REL2GRAPH: AUTOMATED MAPPING FROM RELATIONAL DATABASES TO A UNIFIED PROPERTY KNOWLEDGE GRAPH

A PREPRINT

Ziyu Zhao
School of Physics, Mathematics and Computing
The University of Western Australia
Perth, Australia
ziyu.zhao@research.uwa.edu.au

Wei Liu
School of Physics, Mathematics and Computing
The University of Western Australia
Perth, Australia
wei.liu@uwa.edu.au

Tim French
School of Physics, Mathematics and Computing
The University of Western Australia
Perth, Western Australia
tim.french@uwa.edu.au

Michael Stewart
School of Physics, Mathematics and Computing
The University of Western Australia
Perth, Western Australia
michael.stewart@uwa.edu.au

October 27, 2023

ABSTRACT

Although a few approaches are proposed to convert relational databases to graphs, there is a genuine lack of systematic evaluation across a wider spectrum of databases. Recognising the important issue of query mapping, this paper proposes an approach Rel2Graph, an automatic knowledge graph construction (KGC) approach from an arbitrary number of relational databases. Our approach also supports the mapping of conjunctive SQL queries into pattern-based NoSQL queries. We evaluate our proposed approach on two widely used relational database-oriented datasets: Spider and KaggleDBQA benchmarks for semantic parsing. We employ the execution accuracy (EA) metric to quantify the proportion of results by executing the NoSQL queries on the property knowledge graph we construct that aligns with the results of SQL queries performed on relational databases. Consequently, the counterpart property knowledge graph of benchmarks with high accuracy and integrity can be ensured. The code and data will be publicly available. The code and data are available at github[1].

Keywords Knowledge Graph Construction · Database migration · Query mapping

1 Introduction

A graph is a collection of interconnected triplets, where each triplet is composed of two nodes and an edge connected between them. Both nodes and edges can hold a set of attributes or properties in the form of key-value pairs [Putrama and Martinek, 2022], such as a property graph which extends graphs by adding labels/types and properties for nodes and

[1] https://github.com/nlp-tlp/Rel2Graph

edges [Marton et al., 2017]. Knowledge Graph Construction (KGC)[2] is an active research area aiming at representing knowledge from abundant data in a unified and standardized way. Any triple in a knowledge graph can be annotated with additional facts. It is a complex process to construct a knowledge graph from heterogeneous sources because it requires extracting and integrating information with respect to some uniform semantics. Heterogeneous sources range from unstructured data, such as plain text, to structured data, including tabular data such as CSVs and relational databases, and tree-structured formats, such as JSON and XML. We visualize the KGC process as a trio of paradigms shown in Figure 1. The uppermost paradigm embodies the knowledge extraction from unstructured data through a pipeline that encompasses natural language processing (NLP) tasks including named entity recognition (NER) [Sang and De Meulder, 2003], entity linking [Milne and Witten, 2008], relation extraction [Zelenko et al., 2003] and event extraction [Chen et al., 2015]. The second one converges domain expertise to yield an informed knowledge graph such as Wikidata[3], ConceptNet[4], and Bio2RDF[5]. Notably, the semantic lexicon derived from domain expertise assumes a pivotal role in annotating the extracted triplets, thereby enhancing their contextual significance. The third paradigm is characterized by the transformation of structured data such as tabular data into a knowledge graph. This transformation can be achieved through the Extract, Transform, and Load (ETL) process. An alternative avenue involves semantic table annotation [Abdelmageed and Schindler, 2020].

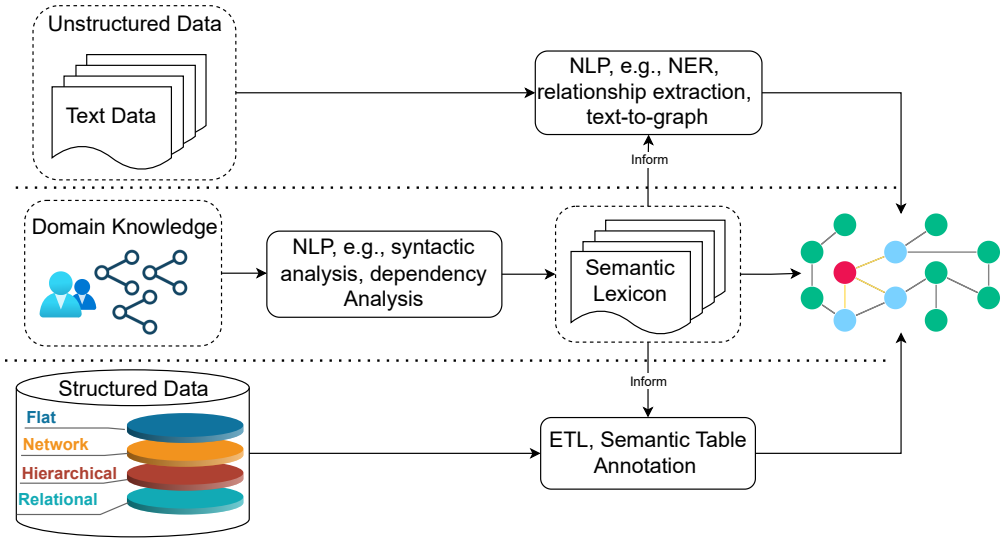


Figure 1: The Taxonomy of Knowledge Graph Construction.

In this paper, we align with the ETL paradigm, harnessing its streamlined mechanism to automatically construct a property knowledge graph in Neo4j[6] from arbitrary relational databases (Section 4.1). A relational database is a structured collection of tabular data. Tabular data refers to data that is organized into a two-dimensional structure with rows and columns, i.e. a table. Each table represents a specific entity or concept, and the relationships between these entities are established through primary keys and foreign key constraints. The resulting knowledge graph from relational databases can be efficiently utilized in a variety of downstream NLP tasks, including semantic parsing [Sorokin, 2021], natural language interface [Zhao et al., 2022], and question answering [Chakraborty et al., 2019]. Among these, semantic parsing plays an essential role and it aims to translate a natural language utterance into a formal query language, such as Text-to-SQL and Text-to-Cypher. Consequently, we also address the challenge of query mapping, specifically focusing on the translation of SQL into Cypher queries (Section 4.2). This undertaking is critical for aligning the application layer, which deploys semantic parsing, with the new persistent layer, ensuring optimal efficiency. In this context, we repurpose well-established semantic parsing benchmarks, originally designed for relational databases, such as Spider [Yu et al., 2018] and KaggleDBQA [Lee et al., 2021], to propel advancements in the retrieval of large graph-based knowledge repositories.

[2]Although knowledge graph is a broader concept that goes beyond a property graph, the techniques attached to KGC are also appliable to a property graph construction.

[3]https://query.wikidata.org/

[4]https://conceptnet.io/

[5]https://bio2rdf.org/sparql

[6]neo4j.com/

With respect to querying relational databases, the challenge becomes particularly pronounced when dealing with intricate inter-table relationships. The inherent complexity arises from the rich relational structure of entities within complex relational databases. This complexity is most evident in the endeavor to craft a target SQL query that encompasses multiple join operations. In contrast, a property graph naturally accommodates interconnected tabular data due to its explicit representation of relationships through edges between nodes. Figure 2 displays our query mapping process[7]. The initial query instance represents a sophisticated SQL query featuring two `JOIN ON` statements. However, once our query mapping rules are applied, the resulting Cypher query that maintains semantic equivalence no longer necessitates resource-intensive join operations. The divergences between SQL and Cypher are discussed in Section 3.

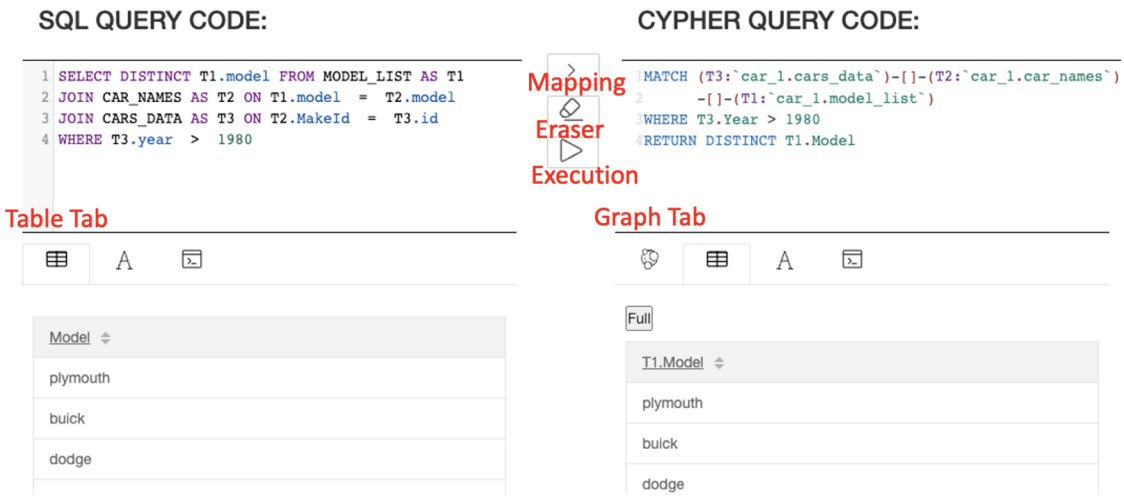


Figure 2: The web-based interface visualises the conjunctive query mapping process of our proposed `Rel2Graph`. It enables users to compose an SQL query and obtain the semantically equivalent Cypher query.

