Employing Graph Databases As a Standardization Model for Addressing Heterogeneity and Integration

Dippy Aggarwal and Karen C. Davis

University of Cincinnati, Cincinnati OH 45220, USA
aggarwdy@mail.uc.edu, karen.davis@uc.edu

Abstract. The advent of big data and NoSQL data stores has led to the proliferation of data models exacerbating the challenges of information integration and exchange. It would be useful to have an approach that allows leveraging both schema-based and schema-less data stores. We present a graph-based solution that attempts to bridge the gap between different data stores using a homogeneous representation. As the first contribution, we present and demonstrate a mapping approach to transform schemas into a homogeneous graph representation. We demonstrate our approach over relational and RDF schemas but the framework is extensible to allow further integration of additional data stores. The second contribution is a schema merging algorithm over property graphs. We focus on providing a modular framework that can be extended and optimized using different schema matching and merging algorithms.

Keywords: schema integration, graph databases, schema mapping, Neo4j

1 Introduction

Recently, there has been a shift in the volume of data generated, diversity in data forms (structured, semi-structured and unstructured) and an unprecedented rate at which data is produced. The data possessing these characteristics is termed as big data and the field is forecasted to grow at a 26.4% compound annual growth rate through 2018, according to a report released by Intelligent Data Corporation [34]. The need for faster analysis over big data with current storage and computation power poses a major challenge for enterprise data infrastructures. Two alternatives to handle analysis over this newer form of data are: (a) scale-up (adding CPU power, memory to a single machine), and (b) scale-out (adding more machines in the system creating a cluster). The main advantage of scaling out vs. scaling up is that more work can be done by parallelizing the workload across distributed machines. Many existing relational database systems are designed to perform efficiently on single-machines. It should not be misconstrued that relational systems cannot scale-out at all. They can, but they lose the features that they are primarily designed such as ACID compliance, for example [35].

NoSQL (Not Only SQL) describes an emerging class of storage models designed for scalable database systems. NoSQL data stores advocate new and relaxed forms of data storage that do not require a schema to be enforced for the underlying data. Instead, a schema is identified and generated at the application side when the data is read from the system. This concept of postponing schema definition to a later point has enabled NoSQL storage models to be applied in many real-world use-cases. However, the popularity has also created a notion that schemaless data management techniques are more suitable for solving emerging data problems than schema-based structures.

While NoSQL data stores offer interesting and novel solutions for managing big data, they are also not a panacea for all data management related scenarios. In scenarios that need query optimization, data governance, and integrity, schema-based stores offer a better solution. Furthermore, a large amount of enterprise data still resides in relational databases. The greater scalability of NoSQL databases over relational databases comes at a price. Most NoSQL systems compromise certain features, such as strong consistency, to achieve efficiency over other critical features of performance and availability. Organizations such as Facebook and Hadapt who have widely embraced big data technologies also choose a data store on a use-case basis as opposed to leveraging a single big data storage technique for all their applications and data storage requirements [36]. More recently, a number of SQL implementations have also emerged that are built over platforms and programming models such as MapReduce that support big data [37–42].

Another field that is growing rapidly is semantic web and linked data technologies. Linked data builds upon the existing web and offers the idea of annotating the web data (and not just the documents) [43] using global identifiers and linking them. The data is published and organized using RDF (Resource Description Framework), which is based on a subject-object-predicate framework. RDF schema specifications and modeling concepts differ from relational databases and NoSQL data models, thus supporting the need and demand for creating schema and data integration solutions. Hitzler et al. [43] identify linked data as a part of the big data landscape.

These recent developments indicate two ideas: (1) the significance and prevalence of structured data in the enterprise world, and (2) the unique advantages possessed by different classes of data stores. In order to reap benefits from all of them, it is important to bring them together under a homogeneous model. This would serve two purposes: (1) offer more complete knowledge by combining data stored in isolated sources, and (2) facilitate harnessing value from each of the data stores (schema-based and schema-less), thus making them complementary and not competitive solutions.

In this paper, we address this need by adopting graphs as a means towards standardization and integration of different data stores, thus handling the variety characteristic of big data. Our selection of a graph model is based on the following observations:

1. Graph databases are a NoSQL data storage model and thus support the big data processing framework [51].
2. Graphs provide a simple and flexible abstraction for modeling artifacts of different kinds in the form of nodes and edges.
3. Graph databases are attracting significant attention and interest in the past few years as highlighted from the Google Trends analysis shown in Figure 1. The values are normalized representing the highest value in the chart as 100% and the x-axis labels are marked in two-year time intervals.
4. The graph model adopted in our work, Neo4j, possesses a query language called Cypher and allows programmatic access using API.

Fig. 1. Trend of web search interest for graph databases [29]

The main contributions of this paper are as follows:

1. A concept-preserving, integrated graph model that addresses the model heterogeneity and variety dimension of the big data landscape.
2. A software-oriented, automated approach to transform relational and RDF schemas into a graph database.
3. A proof-of-concept that illustrates the potential of graph-based solutions towards addressing diversity in data representations.
4. A framework accompanied by a proof-of-concept for schema merging over property graphs.

The rest of the paper is organized as follows. Section 2 presents an overview of the concepts and terms that are used frequently in the paper. These include a discussion of property graphs and Neo4j graph database in particular, and relational and RDF data models as our native models of interest. Section 3 describes our transformation rules for converting a relational schema to a property graph. We leverage the approach proposed by Bouhali et al. [52] for converting RDF to a property graph representation and extend it to support additional models. Next, we present a proof-of-concept to illustrate the implementation of our transformation rules. We consider schema excerpts for relational and RDF

models and show their corresponding property graphs generated using the transformation rules. Section 4 introduces our architecture and the motivation behind mapping non-graph based schemas to a property graph. The evaluation issues for schema mapping are reported in Section 5. We next present our approach towards schema integration over property graphs in Section 6. The challenges and our proposed solution towards resolving them are illustrated using an example. Section 7 presents the schema integration algorithm. In section 8, we use two case-studies to evaluate our approach for schema integration. Section 9 addresses the related work in graph-based integration and transformations while Section 10 offers conclusions and future work.

2 Background

In this section, we overview the concepts that form a foundation for our research. These include relational and RDF schemas that serve as input schemas. There are many graph based models; we consider the property graph as our model of interest and introduce it here briefly. We leverage Neo4j [53] in our work.

2.1 RDF

RDF stands for Resource Description Framework. It refers to the model and RDF schema is commonly abbreviated as RDFS. RDF allows annotating web resources with semantics. The resources and semantic information is represented in the form of classes and properties which form two core concepts of an RDF schema.

The notion of classes and objects in RDFS differ from the similar concepts that exist in conventional, object-oriented systems [54]. In many systems, classes contain a set of properties and each of the class instances possesses those properties. However, in RDF, properties are described in terms of classes to which they apply. These classes are referred to as *domain* in RDF schema, and similarly, the values that a certain property can hold is described using *range*. For example, we could define a property *member* that has domain *Group* and its range would be *Person*. Nejdl et al. [44] and W3C specification [45] offer a summary of RDF classes and properties.

2.2 Relational Schema

A relational schema is described as a set of relations which consists of a a set of attributes and constraints. The relations are connected by different types of relationships with cardinality constraints. There are two major types of constraints: entity integrity and referential integrity. The entity integrity constraint states that every relation has a set of attributes, termed *primary key*, that uniquely identifies each of the tuples in the relation. The primary key attribute(s) may not be null. The referential integrity constraint applies to a pair of relations that are associated with each other. Having this constraint signifies that every value

Table 1. Relational schema excerpt. Top *Employee* and Bottom *Organization*

Name	SkypeID	Gender	DOB	employeeID	address	ssn	orgId
Marissa White	marissa@yahoo.com	Female	04-09-1983	mwhite	2600 clifton	1234567890	uc_org
Jason Doe	jason@yahoo.com	Male	04-09-1973	jdoe	3200 clifton	6789083455	uc_org

Name	Location	orgId
University of Cincinnati	2600 Clifton	uc_org

of one attribute in one of the participating relations comes from the set of values of a primary key attribute in the other relation.

2.3 Neo4j and Property Graphs

Graphs provide a simple and flexible abstraction for modeling artifacts of different kinds in the form of nodes and edges. The graph model adopted in our work, Neo4j [53] supports automation and has a query language [46]. The database is available for download for free and there is vast technical support and a large user base. Neo4j has also been ranked as the most popular graph database by db-engines.com [47].

The graph database in our work, Neo4j [53], organizes its data as a labelled property graph in which the nodes and relationships possess properties and can be annotated with labels. This allows augmenting the graph nodes and edges with semantics. The concept of properties is analogous to attributes in traditional conceptual models such as the Entity-Relationship Model. In property graphs, properties are key-value pairs. Figure 2 presents an example of data modeling using a property graph.

