Graphiti: Bridging Graph and Relational Database Queries

YANG HE, Simon Fraser University, Canada
RUIJIE FANG, University of Texas at Austin, USA
IŞIL DILLIG, University of Texas at Austin, USA
YUEPENG WANG, Simon Fraser University, Canada

This paper presents an automated reasoning technique for checking equivalence between graph database queries written in Cypher and relational queries in SQL. To formalize a suitable notion of equivalence in this setting, we introduce the concept of *database transformers*, which transform database instances between graph and relational models. We then propose a novel verification methodology that checks equivalence modulo a given transformer by reducing the original problem to verifying equivalence between a pair of SQL queries. This reduction is achieved by embedding a subset of Cypher into SQL through syntax-directed translation, allowing us to leverage existing research on automated reasoning for SQL while obviating the need for reasoning simultaneously over two different data models. We have implemented our approach in a tool called Graphiti and used it to check equivalence between graph and relational queries. Our experiments demonstrate that Graphiti is useful both for verification and refutation and that it can uncover subtle bugs, including those found in Cypher tutorials and academic papers.

CCS Concepts: • **Software and its engineering** → **Automatic programming**; **Software verification**; **Formal software verification**; • **Theory of computation** → **Program verification**.

Additional Key Words and Phrases: Program Verification, Equivalence Checking, Relational Databases, Graph Databases.

ACM Reference Format:
Yang He, Ruijie Fang, Işıl Dillig, and Yuepeng Wang. 2025. Graphiti: Bridging Graph and Relational Database Queries. *Proc. ACM Program. Lang.* 9, PLDI, Article 216 (June 2025), 25 pages. https://doi.org/10.1145/3729319

1 Introduction

Over the past decades, *graph* databases have garnered significant attention from both industry and academia, offering more flexible data models with different trade-offs compared to *relational* databases. As a result, developers are increasingly interested in migrating relational database applications to graph databases [54], and systems like Apache Age [1] aim to incorporate graph components into relational databases to help with this transition.

Nevertheless, transitioning between relational and graph databases often requires developers to convert queries from one model to the other, and, as evidenced by numerous posts on online forums [3, 40, 49], translating between relational and graph queries can be quite challenging due to misunderstandings of joins versus relationships, aggregation semantics, and other complexities. In fact, we have identified multiple incorrect translations in existing literature where queries claimed to be equivalent were, in reality, non-equivalent [32, 39, 42]. This underscores a growing need for rigorous reasoning about the equivalence of graph and relational queries.

Authors' Contact Information: Yang He, Simon Fraser University, Burnaby, Canada, yha244@sfu.ca; Ruijie Fang, University of Texas at Austin, Austin, USA, ruijief@cs.utexas.edu; Işıl Dillig, University of Texas at Austin, Austin, USA, isil@cs.utexas.edu; Yuepeng Wang, Simon Fraser University, Burnaby, Canada, yuepeng@sfu.ca.

ACM 2475-1421/2025/6-ART216
https://doi.org/10.1145/3729319

Although significant progress has been made in verifying the equivalence of relational database queries [10–12, 57], to the best of our knowledge, no existing work addresses the equivalence verification problem between relational and graph queries. A key challenge in this area arises from the distinct data models of relational and graph databases. Relational databases organize data in tables, with queries operating on rows and columns through well-defined operations like joins and aggregations. In contrast, graph databases represent data as nodes and edges, with queries typically expressing relationships and traversals through graph structures. This fundamental difference in data representation complicates both the definition of equivalence between graph and relational queries and the verification process itself.

This paper takes a first step towards developing automated reasoning techniques that can be used to check equivalence between relational queries written in SQL and graph database queries implemented in Cypher [19], the most popular graph database query language. Our formal reasoning technique is built on a novel method that embeds a subset of the Cypher language into SQL, building on the insight that paths in a graph database instance correspond to joins of rows in a relational database. This observation not only allows us to translate Cypher queries to SQL in a syntax-directed way but also facilitates checking equivalence between Cypher and SQL queries. At a high level, our approach hinges on three crucial components: **(1)** a formal foundation for defining equivalence between graph and relational database instances (over arbitrary schemas); **(2)** a correct-by-construction technique for translating graph database queries to relational queries (over a specific schema); and **(3)** a novel verification methodology that leverages (2) to establish equivalence between Cypher and SQL queries that operate over any arbitrary schema. We next explain each of our contributions in more detail.

Formal foundation for graph and relational database equivalence. As mentioned earlier, a key challenge in reasoning about equivalence between graph and relational queries is the lack of a straightforward mapping between the data models. To address this, we introduce the concept of *database transformer*, adapted from prior work on schema mappings [17, 37, 59], which allows transforming a database instance from one data model (graphs) to an equivalent instance in another model (relational databases). This transformation forms the foundation for defining equivalence between graph and relational database queries.

Correct-by-construction transpilation. Building on this notion of database transformer, we introduce the concept of a *standard database transformer (SDT)* as the default correctness specification. In simple terms, the SDT provides a set of transformation rules to map graph elements (nodes and edges) into relational tables, maintaining the structure and semantics of the graph within the relational model. For example, nodes in a graph schema are transformed into tables in the relational schema, where attributes of the node become columns, and edges are represented as relationships between these tables with foreign keys. The resulting relational schema is referred to as the *induced relational schema*. Our method then defines syntax-directed transpilation rules to convert any Cypher query into a SQL query over this induced schema. The core insight is that Cypher path queries, which perform pattern matching over subgraphs, can be mapped to relational joins. However, the actual translation is tricky due to the fact that Cypher supports flexible, multi-step pattern matching over graph structures, which demands careful handling to ensure that pattern matching in Cypher—whether simple or complex—is accurately represented as joins in SQL, preserving the semantics of the original graph query.