In this endeavor, we introduce an approach `Rel2Graph` to construct a property knowledge graph automatically from relational databases schema dump. The `Rel2Graph` approach also supports the mapping of conjunctive SQL queries into pattern-based Cypher queries. We evaluate our proposed approach using the widely-adopted Spider [Yu et al., 2018] and KaggleDBQA [Lee et al., 2021] benchmarks. Spider is a large-scale dataset. It contains 5,693 unique complex SQL queries on 200 databases with multiple tables, covering 138 different domains. KaggleDBQA is a relatively new cross-domain evaluation dataset of real Web databases. It includes real-world databases from Kaggle[8], a platform for data science competitions and dataset distribution. The relational databases in KaggleDBQA feature abbreviated and obscure column names, domain-specific categorical values, and minimal preprocessing. We employ execution accuracy (EA) [Lin et al., 2020] metric to gauge data consistency at the semantic level. EA quantifies the alignment between the results of SQL queries executed on relational databases and those resulting from graph-based queries performed on the generated property knowledge graph. Consequently, Rel2Graph empowers the utilization of inherent structured data within relational databases for constructing a path-pattern-focused property knowledge graph, highly relevant to semantic parsing task.

The rest of the paper is organised as follows. Section 2 reviews the related work on the KGC process. Section 4 introduces the `Rel2Graph` approach in detail. Section 5 discusses the conversion result and error analysis. Section 6 is the conclusion.

2 Related Work

In this paper, we review the following two paradigms that are used in knowledge graph construction (KGC) tasks, i.e. KGC from unstructured text and KGC from structured data.

2.1 KGC from Text

Knowledge graph construction from unstructured text aims to extract structural information organized in a triplet format. Generally, there are discriminative and generative methodologies for KGC from text data [Ye et al., 2022].

[7]The Web UI is adapted from https://github.com/UNSW-database/SQL2Cypher.
[8]https://www.kaggle.com

Discriminative Methodology The goal is to predict the labels given the input text for each element in a triplet which is then populated into a knowledge graph. For instance, given an input text in maintenance domain *oil leak feed pump*, a NER model identifies the named entity *feed pump* and tags it using `/Item` label, the same as predicting `/Observation` label for *leak*, in order to generate a triplet `</Item, has_observation, /Observation>` [Stewart et al.]. The prediction label set is typically constructed by domain experts. Apart from NER [Chiu and Nichols, 2016], the discriminative models consist of other natural language processing (NLP) tasks, e.g., Entity Linking [Paolini et al., 2021], Relation Extraction (RE) [Wei et al., 2019] and Event Extraction [Chen et al., 2015].

A high-quality annotated text corpus should be ready for the discriminative models. There are a lot of collaborative annotation tools, such as BRAT [Stenetorp et al., 2012], Doccano [Nakayama et al., 2018], Redcoat [Stewart et al., 2019] and QuickGraph [Bikaun et al., 2022], that have been proposed to produce high-quality training datasets for machine learning practitioners. A team of 12 workers has employed Redcoat [Stewart et al., 2019] to carry out collaborative hierarchical entity annotation on maintenance work orders (MWOs), and they use the annotated corpus to train a maintenance NER model. Echidna [Stewart et al.] then uses the pre-trained NER model[9] to detect named entities tagged by the pre-define entity label set as the graph nodes towards constructing a maintenance knowledge graph. During the MWO2KG process, a maintenance asset hierarchy taxonomy has been applied to augment the coarse entity label set to a fine-grained label set, for example, the named entity `pump` (as a graph node) tagged by the coarse label `/Item` is further assigned the fine-grained label set [`/rotating equipment/Item, /Item`] in the sense of enriching maintenance knowledge graph expressiveness. Overall, the discriminative approach to KGC is achieved through the utilization of high-quality training data created by domain experts. However, acquiring high-quality annotated corpora for complex multi-task information extraction is an arduous and costly process for human annotators, and it relies heavily on specialized domain knowledge either stored in dictionaries, gazetteers, or ontology format. Fortunately, there are some available annotation tools, such as QuickGraph [Bikaun et al., 2022], designed to allow domain experts to annotate data for the full pipeline of KGC from text, i.e. NER and RE.

Generative Methodology For the automatic generative methodology to KGC, the goal is to autoregressively generate linearized triplets given the input sentence by fine-tuning sequence-to-sequence (seq2seq) models. For example, BT5 [Agarwal et al., 2020] proposed to use T5 [Raffel et al., 2020] to generate KG in a linearized form. There are a number of proposed research works aiming to generate the graph structure ground up. DualTKB [Dognin et al., 2020] explored GRU [Chung et al., 2014], Transformer [Vaswani et al., 2017] and BERT [Devlin et al., 2018] in an encoder-decoder architecture to enable the text-to-graph translation in both directions using an unsupervised cycle loss, i.e. the back-translations of a sample from a generated triplet back to itself.

The potential issue with seq2seq or end-to-end modeling is that the graph linearisation is not unique and inefficient due to the repetition of graph components multiple times [Melnyk et al., 2022]. GraphRNN [You et al., 2018] decomposes the graph generation process into a sequence of node and edge formations using two RNNs, conditioned on the graph structure. CycleGT [Guo et al., 2020] undertakes text-to-graph in an unsupervised way using an entity extractor followed by a relationship classifier on top of T5. Grapher [Melnyk et al., 2022] is a two-stage KG generation, namely the graph nodes (entities) generation process using T5 and the graph edges generation using the available entity information.

While the idea of formulating KGC from the text in the sequence-to-sequence fashion is free from the constraints of dedicated architectures, expensive labour annotation exercises, and specialized knowledge sources, the sequential and greedy nature of its generation can cause sub-optimal graph structure. It lacks efficient architectures specialized for graph-structured generation output and limited parallel training data. Hence, it is more controllable and efficient to construct large-scale KG through the elaborate discriminative methods in a pipeline strategy to solve a specific task. Nevertheless, the pre-trained language models, such as T5 [Raffel et al., 2020] and BART [Lewis et al., 2019] that store the vast amounts of linguistic knowledge can be exploited for the downstream NLP tasks of domain-specific KG construction from text data in a unified multi-task learning setting.

2.2 KGC from Structured Data

Structured data sources play a fundamental role in the data ecosystem because much of the valuable and reusable information within enterprises and the Web is available as structured data.

ETL Process Extract, transform, and load (ETL) is the process of combining structured data from multiple sources into a unified data warehouse, e.g., a graph database. It originated with the emergence of relational databases to convert transactional databases to relational databases.

[9] https://huggingface.co/nlp-tlp/mwo-ner

Numerous commercial enterprises have developed ETL tools, exemplified by the Neo4j ETL tool[10]. This tool facilitates the importation of data from relational databases into property graph databases hosted within Neo4j. While the Neo4j ETL tool effectively handles basic ETL processes, it may encounter limitations when confronted with exceptionally intricate transformations and tasks involving data cleansing or normalization. In contrast, our proposed `Rel2Graph` approach demonstrates enhanced efficiency in managing these complex tasks. Moreover, the performance of the Neo4j ETL tool may suffer when confronted with substantial datasets or intricate transformation requirements. Additionally, our proposed approach supports customized query mapping, varying from simple queries to complex queries, on top of data migration from large-scale relational databases to property graph databases. This flexibility extends to providing options for data enrichment, encompassing capabilities such as semantic parsing and question answering (KGQA).

For the Semantic Web community, a common strategy involves adopting reference ontologies as global schemas through semantic integration processes that map structured information onto Knowledge Graphs (KGs). Table2Graph [Lee et al., 2015] has been employed to construct graphs based on the Map-Reduce framework over Hadoop, drawing data from three distinct healthcare databases. MDM2G [LACHICHE] has proposed a set of formal rules enabling the conventional star and snowflake schemas to fit in the ETL process from a relational database to a graph database.

In the context of this paper, the most closely related works are those proposed by Putrama et al. [Putrama and Martinek, 2022] and Feng et al. [Feng and Huang, 2022]. The former evaluated the automated graph construction from four relational databases to Neo4j, primarily comparing the total number of records in the source databases to the nodes and edges created in the resulting graph database. On the other hand, the latter evaluated the data consistency using relatively straightforward simple queries but with less efficiency. By contrast, our study undertakes a thorough assessment of automatic database transformation across two distinct benchmarks: Spider and KaggleDBQA. This evaluation encompasses significantly more intricate applications, covering large-scale relational databases. Additionally, it extends to a comprehensive examination of data consistency at the semantic query level.

3 Preliminary - SQL v.s. Cypher

The proliferation of graph databases, driven by the increasing complexity and interconnectedness of data across various domains, has created a pressing need for intuitive and accessible methods to interact with these databases in many real-world scenarios. Property graph databases, for example, Neo4j, Azure DB, OrientDB, and ArangoDB, extend graph databases by adding not only semantic labels to graph nodes and types to graph edges but also properties on both nodes and edges. These additional features enrich a domain data model, providing more context and flexibility to represent real-world applications, such as recommendation systems [Wang et al., 2019] and fraud detection [Pourhabibi et al., 2020]. Property graph databases, in conjunction with declarative query languages, offer a robust solution for storing, retrieving, and analyzing data encompassing intricate graph patterns. This approach proves particularly advantageous when compared to conventional relational databases, as the latter would present challenges in terms of both efficiency and manageability when dealing with such complex graph structures [Szárnyas et al., 2018]. Traditional SQL-based database systems (aka relational databases) have long been the dominant choice for data storage and querying, however, they are not always well-suited for handling highly connected data.