Some key points to note in Figure 2 are as follows: (1) the labels *Person, Book*, and *Author* are depicted in rectangles over the nodes, (2) a node can possess more than one label, and (3) both nodes and relationships may have attributes that are key-value pairs. The name of the relationship is reflected in bold font over the edges. All the nodes with the same label form a group and this leads to an improvement in the query efficiency because a query involving labels limits the the search space to the group of nodes or relationships defined by that label instead of searching through the complete graph [53, 24]. As an example, consider the label *Book* in Figure 2. In this case, when a query is specified to list all books, then only the nodes labelled as *Book* are traversed.

This concludes our brief discussion of the concepts that form the foundation of our research. In the next section, we present our approach and proof-of-concept for transforming relational and RDF databases into a property graph model.

3 Transformation Approach

We leverage the graph transformations proposed by Bouhali et al. [52]. Bouhali et al. focus on converting RDF data into a graph model whereas we envision an

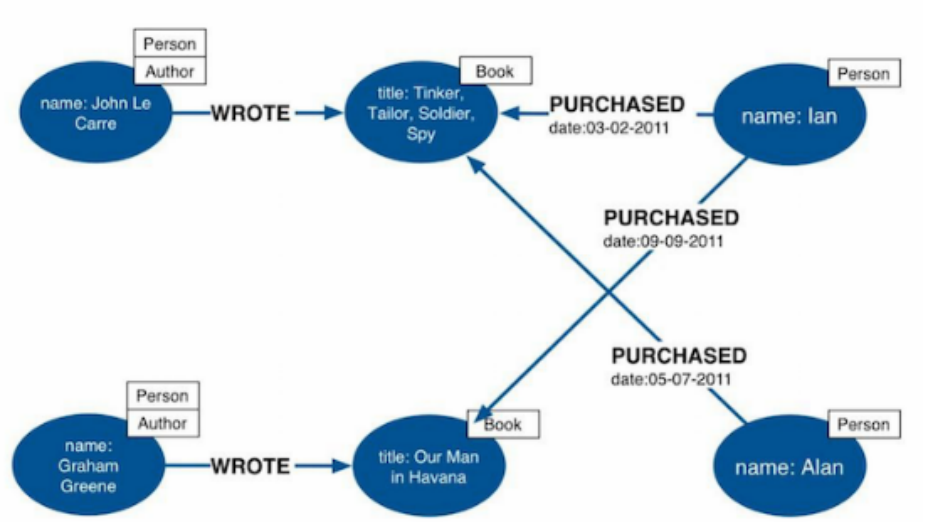

Fig. 2. An example of a property graph [24]

extensible approach that embraces model diversity by allowing multiple models such as relational, RDF, and column-family stores from NoSQL databases, for example, all under one framework. Two issues arise: (1) transformation rules to map native model concepts to the property graph model, and (2) assurance that the individual concepts of native models do not get lost even after all the models are transformed into a graph representation. One of the unique characteristics of our proposed model is its native concept-preserving characteristic. This native concept-preserving characteristic is instrumental in facilitating reverse engineering when the graph representation would need to be expressed in the original model terminology. For example, to publish the data for use in a linked data project, RDF would be the model of choice. Thus, data that is represented using any other format would need to be transformed to the RDF model. Furthermore, in the scenario where the integrated, transformed graph includes information from multiple models, having knowledge about which nodes are originating from a particular model offers an independent view of the data models in use. This lays a foundation for model-specific data extraction or transformation.

We now present transformation rules and proof-of-concept of our graph model representation by considering schema excerpts for relational and RDF schemas. For convenience, we use the general term *schema* throughout the paper to refer to both schema and data when discussing models that have a close coupling of the two. For example, mapping a data source to a Neo4j graph database involves mapping structure and instances, but we refer to this activity as *schema mapping*. Similarly for merging two graph databases, we refer to this process as *schema integration*. We intend for the meaning to be clear from the context provided by the discussion and examples.

3.1 Relational Model to a Property Graph

We begin by addressing the conversion of a relational schema into a property graph representation. Das et al. [48] have proposed a methodology for converting relational databases to RDF. Given that Bouhali et al. [52] have proposed an algorithm for transforming RDF to graph, it would appear that we can combine these two proposals [48, 52] to transform a relational database to a property graph. However, we do not follow this approach in our work for the following two reasons. First, the final property graph model obtained by Buohali et al. [52] does not reflect the native model features. In their work, they focus on transforming RDF to a graph model and thus graph nodes implicitly correspond to the RDF schema. Our work applies to multiple models, not only RDF, and in anticipation of the need for reverse engineering from property graphs to native models, it is important to preserve the identity of native models in the graph representation. Second, developing transformations from the relational model to the property graph model offers a direct route to the target format (property graph in our case) instead of creating an RDF representation as an intermediate step. Table 1 represents a sample relational schema that we consider for illustrative purposes. The schema excerpt describes two entities, *employee* and *organization*, and the relationship between them using a referential integrity constraint on the attribute *orgId*. We present three transformation rules as follows.

Rule 1: Every tuple in the relation is transformed to a node in the property graph. The node is labelled *RelationalResource* and defines a property in the form of name-value pair as type: *Name of the Relation*. The label serves to disambiguate the relational source from the other models that would also be transformed into a graph representation.

Rule 2: For each of the attributes in the relations, a property (name/value pair) is added to the corresponding node in the graph. This node would be the one that has the value for the property type equal to the name of the relation.

Rule 3: For each foreign key, a relationship is created between the nodes corresponding to the two participating relations.

The dashed rectangle on the left in Figure 3 illustrate the relational schema from Table 1 as a graph model in Neo4j based on the transformation rules above. The black box at the bottom left shows the properties (name-value pairs) for the employee, "Jason Doe" from the relational schema.

3.2 RDF Model to a Property Graph

We now focus on the RDF schema. Figure 4 presents RDF data based on a schema excerpt from FOAF (Friend Of A Friend) [27] RDF vocabulary. An RDF vocabulary is an RDF schema formed of specific set of classes and properties that define resources of a particular domain. The example in Figure 4 describes two entities *(Person* and *Group)*, the classes that are used to describe them (*foaf:Person* and *foaf:Group*) and the properties that relate them (*foaf:member*). The properties such as *foaf:name* and *foaf:homepage* are applied to a *Person* entity and their values are either literals or resources described using

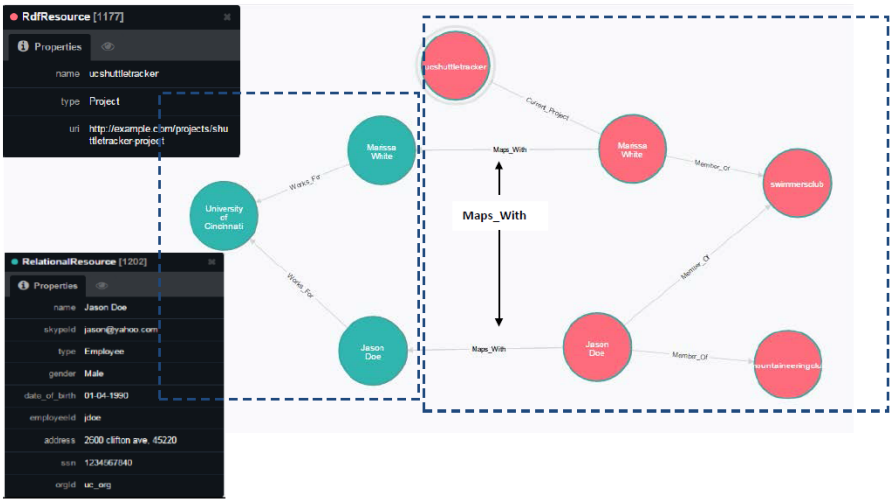

Fig. 3. Excerpts of two heterogeneous schemas originally in different models (left: relational schema, right: RDF schema) unified under a common graph representation

URI (Uniform Resource Identifier). We show only one individual's information in Figure 4 to save space, but the graph in Figure 3 shows two individuals information (Marissa White and Jason Doe), their group memberships, and their organization. The black box at the top left shows the properties (key-value pairs) corresponding to the node identified by the name *ucshuttletracker* and of type *Project* in the RDF schema.

```
<foaf:Group>
 <foaf:name>Swimmersclub</foaf:name>
 <foaf:member>
  <foaf:Person>
        <foaf:name>Marissa White</foaf:name>
        <foaf:homepage rdf:resource="http://homepages.uc.edu/" />
        <foaf:birthday>04-01-1983</foaf:birthday>
        <foaf:gender>Female</foaf:gender>
        <foaf:skypeId>marissa@yahoo.com</foaf:gender>
        <foaf:currentProject rdf:resource="http://www.example.com/projects/ucshuttletracker"/>
  </foaf:Person>
 </foaf:member>
</foaf:Group>
```

Fig. 4. RDF schema excerpt

With the two input schemas transformed to a graph representation, the next natural question to ask is: *What is the additional merit that the common graph representation offers compared to the knowledge that could have been derived from the native model representations*? Figure 3 shows both the relational schema and RDF schema connected by mappings *(Maps_With)* in a graph model. This provides an insight into the question. Figure 3 highlights how one can obtain more details for an employee if these two separate schemas can be integrated,

compared to the information that we originally received from isolated sources. Relating *Employee* and *Person* nodes, we can identify his or her details such as name and gender and also information on the groups that he or she is a member of, or the homepage. Notice that the information on the homepage of a person is only captured by the RDF schema in our example. By unifying them based on common attributes such as *date of birth* or *skypeId*, an application can benefit from incorporating information from both schemas. This additional information may be harnessed by an organization to develop community-outreach programs based on employee outside interests, for example. Graph models represent a solution for depicting a connected environment. We employ the Neo4j Cypher query language to create mappings *(Maps_With)* between the appropriate nodes by comparing values of certain attributes. In Figure 3, we link the employee and person information coming from relational and RDF schemas, respectively, based on *skypeId*. In a general context, the example helps to illustrate a use case for leveraging a graph-based model towards a common representation scheme for model and schema diversity.