Equivalence checking methodology for arbitrary schema. While the transpilation method described above can generate an equivalent SQL query, the equivalence is *modulo* the SDT. However, in practice, the target relational database often uses a different schema (rather than the *induced*

relational schema), so the syntax-directed transpilation method alone is insufficient. To address this gap, we propose a verification methodology that checks equivalence between graph and relational queries modulo any database transformer.

The key insight of our approach is that, instead of directly reasoning about equivalence between Cypher and SQL queries—which would require complex SMT encodings that combine graph and relational structures—we reduce the problem to checking equivalence between SQL queries over different schemas. This reduction allows us to leverage existing techniques and tools for SQL equivalence checking [26, 57], avoiding the need for handling the intricate challenge of reasoning over two fundamentally different data models at the same time.

Figure 1 illustrates the proposed approach for checking equivalence between Cypher and SQL queries. Our method takes four inputs: (1) a Cypher query Q_G, (2) an SQL query Q_R, (3) schemas for the graph and relational databases, and (4) a correctness specification Φ in the form of a database transformer. To establish equivalence between Q_G and Q_R modulo Φ, the method first derives the induced relational schema and the SDT. It then applies a correct-by-construction transpilation technique to generate a SQL query Q'_R that is provably equivalent to Q_G modulo the SDT (but not modulo Φ). In the final step, the method computes a semantic "diff" between the induced and target relational schemas, constructing a *residual database transformer* to align the two relational instances. Finally, an off-the-shelf SQL equivalence checker is then used to check equivalence between Q'_R and Q_R.

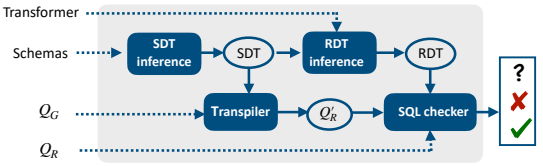


Fig. 1. Overview of approach.

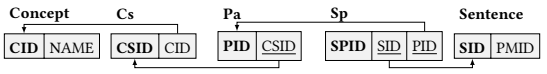
Overall, our proposed methodology has two key advantages. First, when the user just wants to translate a Cypher query to SQL but does not care about the underlying relational schema (or if the desired schema is the same as the induced relational schema), our method can be used to perform correct-by-construction transpilation. Second, given any arbitrary correctness specification (in the form of a database transformer), our method can leverage a combination of the proposed transpilation approach and existing automated reasoning tools for SQL to reason about equivalence between any pair of Cypher and SQL queries.

We have implemented the proposed approach in a new tool called Graphiti for reasoning about equivalence between Cypher and SQL queries and conducted an extensive experimental evaluation of Graphiti on 410 benchmarks. These include 45 benchmarks sourced from public platforms such as StackOverflow, tutorials, and academic papers; 160 benchmarks translated from SQL by students with Cypher experience; and 205 benchmarks translated using ChatGPT. In the first evaluation, we combine Graphiti with the VeriEQL [26] bounded model checker for SQL, performing equivalence verification on all 410 query pairs. This reveals equivalence violations in 34 benchmarks, including 3 from the wild, 4 from manual translations, and 27 from GPT-generated translations. In the second evaluation, we pair Graphiti with the deductive SQL verifier Mediator [57] for full-fledged verification of an aggregation-free subset. This enables unbounded equivalence verification between Cypher and SQL queries, where both tables and graphs can have arbitrary sizes. Here, about 80% of supported queries are verified as equivalent in a push-button manner. Finally, in the third experiment, we show that Graphiti can generate SQL queries that are competitive with manually-written ones in terms of execution efficiency.

Contributions. To summarize, this paper makes the following key contributions:

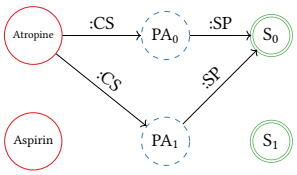
- We propose the first technique for reasoning about equivalence between graph and relational queries, based on a formal definition of equivalence modulo database transformer.

(a) Graph schema. CONCEPT has two property keys: CID and Name. PA has two property keys: PID and CSID. SENTENCE has two property keys: SID and PMID.



(b) Relational schema. Primary keys are in bold. Foreign keys are underlined and connected to references with arrows.

Fig. 2. A pair of relational and graph schemas.



(a) Graph database instance. Aspirin and Atropine denote two Concept nodes, PA_0 and PA_1 are two PA nodes, and S_0 and S_1 are two Sentence nodes.

CID	NAME
1	Atropine
2	Aspirin

Concept

CID	CSID
1	0
1	1

Cs

PID	CSID
0	0
1	1

Pa

SPID	SID	PID
0	0	0
1	0	1

Sp

SID	PMID
0	0
1	0

Sentence

(b) Relational database instance.

Fig. 3. Example graph and relational database instances.

- We introduce the concept of *standard database transformer*, which acts as the default correctness specification for equivalence between Cypher and SQL queries.
- We develop a sound and complete transpilation technique that translates a subset of Cypher queries into equivalent SQL queries, guaranteeing that both queries produce the same results under *standard database transformer*.
- We show how to leverage the proposed transpilation approach to reduce the equivalence checking problem (modulo *any* database transformer) into the problem of checking equivalence between a pair of SQL queries, allowing us to leverage existing work on automated reasoning for SQL.
- We implement these ideas in a tool called Graphiti and conduct an empirical evaluation on 410 benchmarks, showing that our approach can be used for both verification and falsification.

