International Conference on Information Systems Security, pp. 263–282 (2016). Springer

Bell, G., Hey, T., Szalay, A.: Beyond the data deluge. Science 323(5919), 1297–1298 (2009)

Berners-Lee, T., Hendler, J., Lassila, O., et al.: The semantic web. Scientific american **284**(5), 28–37 (2001)

Bonifati, A., Fletcher, G., Voigt, H., Yakovets, N.: Querying graphs. Synthesis Lectures on Data Management **10**(3), 1–184 (2018)

Brodie, M.L., Liu, J.T.: The power and limits of relational technology in the age of information ecosystems. In: On the Move Federated Conferences (2010)

Castro, J., Soto, A.: A comparison between cypher and conjunctive queries. In: AMW (2017)

Chein, M., Mugnier, M.-L.: Graph-based Knowledge Representation: Computational Foundations of Conceptual Graphs. Springer, Berlin (2008)

Chen, P.P.-S.: The entity-relationship model-toward a unified view of data. ACM Trans. Database Syst. (TODS) **1**(1), 9–36 (1976)

Clarke, E.M., Wing, J.M.: Formal methods: State of the art and future directions. ACM Computing Surveys (CSUR) **28**(4), 626–643 (1996)

Codd, E.F.: A relational model of data for large shared data banks. In: Software Pioneers, pp. 263–294. Springer, ??? (2002)

Daniel, G., Sunyé, G., Cabot, J.: Umltographdb: mapping conceptual schemas to graph databases. In: International Conference on Conceptual Modeling, pp. 430–444 (2016). Springer

de Sousa, V.M., Cura, L.M.d.V.: Logical design of graph databases from an entity-relationship conceptual model. In: Proceedings of the 20th International Conference on Information Integration and Web-based Applications & Services, pp. 183–189 (2018)

Finkelstein, S., Schkolnick, M., Tiberio, P.: Physical database design for relational databases. ACM Transactions on Database Systems (TODS) **13**(1), 91–128 (1988)

Fitzgerald, G., Philippides, A., Probert, S.: Information systems development, maintenance and enhancement: findings from a uk study. Int. J. Inf. Manage. **19**(4), 319–328 (1999)

Florescu, D., Levy, A., Suciu, D.: Query containment for conjunctive queries with regular expressions. In: PODS, vol. 9, pp. 139–148 (1998)

Frozza, A.A., Jacinto, S.R., dos Santos Mello, R.: An approach for schema extraction of nosql graph databases. In: 2020 IEEE 21st International Conference on Information Reuse and Integration for Data Science (IRI), pp. 271–278 (2020). IEEE

Ghrab, A., Romero, O., Skhiri, S., Vaisman, A., Zimányi, E.: Grad: On graph database modeling. arXiv preprint arXiv:1602.00503 (2016)

Ghrab, A., Romero, O., Skhiri, S., Zimányi, E.: Analytics-Aware Graph Database Modeling. Technical report, Technical report (2014)

Graves, M., Bergeman, E.R., Lawrence, C.B.: A graph-theoretic data model for genome mapping databases. In: Proceedings of the Twenty-Eighth Annual Hawaii International Conference on System Sciences, vol. 5, pp. 32–41 (1995). IEEE

Griffith, R.L.: Three principles of representation for semantic networks. ACM Transactions on Database Systems (TODS) **7**(3), 417–442 (1982)

Güting, R.H.: Graphdb: Modeling and querying graphs in databases. In: VLDB, vol. 94, pp. 12–15 (1994). Citeseer

Gyssens, M., Paredaens, J., Van den Bussche, J., Van Gucht, D.: A graph-oriented object database model. IEEE Transactions on knowledge and Data Engineering 6(4), 572–586 (1994)

Halpin, T.: Orm 2. In: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", pp. 676–687 (2005). Springer

Hartig, O., Hidders, J.: Defining schemas for property graphs by using the graphql schema definition language. In: Proceedings of the 2nd Joint International Workshop on Graph Data Management Experiences & Systems (GRADES) and Network Data Analytics (NDA), pp. 1–11 (2019)