Neo4j, topping the DB engine ranking of GraphDBMS[11], is a popular industry scale non-relational (NoSQL) property graph database. It offers high-level Cypher query language to specify graph patterns supported by the query engine that uses sophisticated optimization techniques [Marton et al., 2017]. SQL uses the syntax with `SELECT`, `FROM`, `JOIN`, `GROUP BY`, and `HAVING` clauses, while Cypher employs a more pattern-based syntax with `MATCH`, `WHERE`, `WITH`, and `RETURN` clauses. Patterns and pattern-matching are the core building blocks of Cypher queries. A pattern describes the data using nodes, relationships between any two nodes, and their properties. It is very similar to how one typically draws the shape of property graph data on a whiteboard, and intuitively it can better model complex relationships and connected data. Patterns appear in multiple places in Cypher, such as `MATCH` and `WHERE` as shown below. Note that, we only describe the basic and frequent usage of Cypher patterns[12].

1. Match a graph node with a specific label using parentheses, e.g., `MATCH (T1:department)`, where `department` is the node label and `T1` is the alias.
2. Match a graph edge with a type between two nodes such as `MATCH (T1:department)-[T2:management]-()`.
3. Match a pattern alongside specifying properties using curly brackets surrounding a number of key-expression pairs, separated by commas, e.g., `MATCH (T1:department)-[T2:management`

[10] https://neo4j.com/labs/etl-tool/

[11] https://db-engines.com/en/ranking

[12] For more details, please refer to Cypher query language reference.

`{temporary_acting:'Yes'}]-()`, where `temporary_acting` is the property of the edge with value 'Yes'. The pattern is to identify the departments managed by heads who are temporarily acting.

4. Using path patterns in `WHERE` by matching on a relationship type using square brackets and either directed or undirected arrows. The symbol `-[]-` means related to, without regard to the type or direction of the relationship, e.g., `WHERE ()-[T2:management]-()`.

4 Automated Approach Overview

Problem Formulation Given a source dataset denoted as $D_R = (\mathcal{S}, Q_{sql})$, we aim to construct $D_G = (\mathcal{G}, Q_{cyp})$. $\mathcal{S}$ represents a collection of relational databases, and $\mathcal{G}$ denotes a corresponding property knowledge graph. Q_{sql} signifies a set of SQL queries and Q_{cyp} denotes the respective set of Cypher queries.

The objective involves a twofold process, i.e. database mapping ($\mathcal{S} \xrightarrow{\text{map}} \mathcal{G}$) and query mapping ($Q_{sql} \xrightarrow{\text{map}} Q_{cyp}$). Firstly, we incrementally construct the property knowledge graph $\mathcal{G}$ from the relational databases in $\mathcal{S}$ using Algorithm 1 (Section 4.1). Secondly, we translate SQL queries to Cypher queries (Section 4.2). To evaluate the quality of our database and query mappings, we assess data consistency. Specifically, we measure how accurately the Cypher queries Q_{cyp} executed against $\mathcal{G}$ align with the source SQL queries Q_{sql} over $\mathcal{S}$. The execution accuracy (EA) metric is employed to quantify this alignment (Section 5).

Algorithm 1: Rel2Graph Pseudocode

Input: A relational database schema dump $s = \langle T, C, V, PK, FK \rangle$.
Output: A property graph database $\mathcal{G} = \langle \nu, \varepsilon, \mathcal{L}, \mathcal{T}, P_v, P_e \rangle$.
RelDB2GraphBuilder(s)
$\text{ST}_i = \langle t_i, c_i, v_i, pk_i, fk_i \rangle \leftarrow \{s\}$; `// queue for each table`
foreach ST_i **do**
  if $fk_i == \varnothing$ *or* $len(fk_i)! = 2$ *or* $(len(fk_i) == 2 \wedge len(pk_i) == 1)$ **then**
    $\mathcal{L} \leftarrow t_i$
    $P_v \leftarrow \langle c_i, v_i \rangle$
    $\nu = \{\mathcal{L}, P_v\}$; `// Case one: entity tables are turned to graph nodes.`
    else if $len(fk_i) == 2 \wedge (len(pk_i)! = 1 \vee pk_i == \varnothing)$ **then**
    $\mathcal{T} \leftarrow t_i$
    $P_e \leftarrow \langle c_i, v_i \rangle$
    $\varepsilon = \{\mathcal{T}, P_e\}$; `// Case two: linking tables become graph edges.`
  $\mathcal{G} \leftarrow \nu \cup \varepsilon \cup \mathcal{L} \cup \mathcal{T} \cup P_v \cup P_e$
foreach $fk_i \in \text{ST}_i$ **do**
  $t_i \leftarrow \text{ST}_i$; `// Get the table name` t_i `w.r.t.` ST_i
  $t_j \leftarrow \text{ST}_j$, WHERE $fk_i == pk_j$; `// Obtain the table name` t_j `corresponding to` ST_j`, where the` $\langle fk_i, pk_j \rangle$ `match the constraint present in the two entity tables.`
  $\varepsilon \leftarrow \{t_i\_HAS\_t_j\}$; `// Case three: curated graph edges between two entity tables.`
  $P_e == \varnothing$
RETURN $\mathcal{G}$

Continuing the notation introduced in Algorithm 1, we symbolize each relational database within $\mathcal{S}$ as $s = \langle T, C, V, PK, FK \rangle$, where the elements of the tuple correspond to collections of table names, column names, row values, primary keys, and foreign keys. We incrementally convert s into the targeted property knowledge graph $\mathcal{G} = \langle \nu, \varepsilon, \mathcal{L}, \mathcal{T}, P_v, P_e \rangle$, where the elements of the tuple represent graph nodes, graph edges, node labels, edge types, and the properties of nodes and edges, respectively. We illustrate the mapping process following.

4.1 Database Mapping

Metadata Analyzer To ensure comprehensive consideration of various table types and their inherent relationships within the database mapping procedure automatically, we recognize the significance of primary and foreign key constraints. This leads us to define the automated examination of these constraints as a metadata analyzer. Each pair of primary and foreign key constraint is represented as $\langle pk, fk \rangle$, where $pk \in PK$ and $fk \in FK$, for the purpose of distinguishing between entity tables and linking tables (or associative entity tables).

To recognize an entity table within a relational database, several key criteria, including primary keys and foreign keys, play a crucial role. Entity tables often exhibit specific characteristics that set them apart. For instance, they commonly possess a primary key designed to uniquely identify each entity or record. When a table features a primary key composed of a single column, it becomes a strong candidate for representing an entity. Additionally, entity tables tend to be less reliant on relationships with other tables. This is evident when a table lacks foreign keys, which can indicate its status as an entity table. Besides, entity tables tend to have one-to-many or one-to-one relationships with other tables. Tables that participate in many-to-many relationships are less likely to be entity tables. Apart from primary keys, foreign keys, and relationship cardinality, which can be queried using a relational database engine, our analysis includes the utilization of available schema documentation or data dictionaries (e.g., *KaggleDBQA_tables.json*) when migrating KaggleDBQA databases into corresponding property graphs. This documentation explicitly outlines the primary keys and foreign keys associated with related tables, offering extra information in cases where querying primary key and foreign key constraints yields no results.

It is important to acknowledge that while primary keys, foreign keys, and relationship cardinality are important factors, other criteria, such as the semantic meaning of a table, naming conventions, and query analysis, may also come into play when identifying entity tables in different contexts or methodologies. However, in order to provide a structured and automated analysis for identifying entity tables within the context of our metadata analyzer, we ensure that a table qualifies as an entity table if it satisfies any of the following three conditions.

1. A table is devoid of any foreign key, i.e. $fk == \varnothing$.
2. A table either encompasses a sole foreign key or involves more than two foreign keys, i.e. the count of foreign keys ($len(fk)$) is not equal to 2.
3. A table possesses two foreign keys and a primary key consisting of just one column, indicated by the condition $len(fk) == 2 \wedge len(pk) == 1$.

Subsequently, these entity tables are translated into graph nodes, where each node is labelled with the corresponding entity table name. Each row within an entity table corresponds to a node, while the columns in entity tables are transformed into properties of the nodes. This particular scenario is elaborated upon from line 6 to line 9 in Algorithm 1. These entity tables serve as the foundation for constructing a knowledge graph or property graph.

Secondly, we argue a table becomes a linking table if the number of foreign keys ($len(fk)$) is equal to 2, and meanwhile, one of the two additional conditions holds, i.e. the number of primary keys ($len(pk)$) is not equal to 1, or $pk == \varnothing$. Linking tables become graph edges. Each table name is represented by a type on the edges. Each row in a linking table is an edge. Columns of linking tables become edge properties. For further details, refer to the pseudocode outlined from line 10 to line 13.

In addition to the aforementioned scenarios, when neither of the two arguments is met, we establish graph edges typed in the format of *_HAS_* between two entity tables. As illustrated in Figure 3, we generate a graph edge typed as **Student_HAS_Enrolled_in** connecting graph nodes labelled as **Student** and **Enrolled_in**, respectively.