Apart from facilitating an integrated view, another benefit of our approach is that it preserves the native model concepts in the transformed graph model while providing a uniform representation at the same time. By augmenting nodes with the labels (such as *RelationalResource* and *RDFResource*), one can easily identify information that was originally expressed in a particular native model. At the same time, bringing the individual model and schema concepts under an umbrella of common terms (nodes and relationships) facilitates linking and querying them using a single query language.

In a blog post, the Neo4j developer team present an approach to transform a column-oriented data model to a property graph [50]. The goal is to allow loading of data from a Cassandra data store into Neo4j. The mapping between the source (column-oriented) and target (property graph) data models is taken as input from the user and the resulting graph is created by loading the data from Cassandra to a CSV file. Neo4j supports batch creation of a graph from CSV format. Figure 5 shows a sample schema in Cassandra with *p, r,* and *u* as inputs from the user for schema mapping. The label *p* stands for a property, *r* for a relationship, and *u* for specifying unique constraint field. Since our work incorporates schemas originally expressed in multiple heterogeneous models, we can incorporate the Cassandra to Neo4j mapping by labelling the nodes as *Cassandra Tables*. The approach presented in [50] focuses only on the mapping between one set of source and target data models. In the next section, we present the architecture of our approach.

4 Architecture

Our framework for transforming schemas to a graph-based format can be broken down into three main modules:

1. *The database module* holds schemas and exports a database to CSV format to support the automation step in the application module.

```
CREATE TABLE playlist.artists_by_first_letter:
    first_letter text: {p}
    artist text: {r}
    PRIMARY KEY (first_letter {p}, artist {u})
CREATE TABLE playlist.track_by_id:
    track_id uuid PRIMARY KEY: {u}
    artist text: {r}
    genre text: {p}
    music_file text: {p}
    track text: {p}
    track_length_in_seconds int: {p}
```

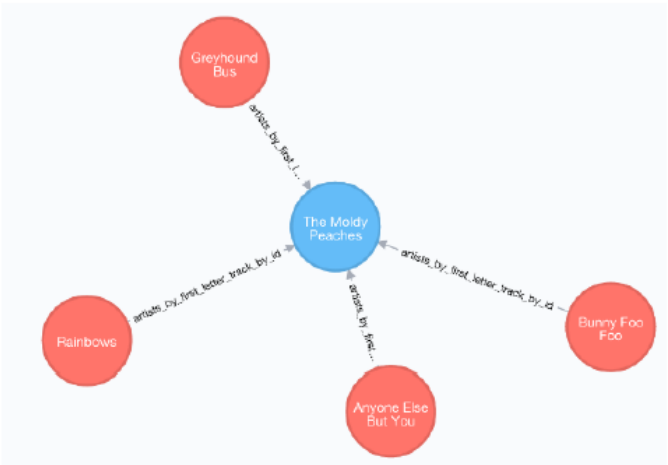

Fig. 5. Placeholders *p,r,* and *u* for schema mapping between a column-oriented store and a property graph model [50]

2. *The application module* offers a presentation layer where a user can select the schema that needs to be transformed to Neo4j, and to allow transformation in a systematic manner. The software implementation in this module employs our transformation rules.
3. *The graph module* uses the Neo4j browser to view the transformed schemas.

Figure 6 presents the architecture of our approach.

We use MySQL as the backend database for relational schemas. The process starts at the database layer which consists of three components: schemas, user-defined stored procedures and the MySQL native export tools. Schemas capture a built-in MySQL database *(information_schema)* and any user defined relational schemas. These user defined schemas are the artifacts that will be transformed to a graph model. The stored procedure reads metadata information from the *information_schema* and identifies all the foreign key relationships in our schema of interest. Figure 7 illustrates a query in our stored procedure to capture all

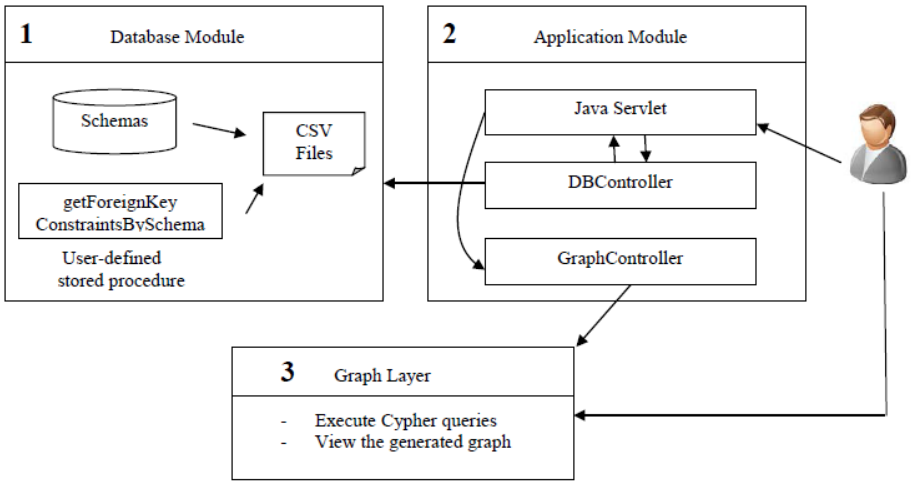

Fig. 6. Architecture of our proposed approach

referential integrity constraints. The reason for collecting all the foreign keys in our database is that we need them to create relationships in our graph model based on the transformation rules from Section 3.

The user first exports the data in each of the relations in the database as a CSV file using MySQL native export data tool. We use the CSV file format since both the Neo4j community and our programming interface which uses Java support CSV files. Furthermore, Neo4j allows batch creation of a graph from CSV format.

```
select
    concat(table_name, '.', column_name) as 'foreign key',
    concat(referenced_table_name, '.', referenced_column_name) as 'references',
    constraint_name as 'constraint name'
from information_schema.key_column_usage
where referenced_table_name is not null
    and table_schema = 'sakila'
```

Fig. 7. MySQL query to capture foreign key relationships in the MySQL sakila database [26]

At the application level we use Java to interact with the database and the graph modules programmatically. A database controller (DBController) manages connection to the database module and a graph management controller (GraphController) handles connection to Neo4j and submits queries to the graph interface through Java. These three application-level components working together along with our transformation rules from Section 3 facilitate automated transformation of a relational schema into a Neo4j property graph.

```
Class.forName("org.neo4j.jdbc.Driver");
Connection con = DriverManager.getConnection("jdbc:neo4j://localhost:7474/");
Statement stmt = con.createStatement();
ResultSet nodes = stmt.executeQuery("LOAD CSV WITH HEADERS FROM \
"file:C:/Users/usplib/ "+tablename+".csv\" AS line
MERGE (m:RelationalResource {id: line.id, type:'"+tablename+"'})
ON CREATE SET m+= line");
```

Fig. 8. Code-snippet illustrating creation of a Neo4j graph for a relational schema programmatically

Figure 8 presents a code snippet that reads a relational schema exported to CSV format and converts it to Neo4j graph. The code focuses on generating nodes which represent the concept of relations in a relational database. Each relation in the schema is exported to a CSV file with the same name as the relation itself. The code reads each of those CSV files and generates nodes with label *RelationalResource*. The Create Set clause in Figure 8 adds properties to those nodes. Each field in the CSV file represents one property.

We now have an understanding of the architecture and the input artifacts that are required for software implementation. Our approach is extensible and only requires transformation rules to be defined between any additional models and the property graph.