2 Motivating Example

In this section, we motivate the problem addressed in this work through an example from prior work [32] that studies SQL analytics for graphs. The queries in this section pertain to a real-world biomedical research database [41].

Incorrect translation. Figure 2a shows a graph schema that contains three types of nodes called `CONCEPT`, `PA`[1], and `SENTENCE`. The `CS` relationship (represented by an edge) links concepts to predication arguments, and the `SP` relationship (also represented as an edge) links predication arguments to sentences. On the other hand, Figure 2b shows the relational representation of the graph model. As shown in Figure 2b, the relational schema contains five tables called `Concept`, `Cs`, `Pa`, `Sp`, `Sentence` that correspond to both the nodes and edges of the graph schema in Figure 2a. For some intuition about the correspondence between the two databases, Figure 3 shows sample instances of both database schemas that contain the same entries "Atropine" and "Aspirin".

Next, consider the SQL and Cypher queries shown in Figures 4a and 4c respectively. The SQL query aims to find all concepts that link to a concept `c1` with `CID = 1` and their corresponding frequencies of connected paths to `c1` through the join of `Cs`, `Pa`, and `Sp` tables. Similarly, the Cypher query first finds all sentences linked to c_1 through the `CS - PA - SP` path and then counts the frequencies of paths from those sentences to all concepts. According to Lin et al. [32], these queries

[1]Here, `PA` stands for *predication argument*

```
SELECT c2.CID, Count (*) FROM Cs AS c2, Pa AS p2, Sp AS s2
WHERE s2.PID = p2.PID AND p2.CSID = c2.CSID AND s2.SID IN (
    SELECT s1.SID FROM Cs AS c1, Pa AS p1, Sp AS s1
    WHERE s1.PID = p1.PID AND p1.CSID = c1.CSID AND c1.CID = 1 )
GROUP BY CID
```

(a) SQL query

c2.CID	Count(*)
1	2

(b) The result of SQL query.

```
MATCH (c1:CONCEPT {CID:1})-[r1:CS]->(p1:PA)-[r2:SP]->(s:SENTENCE)
WITH s
MATCH (s:SENTENCE)<-[r3:SP]-(p2:PA)<-[r4:CS]-(c2:CONCEPT)
RETURN c2.CID, Count (*)
```

(c) Cypher query

c2.CID	Count(*)
1	4

(d) The result of Cypher query.

Fig. 4. A pair of SQL and Cypher queries with their execution results.

$$\texttt{CONCEPT}(\texttt{cid}, \texttt{name}) \rightarrow \texttt{Concept}(\texttt{cid}, \texttt{name}) \qquad \texttt{CONCEPT}(\texttt{cid}, \_), \texttt{CS}(\texttt{cid}, \texttt{csid}, \texttt{cid}, \texttt{pid}), \texttt{PA}(\texttt{pid}, \texttt{csid}) \rightarrow \texttt{Cs}(\texttt{cid}, \texttt{csid})$$

$$\texttt{PA}(\texttt{pid}, \texttt{csid}) \rightarrow \texttt{Pa}(\texttt{pid}, \texttt{csid}) \qquad \texttt{PA}(\texttt{pid}, \_), \texttt{SP}(\texttt{spid}, \texttt{sid}, \texttt{pid}, \texttt{pid}, \texttt{sid}), \texttt{SENTENCE}(\texttt{sid}, \_) \rightarrow \texttt{Sp}(\texttt{spid}, \texttt{sid}, \texttt{pid})$$

$$\texttt{SENTENCE}(\texttt{sid}, \texttt{pmid}) \rightarrow \texttt{Sentence}(\texttt{sid}, \texttt{pmid})$$

Fig. 5. Database transformer for our example. All variables are implicitly universally quantified.

are intended to be equivalent; however, they are actually *not* equivalent due to the subtle differences in Cypher and SQL semantics. At a high level, both queries explore how certain concepts (starting with CID = 1) are linked to other concepts via their occurrence in shared sentences. They do this by traversing relationships (either explicitly in the graph or via joins in the relational database) and aggregating the results to identify how frequently these connections occur. However, the two queries actually differ in how they compute the frequencies and end up providing different results on two database instances that are meant to contain the same data.

In particular, when run on the database instances from Figure 3, the SQL query produces the table shown in Figure 4b whereas the Cypher query produces the one in Figure 4d. These results agree on the CID's of the related concepts; however they differ on the *frequencies* of the relationships, as is evident from the entries in the Count column of these tables.[2] As this example illustrates, queries that *appear* to be ostensibly equivalent can have subtle differences in their semantics, motivating the need for automated reasoning tools that can be used to expose semantic differences between graph and relational queries. In the remainder of this section, we elucidate some important aspects and design choices behind our proposed approach.

Need for database transformers. In order to conclude that the SQL and Cypher queries are semantically different, we need to reason about how they behave when executed on the *same data*. However, because graph and relational databases have such different data models, the input database instances are never identical. Thus, in order to reason about query equivalence, we first need a mechanism for defining *data equivalence*. In our framework, this is done through the concept of *database transformer*, which takes as input a graph database instance D over a certain schema Ψ and produces a relational database instance D' over a relational schema Ψ'. Our concept of database transformer is adapted from prior work on *schema mappings* [17, 37], generalized to model the correspondence between graph and relational databases. For our running example, the correspondence between the two database instances is given by the transformer shown in Figure 5. Intuitively, each rule describes how each table in the relational database can be generated based on nodes and edges in the graph. For example, the rule $\texttt{CONCEPT}(\texttt{cid}, \_), \texttt{CS}(\texttt{cid}, \texttt{csid}, \texttt{cid}, \texttt{pid}), \texttt{PA}(\texttt{pid}, \texttt{csid}) \rightarrow \texttt{Cs}(\texttt{cid}, \texttt{csid})$ specifies if there is an edge CS connecting two nodes CONCEPT and PA in the graph, and the first two properties of CS are cid and csid, then there is a row (cid, csid) in the Cs table. Here, the last two attributes of CS serve as foreign keys referencing to the source and target nodes of the CS edge.