Based on our metadata analyzer, as demonstrated in Figure 3, the entity-relationship (ER) diagram located in the top left illustrates the relational database COLLEDGE_3.SQLITE. It includes an entity table labeled as $t_1 = \mathbf{Faculty}$ with its primary key $pk_1 = \langle \text{FACID} \rangle$, an entity table $t_2 = \mathbf{Department}$ with $pk_2 = \langle \text{DNO} \rangle$, and an linking table $t_3 = \mathbf{Member\_of}$ alongside two foreign keys $fk_3 = \langle \text{FACID}, \text{DNO} \rangle$. The connection between t_1 and t_3 is established via the primary and foreign key constraint $\langle pk_1, fk_3 \rangle$ where the $pk_1 \subset fk_3$ holds. Therefore, the linking table t_3 is mapped to graph edges, connecting the graph nodes converted from the entity tables t_1 and t_2. An instance of such mapping is depicted in the top right of Figure 3.

What's more, illustrated in the bottom left, we offer an example of a hyperedge e typed as **Enrolled_in**. The concept of a hyperedge arises from graph theory to represent relationships that involve more than two graph nodes. A hyperedge is useful when dealing with relationships that do not fit neatly into a one-to-one connection but involve multiple entities interacting together. Nevertheless, due to the limitation of the Neo4j data model, we can not visualise such hyperedges. Consequently, we adopt an alternative approach by turning the hyperedge e into graph nodes and apply the mapping rule outlined in the last scenario to curate three distinct graph edges connecting the transformed hyperedge e to the

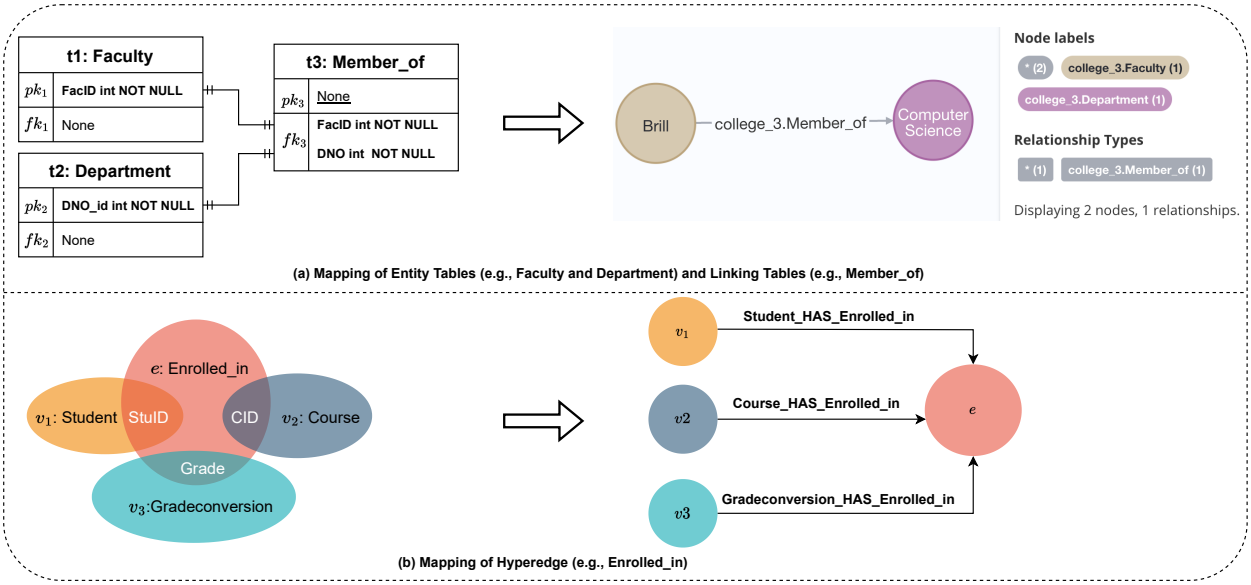


Figure 3: An illustration showcasing the handling of three mapping scenarios. These include: a) mapping entity tables such as `Faculty` and `Department` into graph nodes labelled with their respective table names, and mapping the linking table `Member_of` into graph edges, typed by the table name, and b) the transformation of a hyperedge represented as `e`.

nodes v_1, v_2, and v_3 correspondingly. This adaptation allows us to capture the essence of hyperedge relationships within the constraints of the visualization framework.

Data Repairment During the database mapping procedure, the presence of data issues, for example, inconsistencies in data types of columns working as primary keys and foreign keys, can lead to errors and mismatching issues during the query mapping procedure in Section 4.2. We summarize and address four main data issues identified by our metadata analyzer as follows.

1) **No Primary Key**. Within the Spider database, 95 out of the 876 tables (approximately 10.8%) do not feature a primary key. The approach to addressing the absence of a primary key in a relational database can be highly context-dependent, particularly during the database mapping process. In our specific case, we have identified that the absence of a primary key would result in the inability to recognize both primary and foreign key constraints. To mitigate this issue, we opt to designate a column or combination of columns that are already functioning as foreign keys in other related tables as the primary keys for these tables. In situations where no suitable candidate exists, we choose to leave the table as-is, maintaining its status without a primary key.

2) **No/Incorrect Foreign Key**. In the KaggleDBQA dataset, explicit foreign keys are not defined within the STUDENTMATHSCORE.SQLITE database, neither in the schema dump nor mentioned in the *KaggleDBQA_tables.json* file. However, we observed that out of the total twenty-eight provided SQL queries, twenty-one of them involve the utilization of `JOIN ON` statements. In these instances, specific column names are employed in a manner reminiscent of foreign keys within the SQL `JOIN ON` statements. For instance, we can deduce the presence of a foreign key relationship involving `state_code` from the following statement:

```
FINREV_FED_17 as T1 JOIN FINREV_FED_KEY_17 as T2
ON T1.state_code = T2.state_code
```

Similarly, we can infer the existence of a foreign key association involving `state` from this statement:

```
FINREV_FED_KEY_17 as T2 JOIN NDECoreExcel_Math_Grade8 as T3
ON T2.state = T3.state
```

To address this situation and establish meaningful graph edges within the SQLite databases that lack explicitly defined foreign key constraints, we propose a solution. This solution leverages the column names utilized in SQL `JOIN ON` statements as implicit associations, allowing us to create coherent graph relationships.

In the Spider dataset, the error in the provided SQL `CREATE` statements of MUSICAL.SQLITE is in the foreign key constraint definition of the `actor` table. Specifically, the FOREIGN KEY clause references the `actor` table itself rather than referencing the `musical` table, which is logically incorrect and can lead to database integrity issues. To correct the error, the FOREIGN KEY clause in the `actor` table should be modified to correctly reference the `musical` table and its primary key.

```
CREATE TABLE musical (
Musical_ID int,
Name text,
Year int,
Award text,
Category text,
Nominee text,
Result text,
PRIMARY KEY (Musical_ID)
);

CREATE TABLE actor (
Actor_ID int,
Name text,
Musical_ID int,
Character text,
Duration text,
age int,
PRIMARY KEY (Actor_ID),
FOREIGN KEY (Musical_ID) REFERENCES actor(Actor_ID)
);
```

As a result, we verify and maintain the consistency of foreign keys within the database mapping process. Above all, we take the precaution of excluding foreign keys from the set of properties associated with an edge. This helps prevent potential data consistency issues when altering the related primary keys of nodes.

3) **Database content related Issue**. In the Spider dataset, out of the 876 tables, 11 tables (1.3%) contain duplicate rows. During the graph construction process, all duplicate rows are automatically eliminated, although this may lead to discrepancies when calculating the execution accuracy score. Besides, there are seven relational databases, each of which contains only table columns but no table content. In these cases, we assign an empty value to each table column to avoid collapsed database migration.

4) **Graph Node Label Differentiation Issue**. One of the drawbacks of Neo4j is its inability to differentiate between graph node labels. For instance, consider two tables both named `singer` in SINGER.SQLITE and CONCERT_SINGER.SQLITE. If we were to store both databases in the same graph database without reformatting the table names, it could lead to a decrease in the execution accuracy score.

As depicted in Table 1, even when the generated Cypher query is correct, the EA may not exactly match the gold SQL execution answer. A closer examination reveals that the discrepancy, such as the case of `Justin Brown`, belongs to CONCERT_SINGER.SQLITE rather than SINGER.SQLITE. This error can be rectified by renaming each table using the format `domain_name.table_name`.

5) **Others**. While the above issues represent the main challenges we address during the graph construction process, it is important to acknowledge that other conditions for data repair may exist in different contexts. These could include issues related to data formatting issues such as the semantic meaning of a table and column naming conventions. Our approach is designed to provide a structured and automated analysis for identifying and repairing data issues within the context of our metadata analyzer. These conditions ensure that tables are categorized accurately based on their relational structure.