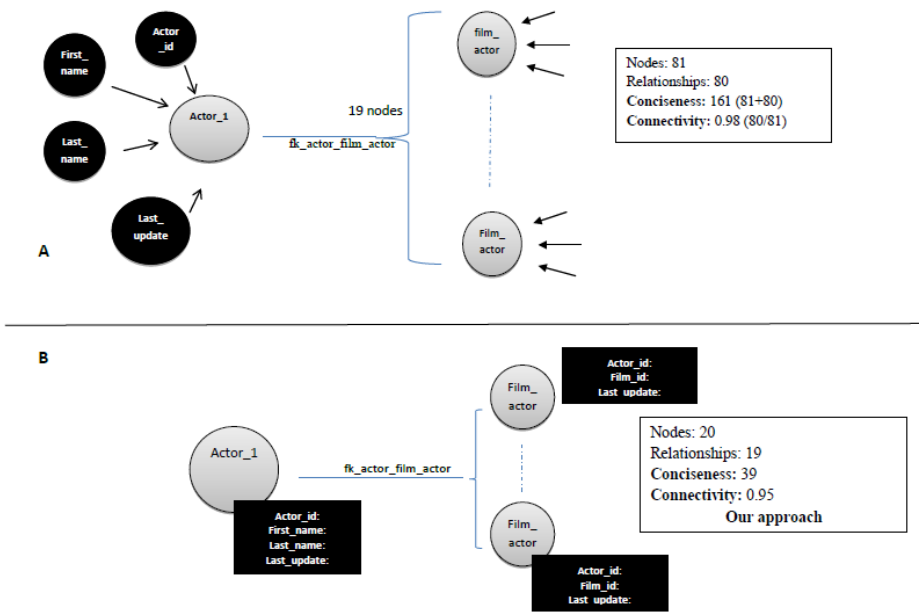

Fig. 9. (a)(top) Modeling attributes as nodes, (b) (bottom) Our approach to modeling attributes as key-value pairs

5 Evaluation

In this section, we discuss both the qualitative and quantitative analysis of our mapping approach. We leverage the evaluation metrics proposed by Buohali et al. [52] and also discuss qualitative merits of our proposal.

The quantitative evaluation metrics we consider are *conciseness* and *connectivity* of the graph. Conciseness is given by the total number of nodes and relationships and can be used to calculate the graph size. Connectivity is calculated by dividing the number of relationships with the total number of nodes. We apply these measures on the generated graph for an open source MySQL database, sakila, in Table 2.

Table 2. Evaluation metrics results for MySQL database - sakila [26]

Total nodes	47273	Conciseness	62682
Total relationships	15409	Connectivity	0.32

Buohali et al. [52] state that for efficient processing over a graph, connectivity should be at least 1.5, which would signify strong connections in the graph. The connectivity value for our graph is quite low from their benchmark perspective. However, on further investigation, we identify why a low value may not always signify a non-desirable characteristic.

First, according to our transformation rules (Rule 3 in Section 3), the only relationship between two nodes that occurs in the target graph model comes from a foreign key relationship in the relational model. This sets the range for the number of relationships between two nodes for a particular constraint to be 0 to max(*n1, n2*) where *n1* and *n2* correspond to the number of each of the two node types. Thus, based on our transformation rules, the number of relationship instances for a particular relationship type (in our case, a foreign key constraint) cannot exceed the number of nodes and hence the connectivity cannot exceed 1. The reason we have an even lower number is that some relations such as *film_text* are not even linked to other relations in the schema.

From this investigation, we come to the conclusion that strong connectivity between nodes in a graph certainly is good for processing but it also does not automatically lead to the conclusion that a lower number is not desirable. The two metrics of conciseness and connectivity can also offer some ideas when we need to make a choice among multiple solutions. A graph with high connectivity is good for processing but if it comes at a price of increasing the graph size (less concise), then this would also lead to an increase in the cost of traversal because of increased path lengths. A larger graph size also implies higher storage requirements.

We evaluated this trade-off between conciseness and connectivity using an alternate mapping and the results are shown in Table 2. For the alternate mapping, we considered modeling attributes as nodes instead of properties (key-value pairs). This resulted in an increase in the number of nodes as well as relationships. The additional relationships come from the new edges created between

attributes and entity nodes. We take a small example from sakila database to illustrate the two different mappings and their impact on conciseness and connectivity. Figure 8 shows two graph representations corresponding to two different mappings.

Figure 9a shows a property graph model where attributes are modelled as individual nodes. The conciseness of the graph (which is captured by total number of nodes and relationships) is 161 and connectivity equals 0.98. Our approach shown in Figure 9b shows how conciseness is increased based on our proposed set of mappings which model attributes as node properties and not as separate nodes. The connectivity does not show much difference and this is because of the nature of the native model and the types of relationships it exhibits. Edges in the graph are generated by foreign key constraints in the relational model.

Figure 9b represents the graph model based on our transformation rules from Section 3; Figure 9a captures an alternate mapping where emphasis is placed on increasing the graph connectivity. We modeled the attributes of a relation as separate nodes and created additional relationships between each of those attributes *(actor_id, first_name, last_name, and last_update)* and the relation node *(actor_1)*. The node labelled *actor_1* represents the data tuple from the actor relation that has *actor_id* equal to one. Similarly, the node *film_actor* represents the relation *film_actor* in the sakila database. The actor node has foreign key relationships with 19 *film_actor* nodes. Based on the *sakila* database, this signifies that the particular actor has acted in 19 films.

Table 3 captures the evaluation metrics from both approaches.

Table 3. Evaluation metrics results for MySQL database - sakila using two mapping transformations

Evaluation criteria	Our approach	Alternate mapping (Figure 9a)
Total nodes	47273	62967
Total relationships	15409	66239
Conciseness	62682	129206
Connectivity	0.32	1.05

The results from Table 3 and the example from Figure 9 illustrate two key ideas: (1) the connectivity depends on the nature of original model, and (2) a higher connectivity may come at the cost of an increase in the graph size. Qualitatively, the merit of our proposal for schema mapping lies in the integration between multiple, heterogeneous models under a common graph framework. The *Maps_With* relationship as shown in Figure 3 creates many additional relationships which were not even present when the transformed graph models of relational and RDF schema were studied separately.

6 Schema Integration over Property Graphs

In this section we propose a framework for schema integration over property graphs. We consider two input schemas expressed as a property graphs and

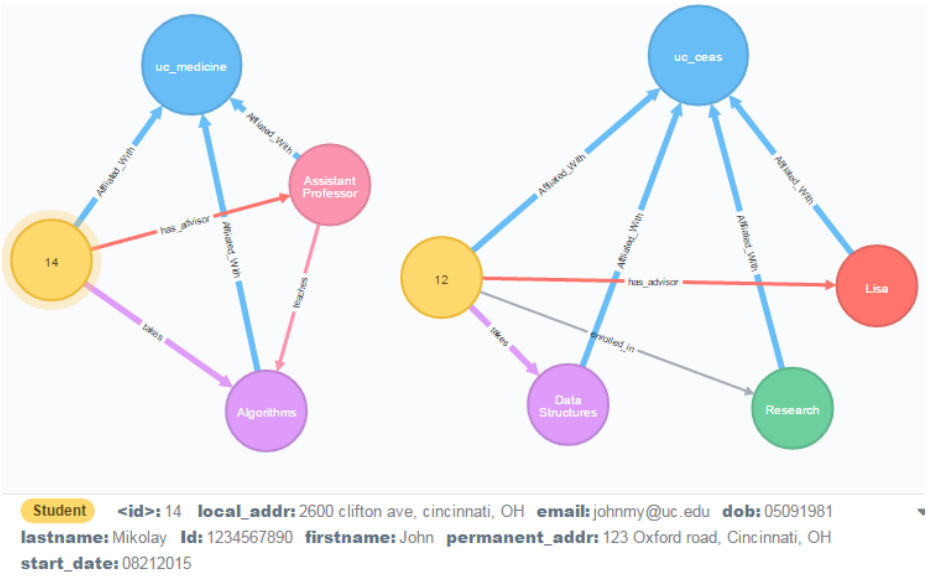

Fig. 10. Two schemas (left: Medicine Graph, right: Engineering Graph) modelled using property graphs

define an algorithm to integrate the two graphs to generate a final integrated schema. The foundation for mapping heterogeneous models to a property graph is established in Section 4.

There have been significant research and industry implementations that offer solutions towards schema integration [11–16, 23, **?**]. Solutions exist in the areas of specifying or semi-automatically identifying schema mappings which serve as a foundational step in schema integration. The contribution of our proposal lies in providing an infrastructure that leverages existing mapping algorithms but over a new modelling paradigm, the property graph.

We discuss our schema integration approach using two property graphs shown in Figure 10. The schemas in both the graphs capture information about entities *student*, *faculty*, *courses*, and *department* and their relationships. The two schemas correspond to two different departments: *uc_ceas* and *uc_medicine* shown as the topmost node in Figure 10.

While there are several commonalities as a result of the common domain, the schemas also exhibit differences creating challenges in schema integration. The differences and our proposed solutions for addressing each of them are as follows:

1. Relationship semantics: The property graph *uc_medicine* allows for a student to have one advisor, if he or she has one (it is optional) while the graph *uc_ceas* requires a student to have an advisor. A student may not be under a research program and thus may not even have an advisor according to the *uc_medicine* schema. We model these min/max constraints using properties

(participation and *cardinality)* over relationships in our property graphs. As a solution to address this scenario, in the final integrated schema, we impose a min constraint of 0 and max constraint of 1 for the *has_advisor* relationship. The idea behind picking this set of participation and cardinality constraints is that it leads to information preservation of the two native schemas. Following the *uc_medicine* schema in Figure 10, if a student does not have any advisor, our integrated schema would allow that while also allowing a student to have an advisor if that happens to be the case in the sample schemas.