[2]An equivalent Cypher query of the SQL query in Figure 4a is shown in the Appendix of the extended version [25].

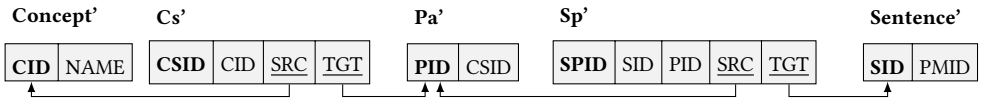


Fig. 6. Induced relational schema. Primary keys are in bold. Foreign keys are underlined and connected to references with arrows.

Induced relational schema and default transformer. As mentioned in Section 1, our approach to reasoning about equivalence between SQL and Cypher queries relies on first transpiling the given Cypher query to a SQL query over the *induced relational schema*, which corresponds to a natural relational representation of the graph database. In particular, Figure 6 shows the induced relational schema for Figure 2a and is obtained by (a) translating both node and edge types in the graph schema into tables, and (b) translating incidence and adjacency information between node and edge types into *functional dependencies*. For instance, nodes of type `CONCEPT` are mapped to the `Concept` table, edges of type `CS` are mapped to the `Cs` table, and so on. To handle functional dependencies, we need to introduce foreign keys in the corresponding tables. Compared to the relational schema in Figure 2b, the induced relational schema preserves the attributes while using additional attributes (i.e. `SRC` and `TGT`) as foreign keys to represent functional dependencies. For example, for edges of type `CS`, the source node is of type `CONCEPT`; thus, the induced relational schema has the `SRC` attribute as a foreign key to `CID` of the `Concept` table. Similarly, the `TGT` attribute is considered a foreign key to the `PA` table. Given the original graph database schema, our method constructs a so-called *standard database transformer* Φ_{sdt} that can be used to convert any instance of the given graph schema to a relational database over its induced schema.

Syntax-directed transpilation. Intuitively, the standard database transformer establishes a one-to-one correspondence between elements of the graph database and the corresponding entries in the induced relational database. Hence, we can use syntax-directed translation to directly transpile the Cypher query to a SQL query over the induced schema. Here, the transformer Φ_{sdt} fixes the mapping between "atomic elements" of both databases; thus, atomic queries over nodes and edges in Cypher can be translated to atomic queries over SQL tables. This forms the base case for an inductive transpilation scheme, leveraging the key insight that Cypher path queries correspond to relational joins. For example, a Cypher path query like **`MATCH`** (u)`-[:CS]->`(v) translates to a SQL join query between the `Concept`, `Cs`, and `Pa` tables, since the database transformer specifies that the `CS` edge type maps to the `Cs` table in the relational schema. Such a transpilation scheme is conceptually simple, but as our final transpilation result in Figure 7 shows, one must take significant care to address the different types of path patterns in Cypher syntax, in addition to translating compositions of path queries with other Cypher operators such as aggregation. Specifically, the first pattern matching in Figure 4c is translated to `T1` while the `WITH` clause propagates the intermediate results to `T2`. Similarly, the second pattern matching is translated to `T3`. Due to the shared node `s:SENTENCE` in these two path patterns, we join `T2` and `T3` as `T4`. The final `RETURN` clause is translated to `GroupBy` because of the aggregation expression.

```
WITH T1 AS ( SELECT c1.CID AS c1_CID, …, s.SID AS s_SID
     FROM Concept AS c1 JOIN CS AS r1
          JOIN PA AS p1 JOIN SP AS r2 JOIN Sentence AS s
     ON c1.CID = 1 AND c1.CID = r1.SRC AND … AND r2.TGT = s.SID ),
     T2 AS ( SELECT s_SID FROM T1 ),
     T3 AS ( SELECT s.SID AS s_SID, …, c2.CID AS c2_CID
     FROM Sentence AS s JOIN SP AS r3
          JOIN PA AS p2 JOIN CS AS r4 JOIN Concept AS c2
     ON s.SID = r3.TGT AND … AND r4.SRC = c2.CID ),
     T4 AS ( SELECT * FROM T2 JOIN T3 ON T2.s_SID = T3.s_SID )
SELECT T4.c2_CID, Count (*) FROM T4 GROUP BY T4.c2_CID
```

Fig. 7. Transpilation result for the Cypher query.