4.2 Query Mapping

Graph data models are known for their efficient pattern-matching mechanisms in dealing with highly connected data when relational databases resort to multiple expensive `JOIN ON` operations via foreign keys. Take the two examples in Figure 4 for instance. In Example 1, *Which department has more than 1 head? List the id, name, and number of heads*, we can directly map the `JOIN ON` statement into a Cypher pattern in `MATCH` clause statement where the foreign key *department_id* is omitted. For the question of Example 2, *How many departments are led by heads who are not*

Example 1	**A nested query example.**
NLQ:	`What is the name of every singer that does not have any song?`
SQL query:	`SELECT Name FROM singer WHERE Singer_ID NOT IN (SELECT Singer_ID FROM song)`
SQL query ER:	**[['Alice Walton'], ['Abigail Johnson']]**
Cypher query:	`MATCH (si:singer) WHERE NOT (si:singer)-[]-(:song) RETURN si.Name`
Cypher query ER:	**[['Justin Brown'], ['Alice Walton'], ['Abigail Johnson']]**
Example 2	**An example that duplicate elements are dropped automatically.**
NLQ:	`Which skill is used in fixing the most number of faults? List the skill id and description.?`
SQL query:	`SELECT T1.skill_id, T1.skill_description FROM Skills AS T1 JOIN Skills_Required_To_Fix AS T2 ON T1.skill_id = T2.skill_id GROUP BY T1.skill_id ORDER BY count(*) DESC LIMIT 1`
SQL query ER:	**[[3, 'TV, Video']]**
Cypher query:	`MATCH (T1:'assets_maintenance.Skills')-[T2:'assets_maintenance.Skills_Required_To_Fix']-() WITH T1.skill_description AS skill_description, count(T1.skill_id) AS count, T1.skill_id AS skill_id RETURN skill_id,skill_description ORDER BY count DESC, LIMIT 1`
Cypher query ER:	**[[2, 'Mechanical']]**

Table 1: The examples for error analysis. ER is short for execution results.

mentioned, we can convert the nested sub-query into a graph pattern in the `WHERE` statement. The equivalent Cypher clauses of both examples do not need to explicitly mention the foreign key *department_id*. Foreign keys become redundant in graph databases. Due to the fact that Cypher queries are case-sensitive, we normalised all the schema items appearing in the source case-insensitive SQL queries by using the graph schema which is aligned with relational databases.

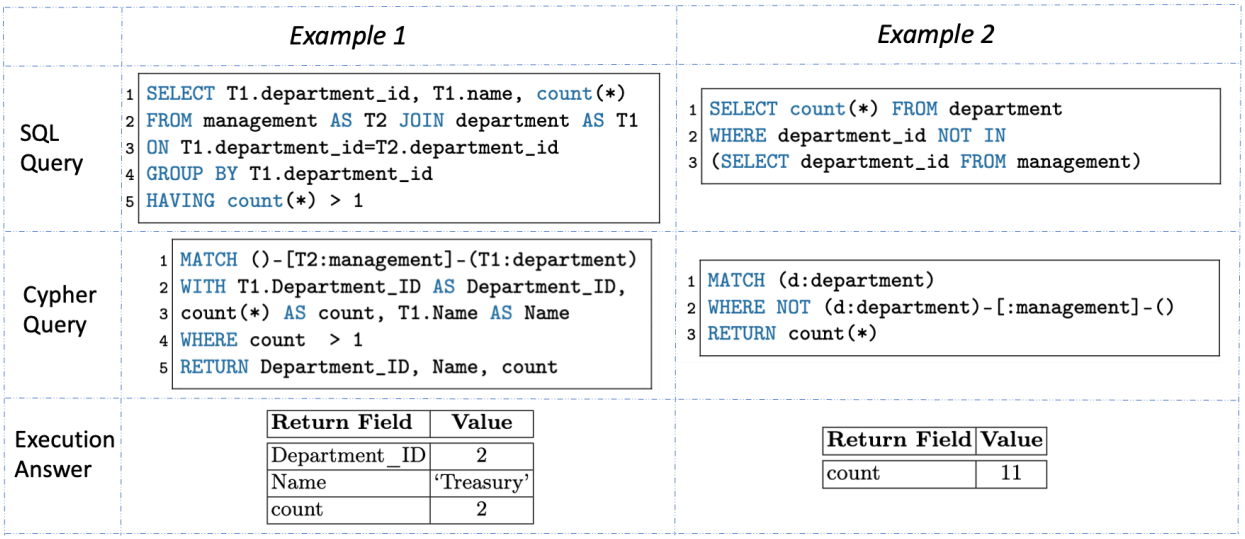


Figure 4: The goal SQL queries (top) of example 1 and example 2, their respective Cypher queries (middle) and the Cypher queries execution answers (bottom).

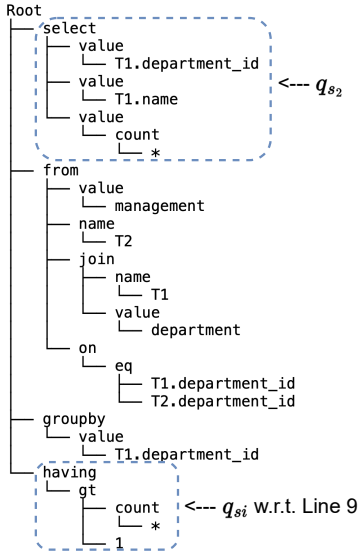
Therefore, the essential translation tasks for SQL2Cypher are to sort out the equivalent mapping between SQL and Cypher clauses and identify graph patterns. We denote a list of SQL keywords as SQL_{key}, i.e., `[FROM, WHERE, SELECT, HAVING, GROUP BY, ORDER BY, LIMIT, OFFSET, UNION]`, and a list of Cypher keywords as $Cypher_{key}$, i.e., `[MATCH, WHERE, RETURN, WITH...[AS], ORDER BY, LIMIT, OFFSET]`. We allign `FROM` with `MATCH`, `SELECT` with `RETURN`, `HAVING` with `WITH...[AS]`, `GROUP BY` with `count()`, and the rest keywords stay the same. We map `JOIN` statements to graph patterns. We chain schema items occurring in nested sub-queries or group queries together by `WITH` clause, piping the results from the nested query as the starting points of the parent query. The mapping process is repeated until all SQL clauses are visited. The pseudocode of SQL-to-Cypher process is described in Algorithm 1. We use PARSE(Q_{sql})[13] to parse a SQL query into a JSON-IZABLE parse tree (Line 6), where s_i refers to a SQL keyword appearing in SQL_{key} list and each s_i corresponds to its equivalent Cypher keyword c_i in $Cypher_{key}$ list. The mapping between s_i and c_i becomes a KEY-VALUE pair of MAP_KEY (Line 1). q_{si} is the JSON-IZABLE parse tree w.r.t. s_i. The parsed tree hierarchy is illustrated next to Algorithm 1. Given Example 1 in Figure 4, for Line 9 in Algorithm 1, q_{si} with respect to `HAVING` is shown in the dotted box at the bottom right of the parsed tree.

[13]https://github.com/mozilla/moz-sql-parser

Algorithm 2: Convert SQL to Cypher queries.

Constants: MAP_KEY = {`FROM: MATCH, WHERE: WHERE, SELECT: RETURN, HAVING: WITH...[AS], GROUP BY: count(), ORDER BY: ORDER BY, LIMIT: LIMIT, OFFSET: OFFSET`}; FUN_AGG=`[count(), max(), min(), avg()]`
Input: Q_{sql} is a SQL query. $\mathcal{G}$ refers to the given graph schema.
Output: Q_{cyp} is the converted Cypher query.
Function $F(Q_{sql}, \mathcal{G})$
  Q_{cyp}=[ ] ; // Initialise empty list.
  $S_t = \{s_i : q_{si} | s_i \in SQL_{key}\} \leftarrow \text{PARSE}(Q_{sql})$; // S_t refers to tree-structure SQL query clause statments
  foreach (s_i, q_{si}) IN S_t **do**
    $c_i \leftarrow \text{MAP\_KEY}[s_i]$; // Cypher clause keyword w.r.t. s_i
    if q_{si} IN FUN_AGG **then**
      $q_{c2} = \{c_2 : F(q_{s2}, \mathcal{G})\}$; // where s_2 = SELECT and c_2 = RETURN
      q_{ci}= {`WITH...[AS]`:$F(q_{c2}[c_2], \mathcal{G})$}
    else if q_{si} IS A NESTED SUB-QUERY **then**
      $q_{c2} = \{c_2 : F(q_{s2}, \mathcal{G})\}$; // where s_2 = SELECT and c_2 = RETURN
      q_{ci}= { `WITH...[AS]`: $F(\langle q_{ci}[c_i], q_{c2}[c_2]\rangle, \mathcal{G})$}
    else
      $q_{ci} = \{c_i : F(q_{si}, \mathcal{G})\}$
    Q_{cyp}.APPEND(q_{ci}) ; // append the ith Cypher clause statement.
  return Q_{cyp} ; // S_t IS VISITED ENTIRELY.

The S_t obtained via PARSE(Q_{sql}) for Example 1:



The mapping algorithm presented in Algorithm 2 has been meticulously designed to address the unique challenges and advantages of translating SQL queries into Cypher queries for graph databases. Several key considerations drove the design choices:

1) **Efficient Handling of Highly Connected Data**. Graph data models excel in handling highly connected data, eliminating the need for multiple costly `JOIN ON` operations used in SQL queries upon relational databases. Therefore, the algorithm is designed to seamlessly translate SQL queries into Cypher patterns, taking advantage of the graph data model's inherent ability to represent complex relationships without the explicit use of foreign keys. As the relationships between nodes are explicitly represented through edges. Thus, the algorithm is designed to omit unnecessary references to foreign keys in the translated Cypher queries, resulting in more concise and efficient queries.

2) **Case Sensitivity Compatibility** Cypher queries are case-sensitive, whereas SQL queries may not be. To ensure compatibility, the algorithm normalizes all schema items from the source SQL queries to adhere to Cypher's case sensitivity, providing consistent and accurate evaluation.