2. Non-overlapping attributes in two similar entities across the two schemas: this addresses the scenario where an entity in one schema has certain attributes which are not present in the similar entity in the other schema. As an example, the *Faculty Member* node in the *uc_ceas* schema does not store information about faculty's *start_date* and *department* while the *uc_medicine* schema does. This scenario can even be extended to entities and relationships such that one schema may be capturing additional information about a domain that is not covered in the other schema. As an example, consider the entity *ResearchCredits* in the *uc_ceas*. It is not present in the schema for college of medicine. Our approach adds each of the unique attributes from each of the two schemas to the final integrated schema.
3. Differences in constraints: this difference may occur where the properties from two similar entities in the two schemas have different data type or uniqueness constraints [49]. A uniqueness constraint on a property ensures that no two nodes in the graph hold the same value for that property. As an example of the differences, *studentId* for a student entity in *uc_ceas* allows string values while the data type for the corresponding property *Id* in the *uc_medicine* only allows integers. Apart from the difference in data types, the uniqueness constraint for one schema may be composite (consisting of multiple properties) while the other schema may be defining a single-attribute constraint.
   There are multiple ways to resolve this scenario. As an example, if the difference is in terms of data types, then the data type with a wider range of values (String over integer) can be considered for the final integrated schema. However, consider another scenario where the difference is also in terms of number of attributes representing the uniqueness constraint. One schema may have the constraint defined on a set of attributes (composite) while the other schema uses a single attribute constraint. Considering the possibilities of multiple ways in which differences in the constraints can manifest, we consider the idea of adopting a surrogate key in the final merged schema. The original constraints on the attributes from each of the two schemas will also be copied to the merged schema to prevent any information loss.
4. Difference in field names/entities: entities or attributes across the two schemas may pertain to the same concept but use different names and terminologies. Iin the example property graphs, the terms *lname* and *lastname* for *Student* node use different terms for the same concept.
   The literature offers numerous solutions for identifying and resolving such conflicts using lexicon, ontology, or string algorithms [17–20].

The final integrated schema addressing the four heterogeneity scenarios above is obtained using the algorithm described in the next section.

7 Algorithm

The core algorithm can be summarized as follows. Start with one input graph as the base graph. Merge the second into the graph based on a likelihood match for each new node against the base graph. The match between two nodes is determined based on the four solutions described in Section 6, algorithms from the literature and a user-defined threshold value. If there is no good match, the node is not merged but added as a new node to the integrated graph. Figures 11-13 presents our algorithm as three main modules - *determineNodeTypesForMergedSchema, mergeNodes, and mergeRelationships.*

```
Input: Two property graphs, G1 = (N1, E1, NL1, EL1) and G2 = (N2, E2, NL2, EL2)
       Ni, Ei, NLi, Eli correspond to the nodes, edges, node-labels and edge-labels in the graph
       match_threshold = <a number between 0 and 1>
Output: Merged schema graph: G = (N, E, NL, EL) where
                                N = N1 U N2
                                E = E1 U E2
                                NL = NL1 U NL2
                                EL = EL1 U EL2
       Initialize G: Empty
1: function determineNodeTypesForMergedSchema(G1, G2)
       1.1 for all lbl in NL1(G1) do
             NL(G) ← lbl        //Add each unique label from graph G1 to merged graph G
           end for
       1.2 for all l in NL2(G2) do
       //Determine if the label l matches any of the existing labels in the integrated schema
             1.2.1 match_result ← isMatch(l, NL(G), match_threshold, false)
             1.2.2 if (match_result)
                     a) matched_node ← isMatch(l, NL(G), match_threshold, true)
                     b) map_labels.add (l, matched_node)
                   else                                          //no match found
                     NL(G) ← l
                   endif
           end for
   end function
```

Fig. 11. Module for unifying the node-types from input schemas

The input schemas modelled using property graphs *(G_1 and G_2)* are represented using a four-element tuple. The four elements are sets of nodes *(N)*, edges *(E)*, node-labels *(NL)*, and edge-labels *(EL)* in the graph. The notation *NL(G)* and $NL_1(G_1)$ refers to node-labels in the property graph G and G_1 respectively.

```
2. function mergeNodes(G1, G2)
    2.1 for all n in N1(G1) do
      2.1.1 If node's label (nl) matches one of the labels in the merged schema labels:
            //Create a node in the merged schema with label and properties
                N(G) ← n
          else
            //Find the mapping of the node's label to the set of labels for merged schema
              a) mapped_label ← map_labels.find(nl)
              b) Create a node in the merged schema with label mapped_label and properties
          endif
      2.1.2 Add n to set of nodes in the merged schema
                N(G) ← n
        end for
    2.2 for all n in N2(G2) do
      2.2.1 If node's label (nl) matches one of the labels in the merged schema labels:
          a) Identify a node in the merged schema with the same label. Call it n_ms
          b) Call mapAttributesForEntity(n, n_ms)
          c) Call addNonOverlappingAttributes(n, n_ms)
          else
          //Find the mapping of the node's label to the set of labels for merged schema
            a) mapped_node_label ← map_labels.find(nl)
            b) if (mapped_node_label is not null)
                  Call mapAttributesForEntity(n, mapped_node_label)
              else
                  Create a new node n and add it to the merged schema
              end if
          end if
      end for
  end function
```

Fig. 12. Module for merging the nodes from input schemas

The output, merged schema is represented as *(N, E, NL, EL)* where each of the set elements is the union of the corresponding elements from the input graphs. The set of nodes N in the merged schema is the union of $N_1(G_1)$ and $N_2(G_2)$. The merged schema is initialized as empty.

The algorithm consists of three main parts: (a) capturing all unique node types from all the input schemas into the final integrated schema, (b) union of nodes N_1 and N_2, and (c) union of relationships E_1 and E_2.

The algorithm proceeds by first identifying the unique node-types across all the input schemas. Each node-type can be considered as an artifact/entity holding a certain set of properties modelled using key-value pairs. Figure 11 shows this module. In our example for schema integration here, we use node labels to capture the node type. The module (Figure 11) copies all the labels from the first input schema to the integrated schema (Step 1.1). Step 1.2 then iterates over each of the labels in the second schema, G_2 to compare it against all the labels in the merged schema so far. If a match is found that meets a threshold value, then the node is merged with the matched node. The mapping

between the matched node from the partial merged schema and the new node is also stored (Step 1.2.2).

The algorithm uses the *isMatch* function which takes four arguments: (a) the node, l , to be searched, (b) set of node labels *NL(G)* in the merged schema, and (c) a threshold value in the range 0 and 1 for matching, and (d) a boolean argument to signify the type of return value. If the fourth argument is true (Step 1.2.1), it returns the matching node, and if false, a boolean value is returned. This return boolean value signifies if the label l exists in the set of partial merged schema labels or not. If no match is found indicating a new node-type, then the label is added to the set of labels of merged schema (else block in step 1.2.2) . The notation *NL(G)* represents set of node-labels for merged schema graph *G*. However, if the node label already exists, then the mapping is stored in a data structure *map_labels* (Step 1.2.2). This function *isMatch* can be customized by applying different schema matching algorithms from the literature.

To understand the rationale behind storing this mapping, refer to our sample input schemas in Figure 10. We have nodes (*Assistant Professor* and *Lisa*) labelled *Faculty* and *Faculty member*. The labels are representing the same entity but using different terms. The final integrated schema consolidates them into a single label *Faculty*. This mapping information is now important to merge nodes of type *Faculty member* in the *uc_medicine* into nodes with type *Faculty* in the merged schema.

The next steps involve adding nodes and relationships in the final integrated schema. Figures 12 and 13 show the modules addressing this functionality.

```
3. function mergeRelationships(G1, G2)
    3.1 for every relationship r in G1 and G2
            3.1.1 Get source node as source
            3.1.2 Get target node as target
            3.1.3 Find the corresponding nodes for source and target nodes in the set of nodes in the merged
                  schema
                      mapped_source = map_node(source, G(N))
                      mapped_target = map_node(target, G(N))
            3.1.4 Create a relationship R in the merged schema with
                       source_node(R) = mapped_source, and
                       target_node(R) = mapped_target, and
                       properties_ms(R) = properties(r)
            3.1.5  Add(G(E), r)
        end for
    end function
end
```

Fig. 13. Module for unifying relationships from input schemas

After identifying the node-types for the integrated graph, the algorithm iterates through every node in the input graphs. It compares node-type (label) with the set of labels in the integrated schema (Step 2.1.1). If an exact match is not found (for example, *Faculty* and *Faculty Member* node-types in Figure 10), the

Fig. 14. Integrated schema

closest mapping is found between the current node's label and the set of labels in the integrated schema (step 2.1.1.a in the else block). A new node is then created with the mapped node-type. The notation *N(G)* refers to set of nodes in the merged property graph *G*. Similarly, the algorithm iterates through each node in the next input graph. If the current node's label matches with one of the node's labels in the partial integrated schema, the functions *mapAttributesForEntity* and *addNonOverlappingAttributes* are invoked.