Checking equivalence. While our approach allows correct-by-construction transpilation from Cypher to SQL, there are several reasons why this is not sufficient. First, our approach does

$$\texttt{Concept'}(\texttt{cid}, \texttt{name}) \rightarrow \texttt{Concept}(\texttt{cid}, \texttt{name}) \qquad \texttt{Concept'}(\texttt{cid}, \_), \texttt{Cs'}(\texttt{cid}, \texttt{csid}, \texttt{cid}, \texttt{pid}), \texttt{Pa'}(\texttt{pid}, \texttt{csid}) \rightarrow \texttt{Cs}(\texttt{cid}, \texttt{csid})$$
$$\texttt{Pa'}(\texttt{pid}, \texttt{csid}) \rightarrow \texttt{Pa}(\texttt{pid}, \texttt{csid}) \qquad \texttt{Pa'}(\texttt{pid}, \_), \texttt{Sp'}(\texttt{spid}, \texttt{sid}, \texttt{pid}, \texttt{pid}, \texttt{sid}), \texttt{Sentence'}(\texttt{sid}, \_) \rightarrow \texttt{Sp}(\texttt{spid}, \texttt{sid}, \texttt{pid})$$
$$\texttt{Sentence'}(\texttt{sid}, \texttt{pmid}) \rightarrow \texttt{Sentence}(\texttt{sid}, \texttt{pmid})$$

Fig. 8. Residual database transformer. All variables are universally quantified.

not guarantee that the transpilation result is the most efficient, so even though one could use existing SQL query optimizers to further optimize the query, the user may want to write their hand-optimized SQL query. Second, the user may want to use a different relational schema rather than our default version. Third, while our transpilation rules allow translating Cypher to SQL, they do not address the reverse direction. Motivated by these shortcomings, our method leverages the proposed transpilation algorithm to perform verification between any given pair of Cypher and SQL queries by utilizing existing tools for SQL. In particular, the key idea is to infer a *residual database transformer* that specifies the relationship between the relational database over the induced schema and the target relational database (as specified by the user-provided database transformer). For our running example, Figure 8 shows the residual transformer that can be used to convert instances of the induced relational schema from Figure 6 to instances of the desired schema. Given such a residual schema, we can use an existing SQL equivalence checker, such as VeriEQL [26], to refute equivalence between these queries and obtain the counterexample shown in Figure 3.

3 Preliminaries

3.1 Background on Graph Databases

A graph database instance is a *property graph*, which contains nodes and edges carrying data. Typically, nodes model entities, and edges model relationships. Each node or edge in the graph stores data represented as pairs of property keys and values. Additionally, each node or edge is assigned a *label*, which describes the kind of entity or relationship it models. A well-formed property graph should conform to a graph schema, which is formalized below.

*Definition 3.1 (***Node/edge type***).* A node type t_{node} is a tuple $(l, K_1, \ldots, K_n)$ where l is the label of the node (e.g., Actor) and $K_1, \ldots, K_n$ are the property keys for that node type (e.g., `name`, `dob`, etc.). An edge type t_{edge} is also a tuple $(l, t_{\text{src}}, t_{\text{tgt}}, K_1, \ldots, K_m)$ where l is a label (e.g., ACTS_IN), t_{src} and t_{tgt} are the types of the source and target nodes respectively, and $K_1, \ldots, K_m$ are the property keys (e.g., `role`) for that edge type.

For each node type $t_{\text{node}} = (l, K_1, \ldots, K_n)$, we define $\text{label}(t_{\text{node}})$ to give the label l of t_{node}, and we assume that K_1 is the *default property key* for t_{node}, which is a key with a globally unique value, similar to a primary key in a relational database. We define $\text{keys}(t_{\text{node}}) = \{K_1, \ldots, K_n\}$ to yield the set of property keys associated with t_{node}. For an edge type $t_{\text{edge}} = (l', t_{\text{src}}, t_{\text{tgt}}, K'_1, \ldots, K'_m)$ we similarly define $\text{label}(t_{\text{edge}}) = l'$ and $\text{keys}(t_{\text{edge}}) = \{K'_1, \ldots, K'_m\}$, with key K'_1 being the default property key. Additionally, we define $\text{dstType}(t_{\text{edge}}) = t_{\text{src}}$ and $\text{srcType}(t_{\text{edge}}) = t_{\text{tgt}}$.

*Definition 3.2 (***Graph database schema***).* A graph database schema Ψ_G is a pair (T_N, T_E) where T_N is a set of node types and T_E is a set of edge types.

For each graph database schema $\Psi_G = (T_N, T_E)$, we assume that the label of each node or edge type uniquely defines it: $\forall t_1, t_2 \in T_N \cup T_E.\text{label}(t_1) \neq \text{label}(t_2)$. Thus, we can use types and labels interchangeably. Additionally, we assume that all property keys are unique inside a given schema Ψ_G, i.e., there are no name clashes between arbitrary pairs of property keys between different types.

*Definition 3.3 (***Graph database***).* An instance of a graph database schema $\Psi_G = (T_N, T_E)$ is a tuple $G = (N, E, P, T)$ where N is a set of nodes, $E \subseteq N \times N$ is a set of edges, $P : (N \cup E) \times \textbf{Keys} \rightarrow \textbf{Values}$, and $T : N \cup E \rightarrow T_N \cup T_E$ gives the type of a node $n \in N$ or an edge $e \in E$.

$$
\begin{array}{llll}
\text{Query} & Q & ::= & R \mid \mathsf{OrderBy}(R, k, b) \mid \mathsf{Union}(Q, Q) \mid \mathsf{UnionAll}(Q, Q) \\
\text{Return Query} & R & ::= & \mathsf{Return}(C, \overline{E}, \overline{k}) \\
\text{Clause} & C & ::= & \mathsf{Match}(PP, \phi) \mid \mathsf{Match}(C, PP, \phi) \mid \mathsf{OptMatch}(C, PP, \phi) \mid \mathsf{With}(C, \overline{X}, \overline{X}) \\
\text{Path Patt.} & PP & ::= & NP \mid NP, EP, PP \\
\text{Node Patt.} & NP & ::= & (X, l) \qquad \text{Edge Patt.} \quad EP ::= (X, l, d) \\
\text{Expression} & E & ::= & k \mid v \mid \mathsf{Cast}(\phi) \mid \mathsf{Agg}(E) \mid E \oplus E \\
\text{Predicate} & \phi & ::= & \top \mid \bot \mid E \odot E \mid \mathsf{IsNull}(E) \mid E \in \overline{v} \mid \mathsf{Exists}(PP) \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi
\end{array}
$$