3) **Structured Mapping Process**. The algorithm's structured mapping process systematically aligns SQL keywords with their corresponding Cypher counterparts, simplifying the translation and ensuring that all clauses are processed methodically.

4) **Handling Nested Queries and Grouping**. To address the complexity of nested sub-queries or group queries, the algorithm employs the `WITH` clause to chain schema items, ensuring that results from nested queries serve as starting points for parent queries. This approach simplifies the translation of complex SQL structures into Cypher patterns.

By designing the mapping algorithm in this manner, we capitalize on the strengths of graph data models, minimize redundancy, and provide a systematic and efficient translation process from SQL to Cypher. These design choices are essential to achieve accurate and performant SQL-to-Cypher translations, facilitating the integration of relational databases with graph databases.

5 Mapping Evaluation

5.1 Datasets

In order to make our proposed approach more general and promise to work out-of-the-box on other datasets and illustrate the effectiveness of the proposed method in real-world scenarios, we evaluated the method on the challenging Spider and the KaggelDBQA datasets. Table 2 shows the data statistics.

Dataset	#DB	#Table/DB	#Row/DB	#Q_{sql}
SubSpider ($\mathcal{S}$)	155	5.1	2K	5,918
KaggleDBQA ($\mathcal{S}$)	8	2.3	280K	272

Table 2: The dataset statistics of SubSpider and KaggleDBQA. Note that if there is at least one table with a number of rows over 6,000, we filter out the whole relational database. This action resulted in a sub-Spider comprising 155 relational databases accompanied by 5,918 intricate SQL queries and simpler ones.

Spider We investigate to map and evaluate a subset of Spider. Specifically, considering the computational capacity, we drop 11 DBs of which there is at least one table containing a threshold number of rows. The Spider comprising more than 10,000 data, is more appropriate for the development of models that prioritize SQL and database knowledge as opposed to the only memorization of domain knowledge or values.

KaggleDBQA Despite the improvements on large-scale relational databases based benchmarks such as Spider, the majority of cross-domain benchmark datasets continue to focus on the database schema as opposed to the database values, thereby resulting in a large gap from the real-world scenarios. KaggleDBQA attempted to mitigate this issue by constructing 272 SQL queries from eight databases. The relational databases in KaggleDBQA are pulled from the open-source data analysis platform Kaggle and are not normalised.

5.2 Evaluations

Check Mapping Consistency and Repair We use the simple "Repair" scenario of the database mapping process (see Figure 5) to evaluate mapping consistency in task 1. In this scenario, the expected data statistics are calculated. Secondly, the mapping result is loaded using `APOC Procedures`, such as `Apoc.meta.graph()`. Then, the statistics of the obtained graph are computed in terms of node labels, edge types, and properties. Next, a difference between the two is identified and revalidated. The mapping process is rechecked and the graph is rebuilt until there is no difference between the expected and actual database statistics.

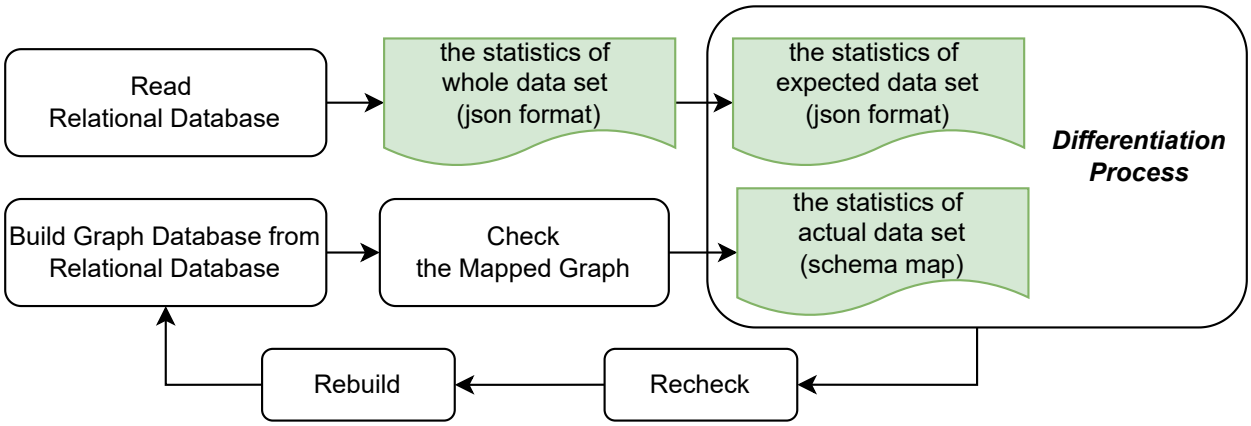


Figure 5: Phases of the `Check Mapping Consistency and Repair` Scenario.

Execution Accuracy (EA) Execution accuracy (**EA**) [Lin et al., 2020] is employed to assess data consistency. This metric quantifies the proportion of information derived from Cypher queries executed on a graph database that aligns with the outcomes of SQL queries performed on relational databases. In the context of this evaluation, we denote the result set as R^n_{cyp}, representing the output of the n^{th} Cypher query Q_{cyp}, while the result set R^n_{sql} corresponds to the outcomes obtained from the corresponding SQL query Q_{sql}. The computation of EA is formulated as follows:

$$EA = \frac{\sum_{n=1}^{N} \mathbb{1}(R^n_{sql}, R^n_{cyp})}{N}, where, \mathbb{1}(R^n_{sql}, R^n_{cyp}) = \begin{cases} 0 & \text{if } R^n_{sql} \neq R^n_{cyp} \\ 1 & \text{if } R^n_{sql} = R^n_{cyp} \end{cases} \tag{1}$$

where R^n_{sql} denotes the result set by executing the Q^n_{sql} query, and R^n_{cyp} denotes the result set by executing the semantically equivalent Cypher query Q^n_{cyp}.

Valid Score (VS) VS is designed to measure the percentage of the cases where SQLs are successfully parsed into JSON-serializable parse trees using an openly available tool[14], and accurately translated into the corresponding Cypher query via our proposed Algorithm 2. Any SQL queries that fail to be parsed into JSON documents will be declared invalid in our context. Compared with the EA metric, the VS could be considered a strict evaluation metric since it also takes into account the reliability of a SQL parser. Formally, the VS can be expressed as:

$$EA = \frac{\sum_{n=1}^{N} \mathbb{1}(R_{sql}^n, R_{cyp}^n)}{N + N(F_{sql})}, where, \mathbb{1}(R_{sql}^n, R_{cyp}^n) = \begin{cases} 0 & \text{if } R_{sql}^n \neq R_{cyp}^n \\ 1 & \text{if } R_{sql}^n = R_{cyp}^n \end{cases} \tag{2}$$

where R_{sql}^n denotes the result set by executing the Q_{sql}^n query, and R_{cyp}^n denotes the result set by executing the semantically equivalent Cypher query Q_{cyp}^n. $N(F_{sql})$ represents the count of cases in which an SQL parser failed to generate JSON documents.

5.3 Results and Analysis

In Table 3, we present the overall statistics of the mapped datasets, including the statistics of the obtained graph databases and the corresponding Cypher queries. The table provides an overview of the graph database generated from the relational databases of SubSpider and KaggleDBQA, including the number of domains (#DB), the count of graph nodes (#Nodes), the total number of edges (#Edges), and the number of Cypher queries that produced accurate results.

Property KG	**#Nodes**	**#Edges**	**#$Q_{correct\_cyp}$**	**#Q_{total_cyp}**	**EA (%)**
CySpider	10,729	11,277	4,929	5574	88.43
CyKaggleDBQA	4,760,601	2,560,961	173	272	63.60

Table 3: The statistics of the property knowledge graphs CySpider mapped from SubSpider and CyKaggleDBQA mapped from KaggleDBQA. EA shows the data consistency accuracy of the dataset mapping process.

Following the definitions and categorizations proposed by J Marton et al. [Marton et al., 2017], we focus on the statistics of Cypher queries as follows.

- *Node and edge patterns* are the basic parts which are written in `MATCH` clause. In this paper, we filter node and edge patterns in `WHERE` clause rather than providing node label/edge type constraints in `MATCH` clause.
- *Filtering patterns* express the uniqueness criterion for nodes and edges in a compact way, such as `WHERE` clause.
- An example of *negation patterns* is demonstrated by example 2 in Figure 4.
- We handle a subset of Cypher *Aggregation* operators, i.e. `count`, `avg`, `max`, `min`, `sum`. Aggregation operators often go together with `WITH` clauses to realize grouping (see example 1 in Figure 4).
- There are 71 examples containing `UNION` clause. Using just `UNION` will combine and remove duplicates from the result set.

Table 4 showcases the statistics of the obtained counterpart dataset in detail. The table is divided into two sections, each providing the key elements of the corresponding graph database and the correctly executed Cypher queries mapped from the SQL queries. The part one sections offer the overviews of the CySpider and CyKaggleDBQA graph databases' statistical data, encompassing details on the number of databases, as well as the labels and types attributed to nodes and edges. Part two sections delve into the statistics related to Cypher queries. This part includes data on node patterns and edge patterns, filtering patterns, negation patterns, and statements that incorporate aggregation operators.

Mapping Analysis Table 3 shows that the EA score of CyKaggleDBQA is lower than that of CySpider, despite CySpider's graph being significantly smaller than CyKaggleDBQA's graph. Additionally, the number of SQL query variants in KaggleDBQA is less than that in CySpider. Hence, the EA score demonstrates sensitivity to both the size of the constructed graph and the diversity of SQL query variants. We may consider exploring alternative evaluation metrics that can provide a more comprehensive assessment of our mapping results.