The final module involves creates edges between nodes in the partial integrated schema. Figure 13 shows this module. The source and target nodes for each relationship in the input graphs is read(Steps 3.1.1 and 3.1.2) and the corresponding nodes in the partial integrated schema are identified (Step 3.1.3). The relationship is then created between the nodes resulting in the final integrated schema (Step 3.1.5).

Figure 14 shows the integrated schema obtained using our algorithm for the sample input property graphs (Figure 10). Some points to note are as follows.

1. The figure shows the integrated schema for the *Student, Faculty, Course* and *ResearchCredits* node types and one relationship *has_advisor*. Note the surrogate key *custom_id* in the *Student* label. The native primary keys of the individual schemas *uc_ceas* and *uc_medicine* are also preserved. Further, note that the schema *uc_medicine* originally employed the term *FacultyMember* instead of *Faculty*. Using string matching algorithms, we consolidated these two labels into one as *Faculty*.
2. The participation and cardinality constraints for *has_advisor* is 0 and 1, respectively, in the final integrated schema. The original min/max constraints in the native schemas were (0,1) and (1,1), respectively.

The framework provided here merges two property graphs, which can be mapped results from heterogeneous source models. The basic features we address for matching nodes and merging them, for example, can be extended with more sophisticated and powerful techniques from the literature. We provaide a modular proof-of-concept for graph merging.

8 Evaluation

In this section, we discuss two case studies to evaluate the effectiveness and coverage of our schema integration algorithm [21, 22]. The first case study refers to a schema integration example by Petermann et al. [57, 58]. The authors provide data models for two heterogeneous systems (enterprise resource planning and customer issue tracking) of a food trading company, and they further employ the models to demonstrate the effectiveness of their graph-based data integration approach [57, 58]. We adopt their data sources to test the effectiveness of our approach and compare our integrated schema with their result [57].

For the second case study, we use the schemas modelled by Batini et al.[56]. Through these case study we discuss the features covered by our integrated schema. Our focus is on providing a framework to illustrate schema integration over property graphs that can be further extended and optimized.

8.1 Example 1

We consider the *FoodBroker* data source presented by Petermann et al. [58]. The model captures two schemas as shown in Figure 15. The result obtained through our algorithm (Figure 16) results in a schema similar to theirs [58]. The *Employee* and *User* nodes are merged along with their attributes into one node *Employee*. Similarly, *Customer* and *Client* entities from the two schemas are consolidated into the *Customer* entity in our integrated schema. In order to highlight schema integration, we constrained sample data in our graph to one instance for each node type. The cardinality constraints (Figure 15) such

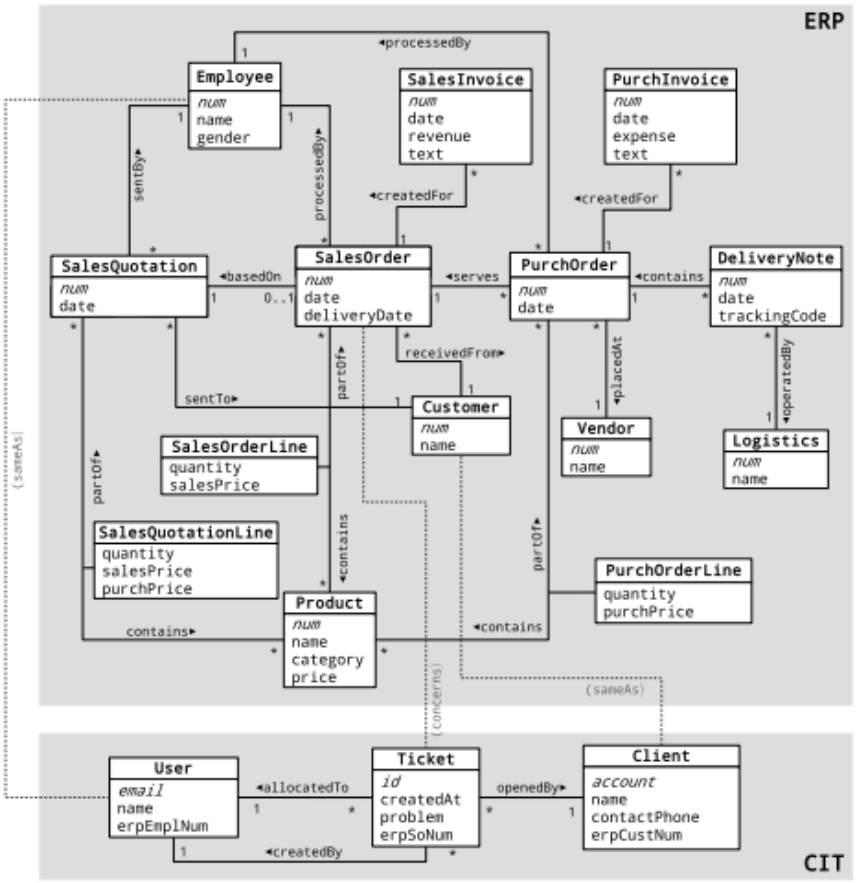

Fig. 15. FoodBroker – Enterprise Resource Planning and Customer Issue Tracking schemas [58]

as one *SalesInvoice* can be created for multiple instances of *SalesOrder* are captured using properties on the relationships in the graph. Once the interschema relationships are determined, the properties of the similar entities are merged.

The added advantage offered by our approach is its ability to handle model diversity. If the native input schemas are in heterogeneous models, the labels of the nodes in their corresponding property graphs (Section 3) can be used to capture the data source or model. This allows the schema administrator to preserve native model information while gaining the benefits of collective information as well that is obtained from the integrated schema.

8.2 Example 2

In this example, we consider schema examples modelled by Batini et al. [56]. The input schemas, *Book* and *Publication*, are shown in Figure 17. The integrated

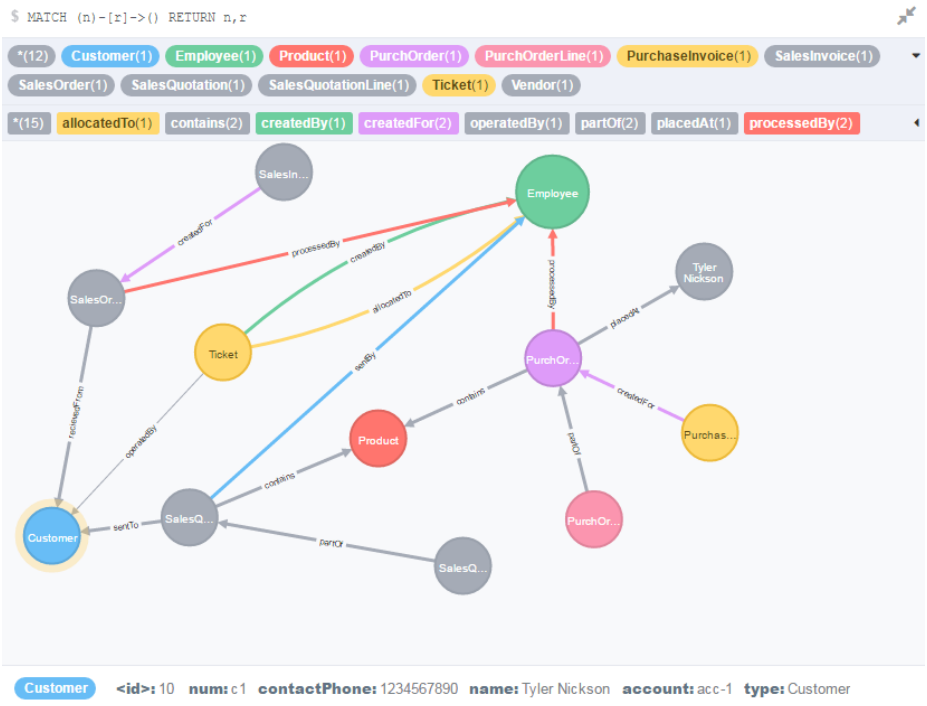

Fig. 16. Integrated schema obtained through our approach (Example 1)

schema obtained through our algorithm is shown in Figure 18. Our integrated schema creates the same entities (*Publisher, Book, University,* and *Topics*). The main features of our schema are as follows:

1. We also capture the the participation and cardinality constraints in our integrated schema (Figure 18). The *Book* schema (Figure 16) originally shows a 1:1 relationship between the entities *Book* and *Topics* while *Publication* schema (Figure 16) exhibits a 1:m relationship between similar entities *Publication* and *Keywords*. In our integrated schema, we resolve this conflict in the cardinalities by modeling the relationship as a 1:m. Batini et al. [56] present and discuss the final integrated schema using a conceptual model (ER). Our approach, based on property graphs, addresses the integration challenge from a graph perspective.
2. We observe that our integrated schema created an additional attribute *Publisher* in the *Book* entity. At this point, our algorithm does not capture the schema conflicts that can arise when some information is modelled as an attribute in one schema and as a relationship in another schema. The information about a book publisher is modelled as a relationship in the *Book* schema and an attribute in the *Publication* schema (Figure 17). Batini et al. [56] identify and cover this conflict. Our framework can be extended to adress this kind of schema heterogeneity.