$X \in$ **Node/Edge Names** $\quad l \in$ **Labels** $\quad k \in$ **Property Keys** $\quad v \in$ **Values** $\quad b \in$ **Bools**

$\mathsf{Agg} \in \{\mathsf{Count}, \mathsf{Avg}, \mathsf{Sum}, \mathsf{Min}, \mathsf{Max}\} \quad d \in \{\rightarrow, \leftarrow, \leftrightarrow\}$

Fig. 9. Featherweight Cypher syntax where $\oplus, \odot$ correspond to arithmetic and logical operators respectively.

We use the notation $P(n, k)$ to give the value of a property key k in node n and analogously define $P(e, k)$ for an edge e. We also use the notation $G \triangleright \Psi_G$ to denote that G is an instance of schema Ψ_G, and we refer to any subgraph of G as a *property graph*.

3.2 Query Language for Graph Databases

To formalize our method, we focus on a subset of the popular Cypher database query language [19]. This subset, which we refer to as "Featherweight Cypher", is presented in Figure 9. [3] A query Q is either a union of return queries R or an order-by statement following one or more such return queries. Each return query R takes as input a clause C, a list of expressions $\overline{E}$, and a list of property key names $\overline{k}$. Intuitively, the return query shapes a list of graphs into a table. Each clause in the return statement is a Match, representing a pattern match over an input property graph, and the patterns are specified using the path pattern PP. We do not include Cypher features such as unbounded-length path queries and graph reachability primitives (e.g., `shortestPath`), which are not expressible in the core SQL fragment considered in this paper.

Example 3.4. Consider the following Cypher query

```
MATCH (n:EMP)-[:WORK_AT]->(m:DEPT) RETURN m.dname AS name, Count(n) AS num
```

that returns a table containing department names and the number of employees. We can represent it using our featherweight Cypher syntax as follows

$$\mathsf{Return}(\mathsf{Match}([(\mathtt{n}, \mathtt{EMP}), (\mathtt{e}, \mathtt{WORK\_AT}, \rightarrow), (\mathtt{m}, \mathtt{DEPT})], \top), [\mathtt{m.dname}, \mathsf{Count}(\mathtt{n.id})], [\mathtt{name}, \mathtt{num}])$$

Here, the match clause retrieves all paths of length one from `EMP` nodes to `DEPT` nodes connected by an edge of type `WORK_AT`. Then, the return clause reshapes the set of matched paths into a table with two columns: `name` and `num`. The `name` column is populated with values corresponding to the property key `dname` of the `DEPT` node, and the `num` column is populated by the count of `n.id`, where `m` and `n` refer to the source and target nodes of the matched edge, respectively.

3.3 Relational Databases

*Definition 3.5 (***Relational schema***).* A *relational database schema* is a pair $\Psi_R := (S, \xi)$ where $S : \mathcal{R} \rightarrow [\mathcal{A}]$ is a mapping from a set of relation names $\mathcal{R}$ to a list of attributes, and ξ is an *integrity constraint*. We assume that all attribute names in a schema are unique.

We represent an integrity constraint as a conjunction of three types of *atomic* constraints:

(1) **Primary key constraints:** A primary key constraint $\mathsf{PK}(R) = a$ specifies that attribute a is the *primary key* for relation R — i.e., there cannot be multiples tuples of R that agree on a.

[3] The denotational semantics is formally described in the Appendix of the extended version [25].

$$
\begin{array}{lcll}
\text{Query} & Q & ::= & R \mid \Pi_L(Q) \mid \sigma_\phi(Q) \mid \rho_R(Q) \mid Q \cup Q \mid Q \uplus Q \mid Q \otimes Q \\
 & & \mid & \texttt{GroupBy}(Q, \overline{E}, L, \phi) \mid \texttt{With}(\overline{Q}, \overline{R}, Q) \mid \texttt{OrderBy}(Q, a, b) \\
\text{Attribute List} & L & ::= & E \mid \rho_a(E) \mid L, L \\
\text{Attribute Expr} & E & ::= & a \mid v \mid \texttt{Cast}(\phi) \mid \texttt{Agg}(E) \mid E \oplus E \\
\text{Predicate} & \phi & ::= & b \mid E \odot E \mid \texttt{IsNull}(E) \mid E \in \overline{v} \mid \overline{E} \in Q \mid \phi \wedge \phi \mid \phi \vee \phi \mid \neg\phi \\
\text{Join Op} & \otimes & ::= & \times \mid \bowtie_\phi \mid \rightarrowtail\!\!\!\!\!\!\bowtie_\phi \mid \rightarrowtail\!\!\!\!\!\!\bowtie_\phi \mid \bowtie\!\!\!\!\!\!\leftarrowtail_\phi
\end{array}
$$

$R \in$ **Relation Names** $\quad a \in$ **Attr Names** $\quad v \in$ **Values** $\quad b \in$ **Bools** $\quad \texttt{Agg} \in \{\texttt{Count}, \texttt{Avg}, \texttt{Sum}, \texttt{Min}, \texttt{Max}\}$

Fig. 10. Featherweight SQL syntax; $\oplus$ and $\odot$ represent arithmetic and logical operators respectively

$$
\begin{array}{lcll}
\text{Transformer} & \Phi & ::= & P, \ldots, P \rightarrow P \mid \Phi\ \Phi \\
\text{Predicate} & P & ::= & E(t, \ldots, t) \\
\text{Term} & t & ::= & c \mid v \mid \_
\end{array}
$$

$E \in$ **Table Names** $\cup$ **Node Labels** $\cup$ **Edge Labels** $\quad c \in$ **Constants** $\quad v \in$ **Variables**

Fig. 11. Syntax of the database transformer.