Hedley, J.: jsoup: Java HTML Parser. Accessed: 2020-04-05 (2020). https://jsoup.org/

Hidders, J.: Typing graph-manipulation operations. In: International Conference on Database Theory, pp. 394–409 (2003). Springer

Johnson, R.B., Onwuegbuzie, A.J., Turner, L.A.: Toward a definition of mixed methods research. Journal of mixed methods research 1(2), 112–133 (2007)

Khalajzadeh, H., Abdelrazek, M., Grundy, J., Hosking, J., He, Q.: Bidaml: A suite of visual languages for supporting end-user data analytics. In: 2019 IEEE International Congress on Big Data (BigDataCongress), pp. 93–97 (2019). IEEE

Khalajzadeh, H., Simmons, A., Abdelrazek, M., Grundy, J., Hosking, J., He, Q.: An end-to-end model-based approach to support big data analytics development. J. Comput. Lang. **58**, 100964 (2020)

Khan, A., Wu, Y., Yan, X.: Emerging graph queries in linked data. In: 2012 IEEE 28th International Conference on Data Engineering, pp. 1218–1221 (2012). IEEE

Kunii, H.S.: Dbms with graph data model for knowledge handling. In: Proceedings of the 1987 Fall Joint Computer Conference on Exploring Technology: Today and Tomorrow, pp. 138–142 (1987)

Lassila, O., Swick, R.R., et al.: Resource description framework (rdf) model and syntax specification. World Wide Web (1998)

Levene, M., Poulovassilis, A.: The hypernode model and its associated query language. In: Proceedings of the 5th Jerusalem Conference on Information Technology, 1990.'Next Decade in Information Technology', pp. 520–530 (1990). IEEE

Levene, M., Loizou, G.: A graph-based data model and its ramifications. IEEE Transactions on Knowledge and Data Engineering 7(5), 809–823 (1995)

Levene, M., Poulovassilis, A.: An object-oriented data model formalised through hypergraphs. Data & Knowledge Engineering **6**(3), 205–224 (1991)

Marciniak, J.J.: Encyclopedia of Software Engineering. Wiley-Interscience, New York (1994)

Megid, Y.A., El-Tazi, N., Fahmy, A.: Using functional dependencies in conversion of relational databases to graph databases. In: International Conference on Database and Expert Systems Applications, pp. 350–357 (2018). Springer

Mior, M.J., Salem, K., Aboulnaga, A., Liu, R.: Nose: Schema design for nosql applications. IEEE Transactions on Knowledge and Data Engineering 29(10), 2275–2289 (2017)

Mugnier, M.-L., Chein, M.: Conceptual graphs: fundamental notions. Revue d'intelligence artificielle **6**(4), 365–406 (1992)

Neo4j: Neo4j. Accessed: 2021-02-27 (2021). https://neo4j.com/

OpenCypher: OpenCypher. Accessed: 2018-10-01 (2018). https://www.opencypher.org/

Oracle: Oracle. Accessed: 2021-02-27 (2021). https://www.oracle.com/middleware/technologies/parallel-graph-analytix.html

P2660.1: "Recommended practices on industrial agents: Integration of software agents and low level automation functions.". Accessed: 2021-03-16 (2020). https://standards.ieee.org/standard/2660_1-2020.html

Paredaens, J., Peelman, P., Tanca, L.: G-log: A graph-based query language. IEEE Transactions on Knowledge and Data Engineering 7(3), 436–453 (1995)

Park, Y., Shankar, M., Park, B.-H., Ghosh, J.: Graph databases for large-scale healthcare systems: A framework for efficient data management and data services. In: 2014 IEEE 30th International Conference on Data Engineering Workshops, pp. 12–19 (2014). IEEE

Pérez, J., Arenas, M., Gutierrez, C.: Semantics and complexity of sparql. In: International Semantic Web Conference, pp. 30–43 (2006). Springer

Pokornỳ, J.: Conceptual and database modelling of graph databases. In: Proceedings of the 20th International Database Engineering & Applications Symposium, pp. 370–377 (2016)