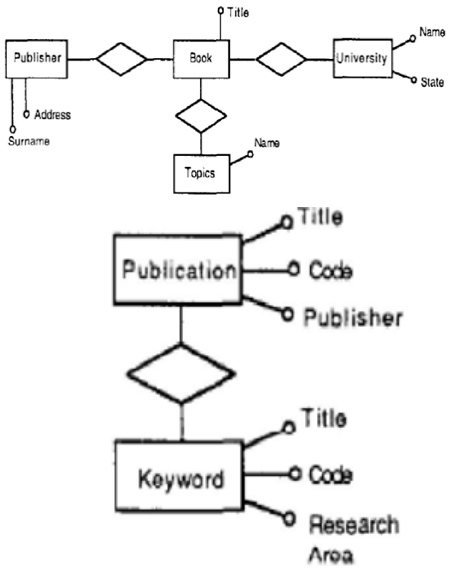

Fig. 17. *Book* and *Publication* schemas [56]

3. Batini et al. [56] also show *Book* entity as a subset of the *Publication* entity, considering that publication can also include journals in addition to books. Our work does not yet address identifying and modelling subset relationships. Again, extensions based on the solutions provided in the literature can be incorporated into our framework.

There are numerous opportunities for extending our work to incorporate modules that address a wide variety of heterogeneous features. The main contribution of our work is in providing a framework that employs property graphs as the representation model. Using graph databases allows addressing schema and data integration using one modeling paradigm.

9 Related Work

Data integration and exchange has received significant research attention by both academia and industry practitioners for more than two decades. One of the frequent approaches is defining an intermediate, canonical model that can capture the commonalities and differences of individual, heterogeneous models, thus providing a uniform representation [5–10, 57]. Relational, XML, and RDF represent some examples of data models that have been considered for this purpose. We identify two key points that may be raised toward our choice of graph model and discuss each of them below. A discussion of these points highlights the rationale and novelty of our solution.

– *Semantic web technologies facilitate integration by establishing links*

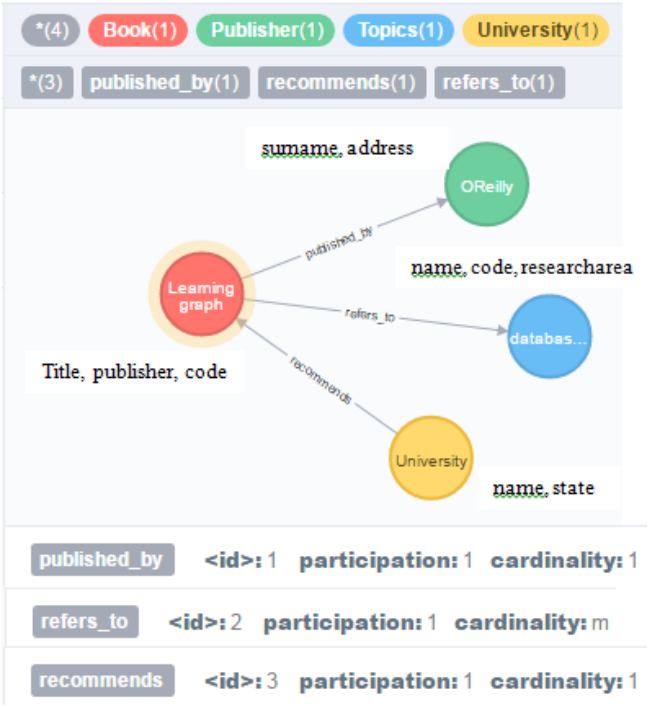

Fig. 18. Integrated schema using our approach (Example 2)

Our idea of employing a property graph model comes from recognizing the data management revolution brought on by big data. In terms of databases, a new class of storage models have emerged called NoSQL databases and graph databases represent one of the categories of the NoSQL family. In this context, we speculate that it would be useful to have frameworks that would allow transformation of different data formats into a model that is amenable to the big data management challenges and our approach represents an effort in this direction. We recognize the immense potential offered by the semantic web research community towards facilitating integration [31–33, 1]. RDF model based on subject-object-predicate framework represents the de-facto standard in the linked data and semantic web community and it already leverages a graph model. However, the choice about selecting one model over the other also depends on the problem and domain at hand [52, 30, 31]. Our vision is to offer an interoperable and integration framework in such a way that it not only facilitates integration of heterogeneous modeling concepts in a flexible and extensible manner, but is also native concept-preserving and aligns with the NoSQL family of data models. Our graph model, as a property graph addresses these requirements.

Furthermore, Bouhali et al. [52] have highlighted potential performance gains in executing large-scale queries over NoSQL graph databases as compared to RDF engines. RDF data needs to be loaded into a SPARQL engine for efficient

query performance and authors cite that while contributions have been made towards optimizing query execution in SPARQL but there still remains scope for further improvement in terms of matching up with the performance offered by graph databases for some large-scale queries. Vasilyeva et al. [30] have also compared a graph model with an RDF store and they conclude that since RDF stores both data and metadata using the same format, it requires RDFS and OWL to distinguish the schema from the actual data.

– *What is the advantage of graph based approach over dominant and successful models such as the relational model?*

In comparison to relational databases, data modeling using graphs offers performance gains in processing interconnected data. This is achieved by avoiding the join operation required in the relational model [25, 2]. Furthermore, the relational model requires a schema to be defined whereas graph models are flexible [3]. Our work towards schema mapping (Section 3) closely aligns with Buohali et al. [52]. Buohali et al. [52] consider translation from RDF to property graphs only. Our application has a broader scope. We build upon their work and extend the scope by making the approach flexible to allow incorporation of additional models.

In terms of schema integration over graph databases, Petermann et al. [57, 58] present a system for graph integration and analytics. The system is based on a property graph model and provides three types of graphs: unified metadata graph (UMG), integrated instance graph (IIG) and business transaction graph (BTG). For generating the metadata graph, their approach extracts the schema of objects to translate it into the property graph model. While our contribution on schema integration is closely related to the focus of their work (generating metadata graph for integration), we also include preservation of native model concepts while handling schema mapping. In case of the need for integration over schemas originally expressed in heterogeneous models, our mapping approach (Section 3) supports annotation of nodes with labels that define the native data model of the schema elements.

The motivation for employing graph based approach also comes from the fact that graph databases belong to the family of NoSQL data stores. Thus, our framework and algorithm for schema mapping and integration over property graphs lay a foundation for integration of schema-based and schema-less data stores.

10 Conclusion and Future Work

We advocate the idea of employing graph databases as a means of bridging the gap between schema-based and schema-less data stores. Our initial results for schema mapping present a proof-of-concept by illustrating transformation of relational and RDF schemas as the first step. We believe that our approach lays a foundation for addressing the variety aspect of big data and bringing traditional data into a big data environment. The second contribution of our work lies in

presenting a schema merging algorithm over property graphs. In this paper, we present a proof-of-concept to illustrate the proposed algorithm. Our approach offers a framework that can be further optimized and it is flexible to incorporate additional schema integration scenarios.

We have translated some traditional models to a property graph. We envision extending our work by incorporating additional data stores. Once we have that achieved that we can incorporate an evaluation study of the transformation process to address the efficiency of the approach. A performance study of querying an integrated graph schema versus disconnected original native schemas is another research direction. The idea of reverse engineering the graph model to obtain the schemas in the original models can also be useful [52, 4] to leverage tools from the native data environments.