(2) **Foreign key constraints:** A foreign key constraint $\mathsf{FK}(R.a) = R'.a'$ specifies that the attribute a in relation R is a *foreign key* corresponding to attribute a' in relation R'. That is, the values stored in attribute a of relation R must be a subset of the values stored in attribute a' of R'.
(3) **Not-null constraints:** A not-null constraint $\mathsf{NotNull}(R, a)$ specifies that the value stored at attribute a of relation R must not be Null.

*Definition 3.6 (***Relational database instance***).* A relational database instance R is a collection of tuples $\{r_1, \ldots, r_m\}$, where each $r_i \in R$ is of the form $(a_1 : v_1, \ldots, a_n : v_n)$. Here, $a_1, \ldots, a_n$ are attributes and $v_1, \ldots, v_n$ are values. We let $\mathrm{Attrs}(r_i)$ return the list of attributes $a_1, \ldots, a_n$ in sequence. We use the notation $r_i.a$ to denote the value stored in attribute a of tuple r_i.

As with graph databases, we use the notation $R \triangleright \Psi_R$, to denote that R is instance of Ψ_R.

SQL query language. In this paper, we consider relational database queries written in SQL. Figure 10 shows the subset of SQL that we use in our formalization. At a high level, this language extends core relational algebra (e.g., projection Π, selection σ, renaming ρ, joins $\otimes$, set and bag unions $\cap, \uplus$) to incorporate `GroupBy`, `OrderBy`, and `With` clauses. It can express a representative fragment of SQL queries that are commonly used in practice. The semantics of these SQL operators are standard and formally defined by prior work such as He et al. [26].

4 Problem Statement

In this section, we first describe the language for database transformers and then formally define the equivalence checking problem between graph and relational databases.

4.1 Language for Database Transformers

In this section, we present a small domain-specific language (DSL), shown in Figure 11, for expressing database transformers. Following prior work [59], our DSL generalizes the standard concept of *schema mapping* [17, 37] for relational databases to a more flexible form. In particular, a database transformer in this DSL is expressed as a set of first-order formulas of the form $P_1, \ldots, P_n \rightarrow P_0$, where each P_i is a predicate that represents a database element, such as a table in a relational database or a node or edge in a graph database. Each predicate is of the form $E(t_1, \ldots, t_n)$ where E corresponds a table name, node labels, or edge labels, and each t_j is a term (variable or a constant), with _ denoting a fresh variable that is not used. All variables are implicitly universally quantified but the quantifiers are omitted in the syntax for brevity. Intuitively, the formula $P_1, \ldots, P_n \rightarrow P_0$ expresses that, if predicates $P_1, \ldots, P_n$ hold over a database instance D, then predicate P_0 holds over another database instance D'.

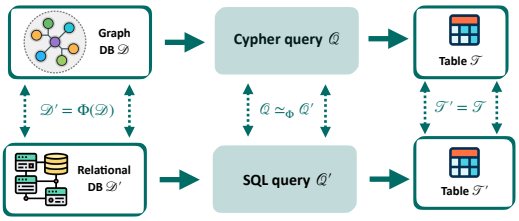


Fig. 12. Definition of equivalence between Cypher and SQL queries where Φ is the user-provided schema transformer. A pair of Cypher and SQL queries are equivalent if they produce the same table (modulo renaming) whenever they are executed on a pair of database instances satisfying Φ.

Semantics. To define the semantics of our transformer DSL, we first represent a database transformer Φ as a set of universally quantified first-order logic formulas, denoted as $[\![\Phi]\!]$. The idea behind the semantics of our DSL is to represent each database instance as a set of ground predicates and then check whether these predicates entail the first-order logic formula $[\![\Phi]\!]$ under the Herbrand semantics.

To make this discussion more precise, we introduce a function C that maps a database instance to a set of ground predicates representing its structure and contents. Formally, C is defined as follows:

$$C(D) = \{E(t_1, \ldots, t_n) \mid E \in D\}$$

The mapping of elements in D to predicates depends on whether D is a relational or graph database:

- **For relational database instance D:** If R is a table in D with a set of records $\{(a_1, \ldots, a_n)\}$, then R is converted to the following set of ground predicates:

  $$\{R(a_1, \ldots, a_n) \mid (a_1, \ldots, a_n) \in R\}$$

  Here, R represents the table name, and each a_i is a constant representing a value in the record.
- **For graph database instance D:** If $N(l, a_1, \ldots, a_n)$ denotes a node with label l and values $a_1, \ldots, a_n$ of property keys $K_1, \ldots, K_n$, nodes are converted to the following set of ground facts:

  $$\{l(a_1, \ldots, a_n) \mid \text{node } N(l, a_1, \ldots, a_n) \in D\}$$

  Similarly, if $E(l, s, t, a_1, \ldots, a_n)$ denotes an edge with label l that connects nodes s and t and has property values $a_1, \ldots a_n$, then edges are converted to predicates as follows:

  $$\{l(a_1, \ldots, a_n, s, t) \mid \text{edge } E(l, s, t, a_1, \ldots, a_n) \in D\}$$

Given a database instance D, $C(D)$ yields a set of ground predicates representing the structure and contents of D. We can now define the semantics of the transformer Φ as follows:

$$\Phi(D) = D' \quad \Leftrightarrow \quad C(D) \cup C(D') \models [\![\Phi]\!]$$

where the notation $S \models \varphi$ indicates that the set S of ground predicates is a Herbrand model of φ.