Our query mapping module initiates by parsing a SQL query into a JSON-serializable parse tree using an openly available tool[15]. Figures 6 and 7 illustrate the EA, VS, invalid SQL2Cypher percentage and invalid parsed percentage

[14]https://github.com/mozilla/moz-sql-parser

[15]https://github.com/mozilla/moz-sql-parser

CySpider Part 1: Graph database Statistics					
Graph Node			**Graph Edges**		
# DB	**# Labels**	**Nodes Count**	**# DB**	**# Types**	**Edge Count**
155	641	10,729	141	461	11,277
Split	**CySpider Part 2: Cypher Queries Statistics**				
	# Node Patterns	**# Edge Patterns**	**# Filtering Patterns**	**# Negation patterns**	**# Aggregation**
Train	3,046	1,684	2,401	113	1,739
Development	407	298	342	20	321

CyKaggleDBQA Part 1: Graph database Statistics					
Graph Node			**Graph Edges**		
# DB	**# Labels**	**Nodes Count**	**# DB**	**# Types**	**Edge Count**
8	17	4,760,601	4	8	2,560,961
Split	**CyKaggleDBQA Part 2: Cypher Queries Statistics**				
	# Node Patterns	**# Edge Patterns**	**# Filtering Patterns**	**# Negation patterns**	**# Aggregation**
GeoNuclearData	22	0	18	0	13
GreaterManchesterCrime	24	0	18	0	21
Pesticide	30	1	14	0	18
StudentMathScore	14	1	9	0	2
TheHistoryofBaseball	19	5	16	0	8
USWildFires	26	0	10	0	22
WhatCDHipHop	22	0	9	0	3
WorldSoccerDataBase	13	0	9	0	7

Table 4: Statistics of CySpider and CyKaggleDBQA, including the graph database statistics and the corresponding Cypher queries, constructed from a subSpider and KaggleDBQA respectively.

from top to bottom and left to right. The VS score and invalid parsed percentage show inverse proportionality characteristics. The lower proportion of invalid parsed SQL contributed to the higher overall valid score, which suggests that a better SQL parser would enhance query mapping accuracy to Cypher queries. The development of an improved SQL parser would be one of the future works. A SQL parser with higher performance would facilitate the better translation of SQL to other database query languages such as SPARQL.

Certain false negative cases arose due to limitations inherent in the Cypher query language. For instance, in the `GreaterManchesterCrime` domain, all the incorrect execution results are owing to the instances of false negatives. These occur when the `GROUP BY Location` statement in the source SQL queries implicitly constrains the relational database engine to sort the field in alphabetical order if it is of text type. However, this cannot be accurately translated into a Cypher clause.

6 Conclusion

This paper proposes an automated approach to migrate data and queries from relational databases to a property graph database. The translation leverages the integrity constraints defined over the source to systematically build a corresponding property knowledge graph. The primary aim of this approach was to enhance the inherent structure of relational data through graph representation, resulting in optimized query operations when addressing semantic-level tasks, e.g. semantic parsing. We use the execution accuracy metric to evaluate the mapping process at the formal semantic query language level including complex queries. We suggested evaluating our approach on another large-scale dataset such as BIRD [Li et al., 2023] to further enhance the mapping process's accuracy, efficiency, and coverage for comparable experimental outcomes.

Acknowledgment

This research is supported by the Australian Research Council through the Centre for Transforming Maintenance through Data Science (grant number IC180100030), funded by the Australian Government.

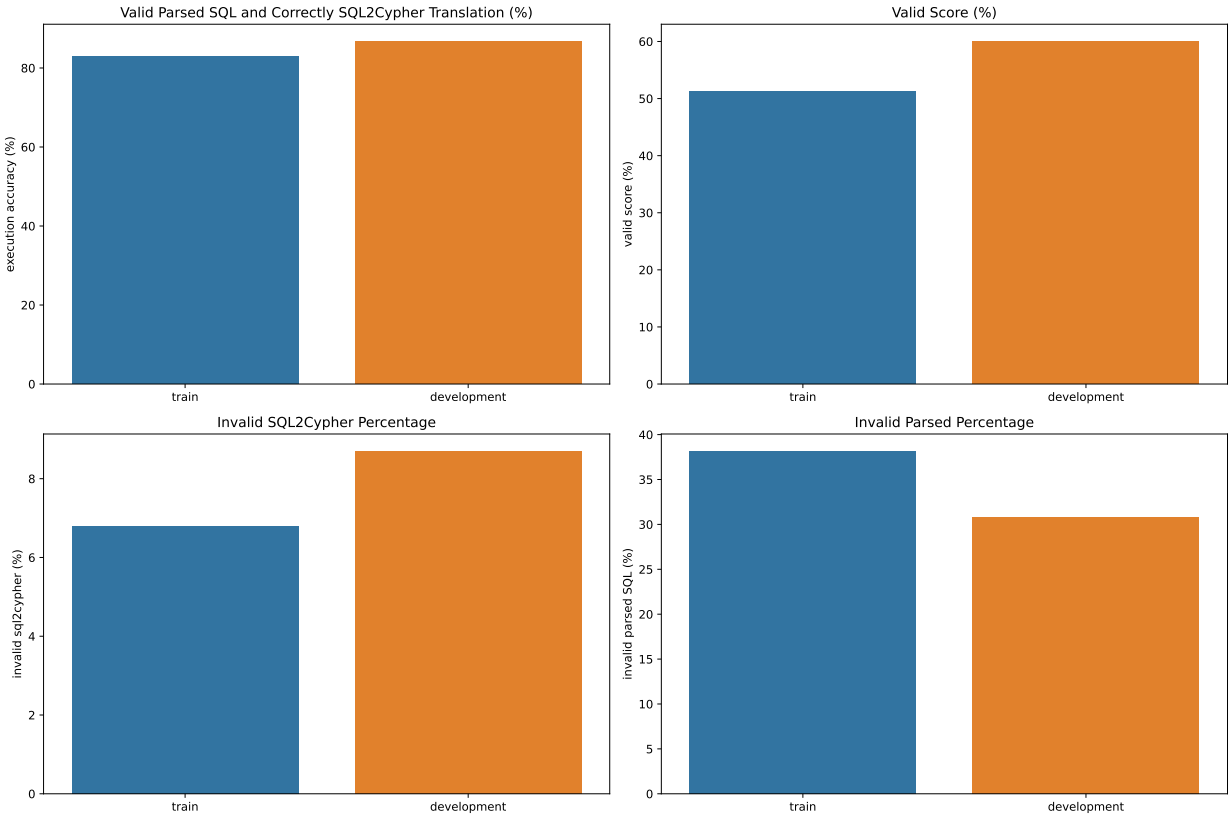


Figure 6: The query mapping quality distribution from the four aspects on SubSpider.

References

I Made Putrama and Péter Martinek. An automated graph construction approach from relational databases to neo4j. In *2022 IEEE 22nd International Symposium on Computational Intelligence and Informatics and 8th IEEE International Conference on Recent Achievements in Mechatronics, Automation, Computer Science and Robotics (CINTI-MACRo)*, pages 000131–000136. IEEE, 2022.

József Marton, Gábor Szárnyas, and Dániel Varró. Formalising openCypher graph queries in relational algebra. In *European Conference on Advances in Databases and Information Systems*, pages 182–196. Springer, 2017.

Erik F Sang and Fien De Meulder. Introduction to the conll-2003 shared task: Language-independent named entity recognition. *arXiv preprint cs/0306050*, 2003.

David Milne and Ian H Witten. Learning to link with wikipedia. In *Proceedings of the 17th ACM conference on Information and knowledge management*, pages 509–518, 2008.

Dmitry Zelenko, Chinatsu Aone, and Anthony Richardella. Kernel methods for relation extraction. *Journal of machine learning research*, 3(Feb):1083–1106, 2003.

Yubo Chen, Liheng Xu, Kang Liu, Daojian Zeng, and Jun Zhao. Event extraction via dynamic multi-pooling convolutional neural networks. In *Proceedings of the 53rd Annual Meeting of the Association for Computational Linguistics and the 7th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 167–176, 2015.

Nora Abdelmageed and Sirko Schindler. Jentab: Matching tabular data to knowledge graphs. In *SemTab@ ISWC*, pages 40–49, 2020.

Daniil Sorokin. Knowledge graphs and graph neural networks for semantic parsing. 2021.

Ziyu Zhao, Michael Stewart, Wei Liu, Tim French, and Melinda Hodkiewicz. Natural language query for technical knowledge graph navigation. In *Australasian Conference on Data Mining*, pages 176–191. Springer, 2022.