Pokorny, J.: Modelling of graph databases. Journal of Advanced Engineering and Computation **1**(1), 04–17 (2017)

Pokornỳ, J., Valenta, M., Kovačič, J.: Integrity constraints in graph databases. Procedia Computer Science **109**, 975–981 (2017)

Reina, F., Huf, A., Presser, D., Siqueira, F.: Modeling and enforcing integrity constraints on graph databases. In: International Conference on Database and Expert Systems Applications, pp. 269–284 (2020). Springer

Reutter, J.L.: Containment of nested regular expressions. arXiv preprint arXiv:1304.2637 (2013)

Rodriguez, M.A., Neubauer, P.: The graph traversal pattern. In: Graph Data Management: Techniques and Applications, pp. 29–46. IGI Global, ??? (2012)

Rodriguez, M.A., Neubauer, P.: Constructions from dots and lines. Bulletin of the American Society for Information Science and Technology 36(6), 35–41 (2010)

Roy-Hubara, N., Rokach, L., Shapira, B., Shoval, P.: Modeling graph database schema. IT Professional 19(6), 34–43 (2017)

Sciore, E., Siegel, M., Rosenthal, A.: Using semantic values to facilitate interoperability among heterogeneous information systems. ACM Transactions on Database Systems (TODS) **19**(2), 254–290 (1994)

Šestak, M., Rabuzin, K., Novak, M.: Integrity constraints in graph databases–implementation challenges. In: Proceedings of Central European Conference on Information and Intelligent Systems, pp. 23–30 (2016)

Šestak, M., Heričko, M., Družovec, T.W., Turkanović, M.: Applying k-vertex cardinality constraints on a neo4j graph database. Future Generation Computer Systems 115, 459–474 (2021)

Sharma, C., Sinha, R., Johnson, K.: Practical and comprehensive formalisms for modeling contemporary graph query languages. Inf. Syst. **102**, 101816 (2021)

Sharma, C., Sinha, R., Leitao, P.: Iaselect: Finding best-fit agent practices in industrial cps using graph databases. In: 2019 IEEE 17th International Conference on Industrial Informatics (INDIN), vol. 1, pp. 1558–1563 (2019). IEEE

Sharma, C., Sinha, R.: A schema-first formalism for labeled property graph databases: Enabling structured data loading and analytics. In: Proceedings of the 6th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies, pp. 71–80 (2019)

Sharma, C.: Design of formal query languages and schemas for graph databases. PhD thesis, Auckland University of Technology (2021)

Sharma, C.: Flux: From sql to gql query translation tool. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1379–1381 (2020). IEEE

Sowa, J.: Conceptual graphs: Draft proposed american national standard. In: International Conference on Conceptual Structures, pp. 1–65 (1999). Springer

Sowa, J.F.: Conceptual graphs for a data base interface. IBM Journal of Research and Development 20(4), 336–357 (1976)

Sowa, J.F.: Conceptual graphs summary. Conceptual Structures: current research and practice **3**, 66 (1992)

Sowa, J.F.: Conceptual graphs. Foundations of Artificial Intelligence **3**, 213–237 (2008)

Tetko, I.V., Engkvist, O., Koch, U., Reymond, J.-L., Chen, H.: Bigchem: challenges and opportunities for big data analysis in chemistry. Mol. Inf. **35**(11–12), 615–621 (2016)

TigerGraph: A Modern graph query language. Accessed: 2020-28-06 (2020). https://www.tigergraph.com/gsql/

Tucker, J., Stephenson, K.: Data, syntax and semantics. Citeseer (2003)

W3C: Resource Description Framework. Accessed: 2021-02-27 (2021). https://www.w3.org/RDF/

W3C: SPARQL 1.1 Query Language W3C Recommendation. Accessed: 2021-01-02 (2013). https://www.w3.org/TR/2013/REC-sparql11-query-20130321/#pp-language

Wood, P.T.: Query languages for graph databases. ACM Sigmod Record 41(1), 50–60 (2012)

Yu, Y., Heflin, J.: Extending functional dependency to detect abnormal data in rdf graphs. In: International Semantic Web Conference, pp. 794–809 (2011). Springer