References

1. Bizer, C., Heath, T., Berners-Lee, T.: Linked data-the story so far, Semantic Services, Interoperability and Web Applications: Emerging Concepts, 205–227, (2009)
2. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: a data provenance perspective, Proceedings of the 48th annual Southeast regional conference, 42, (2010), ACM
3. Miller, J.J.: Graph database applications and concepts with Neo4j, Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA, Vol. 2324, (2013)
4. Ruiz, D.S., Morales, S.F., Molina, J.G.:, Inferring Versioned Schemas from NoSQL Databases and Its Applications, Conceptual Modeling, 467–480, (2015), Springer
5. Fillottrani, P. , Keet, C.M.: Conceptual model interoperability: a metamodel-driven approach, Rules on the Web. From Theory to Applications, 52–66, (2014), Springer
6. Bowers, S., Delcambre, L.: On modeling conformance for flexible transformation over data models, Proceedings of the ECAI Workshop on Knowledge Transformation for the Semantic Web, 19–26, (2002)
7. Atzeni, P., Cappellari, P., Bernstein, P. A.: Modelgen: Model independent schema translation, Data Engineering, 2005. ICDE, 1111–1112, (2005),IEEE
8. Bernstein, P.A.: Applying Model Management to Classical Meta Data Problems, CIDR, (2003), 209–220, Citeseer
9. Atzeni, P., Torlone, R.:, MDM: a multiple-data model tool for the management of heterogeneous database schemes, ACM SIGMOD Record, Vol. 26 (2), 528–531, (1997), ACM
10. Bowers, S., Delcambre, L.:, The uni-level description: A uniform framework for representing information in multiple data models, Conceptual Modeling-ER 2003, 45–58, (2003), Springer
11. Sheth, A.P. Larson, J.A., Cornelio, A., Navathe, S.B, A Tool for Integrating Conceptual Schemas and User Views, (ICDE), 176–183, (1988)
12. Bellström, P., Kop, C.:, Schema Quality Improving Tasks in the Schema Integration Process, International Journal on Advances in Intelligent Systems, Vol. 7 (3&4), 468–481, (2014), Citeseer
13. Bernstein, P.A., Madhavan, J., Rahm, E.:, Generic schema matching, ten years later, Proceedings of the VLDB Endowment, Vol. 4 (11), 695–701, (2011)

14. Klímek, J.,Mlỳnková, I.,Nečaskỳ, M.: A framework for XML schema integration via conceptual model, International Conference on Web Information Systems Engineering, 84–97, (2010), Springer
15. Bellahsene, Z., Bonifati, A., Rahm, E.: Schema matching and mapping, Vol. 57, (2011), Springer
16. Janga, P., Davis, K.C.: Schema extraction and integration of heterogeneous XML document collections, International Conference on Model and Data Engineering, 176–187, (2013), Springer
17. Rahm, E.: Towards large-scale schema and ontology matching, Schema matching and mapping,3–27, (2011), Springer
18. Cai, Q., Yates, A.: Large-scale Semantic Parsing via Schema Matching and Lexicon Extension, 423–433, (2013), Citeseer
19. Falconer, S.M., Noy, N.F.: Interactive techniques to support ontology matching, Schema Matching and Mapping, 29–51, (2011), Springer
20. Cheatham, M., Hitzler, P.:, String similarity metrics for ontology alignment, International Semantic Web Conference, 294–309, (2013), Springer
21. Doan, A., Halevy, A.Y.:, Semantic integration research in the database community: A brief survey, AI magazine,Vol. 26(1), 83, (2005)
22. Vaidyanathan, V., A Metamodeling Approach to Merging Data Warehouse Conceptual Schemas,(2008), University of Cincinnati
23. Bernstein, P., Ho, H.:,Model management and schema mappings: theory and practice, In Proceedings of the 33rd international conference on Very large data bases,1439–1440,(2007), VLDB Endowment
24. Property Graph, Available at:http://neo4j.com/developer/graph-database, Accessed: 2016-01-27
25. Robinson, I., Webber, J.,Eifrem, E.:,Graph databases, O'Reilly Media, Inc., (2013)
26. Sakila Sample Database, Available at:https://dev.mysql.com/doc/sakila/en/, Accessed: 2016-03-14
27. FOAF Vocabulary Specification 0.99 (2014), Available at: http://xmlns.com/foaf/spec/, Accessed: 2016-01-27
28. FOAF Vocabulary Specification 0.9, Available at:http://xmlns.com/foaf/spec/20070524.html, Accessed: 2016-01-27
29. Google Trends, Available at: https://www.google.com/trends/, Accessed: 2016-03-18
30. Vasilyeva, E., Thiele, M., Bornhövd, C., Lehner, W.:, Leveraging Flexible Data Management with Graph Databases, First International Workshop on Graph Data Management Experiences and Systems (GRADES), (2013), ISBN: 978-1-4503-2188-4, Article 12, Available at: http://doi.acm.org/10.1145/2484425.2484437, doi = 10.1145/2484425.2484437, ACM
31. Goble, C., Stevens, R.: State of the nation in data integration for bioinformatics, Journal of biomedical informatics, Vol. 41 (5), 687–693, (2008), Elsevier
32. Halevy, A.Y., Ives, Z.G., Mork, P., Tatarinov, I.: Piazza: Data management infrastructure for semantic web applications, Proceedings of the 12th international conference on World Wide Web, 556–567, (2003), (ACM)
33. Halevy, A., Rajaraman, A., Ordille, J.: Data integration: the teenage years, Proceedings of the 32nd international conference on Very large data bases, 9–16, (2006), VLDB Endowment
34. Big Data and Analytics, Available at: https://www.idc.com/prodserv/4Pillars/bigdata, Accessed: 2016-01-27

35. Cloud Platform Storage: Relational vs. Scale-Out, Available at: http://davidchappellopinari.blogspot.com/2009/02/cloud-platform-storage-relational-vs.html, Last Accessed: 2016-03-14
36. Özcan, F., Tatbul, N., Abadi, D.J., Kornacker, M., Mohan, C., Ramasamy, K., Wiener, J.:, Are We Experiencing a Big Data Bubble?, Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD '14), Snowbird, Utah, USA, 1407–1408
37. Chattopadhyay, B., Lin, L., Liu, W., Mittal, Aragonda, P., Lychagina, V., Kwon, Y., Wong, M.: Tenzing A SQL Implementation On The MapReduce Framework, (2011), Proceedings of VLDB, 1318–1327
38. Teradata Aster Analytics, Available at: http://www.teradata.com/Teradata-Aster-SQL-MapReduce, Accessed: 2016-01-27
39. Sherif, S.:, Use SQL-like languages for the MapReduce framework, Avaiable at: http://www.ibm.com/developerworks/library/os-mapreducesql/os-mapreducesql-pdf.pdf, Accessed: 2016-01-27
40. SQL-on-Hadoop, Landscape and Considerations, Available at: https://www.mapr.com/why-hadoop/sql-hadoop/sql-hadoop-details, Accessed: 2016-01-27
41. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Zhang, N., Anthony, S., Liu, H., Murthy, R.: Hive - a petabyte scale data warehouse using Hadoop, Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA, 996–1005, (2010)
42. Floratou, A.,Minhas, U.F., Özcan, F.:, SQL-on-Hadoop: Full Circle Back to Shared-nothing Database Architectures, Proceedings of VLDB Endow., Vol. 7, 1295–1306, (2014)
43. Hitzler, P., Janowicz, K.: Linked Data, Big Data, and the 4th Paradigm., Semantic Web, Vol. 4(3), 233–235, (2013)
44. Nejdl, W., Wolpers, M., Capelle, C.:, The RDF schema specification revisited, Workshop Modellierung, (2000)
45. RDF Vocabulary Description Language 1.0: RDF Schema, Available at:https://www.w3.org/2001/sw/RDFCore/Schema/200203/, Accessed: 2016-01-27, (2002)
46. Cypher query language, Available at: http://neo4j.com/developer/cypher-query-language/, Accessed: 2016-01-27
47. Neo4j ranking, Available at: http://db-engines.com/en/ranking/ graph+dbms, Accessed: 2016-01-27
48. Das, S., Sundara, S., Cyganiak, R., R2rml: Rdb to rdf mapping language (W3C recommendation), Available at: https://www.w3.org/TR/r2rml/, Accessed: 2016-01-27, (2012)
49. The Neo4j Java Developer Reference v3.0, Available at: http://neo4j.com/docs/java-reference/current/#transactions-unique-nodes, 2016
50. Lyon, W.: Neo4j + Cassandra: Transferring Data from a Column Store to a Property Graph, Available at: https://neo4j.com/blog/neo4j-cassandra-transfer-data/, 2016
51. Hecht, R., Jablonski, S.: NoSQL evaluation: A use case oriented survey, International Conference on Cloud and Service Computing (CSC), 336-341, (2011)
52. Bouhali, R., Laurent, A.: Exploiting RDF Open Data Using NoSQL Graph Databases, Artificial Intelligence Applications and Innovations: 11th IFIP WG 12.5 International Conference, AIAI, Bayonne, France, September 14-17, 2015, 177–190

53. Neo4j database, Available at: http://neo4j.com/, [Accessed: 2016-01-27]
54. Resource Description Framework (RDF) Schema Specification 1.0, Available at: https://www.w3.org/TR/2000/CR-rdf-schema-20000327/, Accessed: 2016-01-27
55. The RDF Schema Specification Revisited, Available at: https://www.w3.org/TR/2000/CR-rdf-schema-20000327/, Accessed: 2016-01-27
56. Batini, C., Lenzerini, M., Navathe, S.B.:A comparative analysis of methodologies for database schema integration, ACM computing surveys (CSUR) 18.4 (1986): 323-364
57. Petermann, A., Junghanns, M., Mller, R., Rahm, E.:Graph-based data integration and business intelligence with BIIIG. Proceedings of the VLDB Endowment 7, no. 13 (2014): 1577-1580
58. Petermann, A., Junghanns, M., Mller, R., Rahm, E.. FoodBroker-Generating Synthetic Datasets for Graph-Based Business Analytics. In Workshop on Big Data Benchmarks, pp. 145-155. Springer International Publishing, 2014.