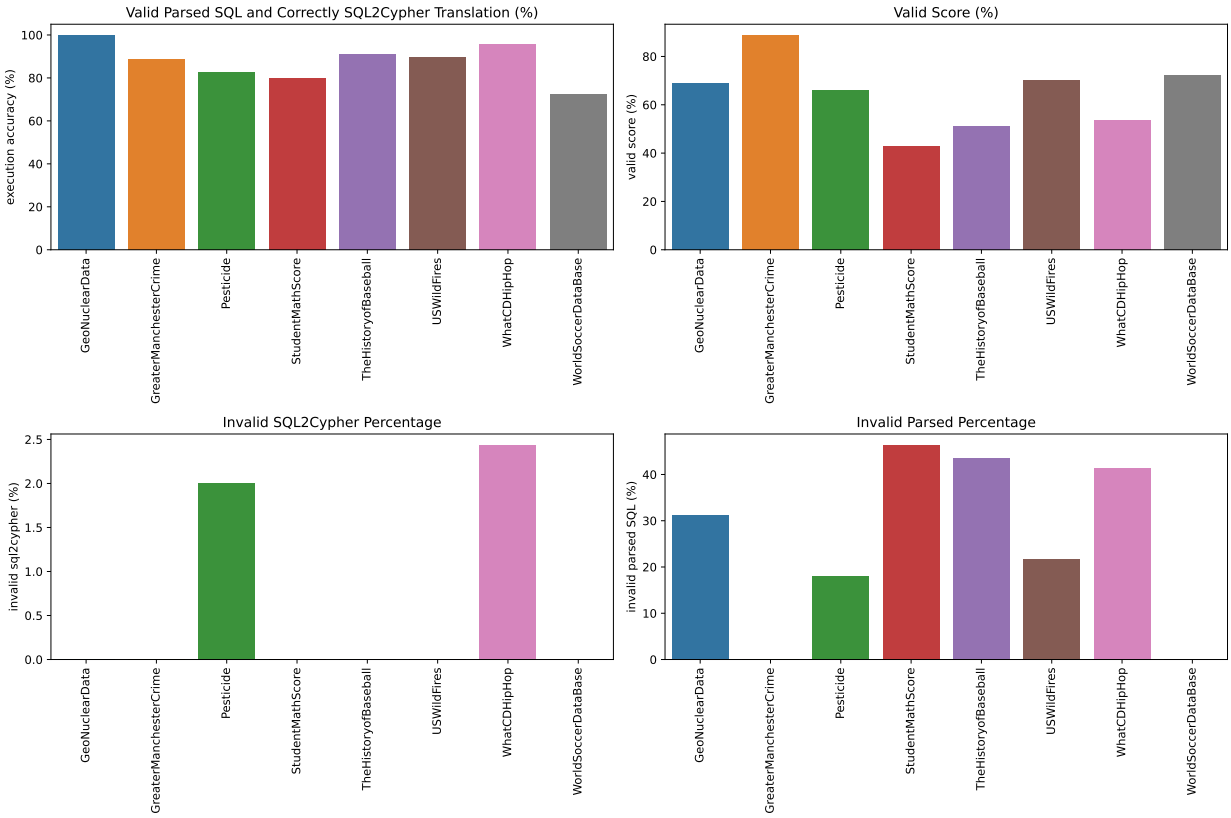


Figure 7: The query mapping quality distribution from the four aspects on KaggleDBQA.

Nilesh Chakraborty, Denis Lukovnikov, Gaurav Maheshwari, Priyansh Trivedi, Jens Lehmann, and Asja Fischer. Introduction to neural network based approaches for question answering over knowledge graphs. *arXiv preprint arXiv:1907.09361*, 2019.

Tao Yu, Rui Zhang, Kai Yang, Michihiro Yasunaga, Dongxu Wang, Zifan Li, James Ma, Irene Li, Qingning Yao, Shanelle Roman, et al. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and Text-to-SQL task. *arXiv preprint arXiv:1809.08887*, 2018.

Chia-Hsuan Lee, Oleksandr Polozov, and Matthew Richardson. KaggleDBQA: Realistic evaluation of text-to-SQL parsers. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing (Volume 1: Long Papers)*, pages 2261–2273, Online, August 2021. Association for Computational Linguistics.

Xi Victoria Lin, Richard Socher, and Caiming Xiong. Bridging textual and tabular data for cross-domain Text-to-SQL semantic parsing. In *Proceedings of the 2020 Conference on Empirical Methods in Natural Language Processing: Findings, EMNLP (Nov 2020)*, 2020.

Hongbin Ye, Ningyu Zhang, Hui Chen, and Huajun Chen. Generative knowledge graph construction: A review. *arXiv preprint arXiv:2210.12714*, 2022.

Michael Stewart, Melinda Hodkiewicz, Wei Liu, and Tim French. Mwo2kg and echidna: Constructing and exploring knowledge graphs from maintenance data. *Proceedings of the Institution of Mechanical Engineers, Part O: Journal of Risk and Reliability*, page 1748006X221131128.

Jason PC Chiu and Eric Nichols. Named entity recognition with bidirectional lstm-cnns. *Transactions of the association for computational linguistics*, 4:357–370, 2016.

Giovanni Paolini, Ben Athiwaratkun, Jason Krone, Jie Ma, Alessandro Achille, Rishita Anubhai, Cicero Nogueira dos Santos, Bing Xiang, and Stefano Soatto. Structured prediction as translation between augmented natural languages. *arXiv preprint arXiv:2101.05779*, 2021.

Zhepei Wei, Jianlin Su, Yue Wang, Yuan Tian, and Yi Chang. A novel cascade binary tagging framework for relational triple extraction. *arXiv preprint arXiv:1909.03227*, 2019.

Pontus Stenetorp, Sampo Pyysalo, Goran Topić, Tomoko Ohta, Sophia Ananiadou, and Jun'ichi Tsujii. brat: a web-based tool for NLP-assisted text annotation. In *Proceedings of the Demonstrations at the 13th Conference of the European Chapter of the Association for Computational Linguistics*, pages 102–107, Avignon, France, April 2012. Association for Computational Linguistics.

Hiroki Nakayama, Takahiro Kubo, Junya Kamura, Yasufumi Taniguchi, and Xu Liang. doccano: Text annotation tool for human, 2018. URL `https://github.com/doccano/doccano`. Software available from https://github.com/doccano/doccano.

Michael Stewart, Wei Liu, and Rachel Cardell-Oliver. Redcoat: A collaborative annotation tool for hierarchical entity typing. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP): System Demonstrations*, pages 193–198, Hong Kong, China, November 2019. Association for Computational Linguistics. doi:10.18653/v1/D19-3033.

Tyler Bikaun, Michael Stewart, and Wei Liu. QuickGraph: A rapid annotation tool for knowledge graph extraction from technical text. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics: System Demonstrations*, pages 270–278, Dublin, Ireland, May 2022. Association for Computational Linguistics.

Oshin Agarwal, Mihir Kale, Heming Ge, Siamak Shakeri, and Rami Al-Rfou. Machine translation aided bilingual data-to-text generation and semantic parsing. In *Proceedings of the 3rd international workshop on natural language generation from the semantic web (WebNLG+)*, pages 125–130, 2020.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *The Journal of Machine Learning Research*, 21(1):5485–5551, 2020.

Pierre L Dognin, Igor Melnyk, Inkit Padhi, Cicero Nogueira dos Santos, and Payel Das. Dualtkb: A dual learning bridge between text and knowledge base. *arXiv preprint arXiv:2010.14660*, 2020.

Junyoung Chung, Caglar Gulcehre, KyungHyun Cho, and Yoshua Bengio. Empirical evaluation of gated recurrent neural networks on sequence modeling. *arXiv preprint arXiv:1412.3555*, 2014.

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. *Advances in neural information processing systems*, 30, 2017.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Igor Melnyk, Pierre Dognin, and Payel Das. Knowledge graph generation from text. *arXiv preprint arXiv:2211.10511*, 2022.

Jiaxuan You, Rex Ying, Xiang Ren, William Hamilton, and Jure Leskovec. Graphrnn: Generating realistic graphs with deep auto-regressive models. In *International conference on machine learning*, pages 5708–5717. PMLR, 2018.

Qipeng Guo, Zhijing Jin, Xipeng Qiu, Weinan Zhang, David Wipf, and Zheng Zhang. Cyclegt: Unsupervised graph-to-text and text-to-graph generation via cycle training. *arXiv preprint arXiv:2006.04702*, 2020.

Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Ves Stoyanov, and Luke Zettlemoyer. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. *arXiv preprint arXiv:1910.13461*, 2019.

Sangkeun Lee, Byung H Park, Seung-Hwan Lim, and Mallikarjun Shankar. Table2graph: A scalable graph construction from relational tables using map-reduce. In *2015 IEEE First International Conference on Big Data Computing Service and Applications*, pages 294–301. IEEE, 2015.

NICOLAS LACHICHE. Performance of nosql graph implementations of star vs. snowflake schemas.

Hui Feng and Meigen Huang. An approach to converting relational database to graph database: from mysql to neo4j. In *2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*, pages 674–680. IEEE, 2022.

Xiang Wang, Xiangnan He, Yixin Cao, Meng Liu, and Tat-Seng Chua. Kgat: Knowledge graph attention network for recommendation. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*, pages 950–958, 2019.

Tahereh Pourhabibi, Kok-Leong Ong, Booi H Kam, and Yee Ling Boo. Fraud detection: A systematic literature review of graph-based anomaly detection approaches. *Decision Support Systems*, 133:113303, 2020.

Gábor Szárnyas, Benedek Izsó, István Ráth, and Dániel Varró. The train benchmark: cross-technology performance evaluation of continuous model queries. *Software & Systems Modeling*, 17(4):1365–1393, 2018.

Jinyang Li, Binyuan Hui, Ge Qu, Binhua Li, Jiaxi Yang, Bowen Li, Bailin Wang, Bowen Qin, Rongyu Cao, Ruiying Geng, et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111*, 2023.