Example 4.1. Consider the graph and relational database instances G, R from Figures 3a and 3b respectively. For the transformer Φ shown in Figure 5, we have $\Phi(G) = R$.

4.2 Equivalence Checking Problem

In this section, we formally define what it means for a pair of graph and relational queries to be equivalent modulo a database transformer Φ, expressed in the DSL of Section 4.1. To this end, we first introduce some necessary definitions and notations.

*Definition 4.2 (***Database query***).* A database query Q over schema Ψ takes as input a database instance D such that $D \triangleright \Psi$ and yields a table. We denote the semantics of Q as $[\![Q]\!]_D$.

The definition above is sufficiently general to describe both SQL and Cypher queries, as both query languages return a table.

Algorithm 1 Methodology for checking equivalence between Cypher and SQL queries

1: **procedure** CheckEquivalence($\Psi_G, Q_G, \Psi_R, Q_R, \Phi$)
Input: Graph and relational schemas Ψ_G, Ψ_R, Cypher query Q_G, SQL query Q_R, transformer Φ
Output: $\top$ for equivalence or $\bot$ indicating failure
2: $(\Phi_{\text{sdt}}, \Psi'_R) \leftarrow$ InferSDT(Ψ_G)
3: $Q_{R'} \leftarrow$ Transpile($Q_G, \Phi_{\text{sdt}}, \Psi'_R$)
4: **return** ReduceToSQL($\Psi_R, Q_R, \Psi'_R, Q'_R, \Phi, \Phi_{\text{sdt}}$)

*Definition 4.3 (***Database equivalence)**. Let D, D' be database instances over schemas Ψ, Ψ' respectively and let Φ be a database transformer that can be used to convert instances of Ψ to instances of Ψ'. Then, D is said to be equivalent to D' modulo Φ, denoted $D \sim_\Phi D'$, if $\Phi(D) = D'$.

According to the above definition, a graph database instance G is equivalent to a relational database instance R if the contents of R can be obtained from G by applying the transformer Φ to G. Next, to define equivalence between graph and relational queries, we need to define what it means for the query outputs to be the same. Since queries in both Cypher and SQL return tables, we need a notion of equivalence between tables.

*Definition 4.4 (***Table equivalence)**. Two tables T and T' are said to be equivalent, denoted $T \equiv T'$, if there exists a bijective mapping π from columns of T to those of T' such that, for each tuple $r \in T$ with multiplicity n, there exists a unique tuple $r' \in T'$ with multiplicity n where $\forall a \in \text{Attrs}(r).\ r.a = r'.\pi(a)$.

In other words, our notion of table equivalence disregards the order of attributes as well as their names, which allows for a more robust notion of query equivalence.[4]

*Definition 4.5 (***Graph-relational query equivalence)**. Let Ψ_G and Ψ_R be graph and relational schemas respectively, and Φ be a transformer from Ψ_G to Ψ_R. A query Q over Ψ_G is *equivalent* to Q' over Ψ_R modulo Φ, denoted $Q \simeq_\Phi Q'$, iff:

$$\forall G, R.\ \ (G \triangleright \Psi_G \wedge R \triangleright \Psi_R \wedge G \sim_\Phi R) \implies [\![Q]\!]_G \equiv [\![Q']\!]_R$$

In other words, Q, Q' are considered equivalent if they produce the same tables (modulo renaming/ re-ordering of columns) when executed on a pair of database instances G, R satisfying $G \sim_\Phi R$. Figure 12 visualizes this definition of equivalence between graph databases and relational databases.

*Definition 4.6 (***Equivalence checking problem)**. Given graph and relational queries Q, Q' and a transformer Φ from graph schema Ψ_G to relational schema Ψ_R, the equivalence checking problem is to decide whether $Q \simeq_\Phi Q'$.

5 Equivalence Checking Algorithm

In this section, we present our algorithm, summarized in Algorithm 1, for checking equivalence between Cypher and SQL queries. As shown in Algorithm 1, our algorithm consists of three steps:

(1) **Schema and transformer inference:** Given the graph database schema Ψ_G, our algorithm first invokes the InferSDT function to infer both the induced relational schema $\Psi'_R = (S, \xi)$ as well as the standard database transformer Φ_{sdt}.

[4]If a query includes an `OrderBy` clause, list semantics will be applied, where the result is an ordered list of tuples. In this case, the equivalence of two tables T and T' is defined such that there exists a bijective mapping π between their attributes, and for each pair of tuples $r \in T$ and $r' \in T'$ at the same index, it holds that $\forall a \in \text{Attrs}(r).\ r.a = r'.\pi(a)$.

$$\frac{t_{\text{node}} = (l, K_1, \ldots, K_n) \quad \xi = (\mathsf{PK}(R_l) = K_1) \quad \Phi = \{l(K_1, \ldots, K_n) \rightarrow R_l(K_1, \ldots, K_n)\}}{t_{\text{node}} \hookrightarrow (\{R_l \mapsto (K_1, \ldots, K_n)\}, \xi, \Phi)} \text{ (Node)}$$

$$\frac{\begin{array}{c} t_{\text{edge}} = (l, t_{\text{src}}, t_{\text{tgt}}, K_1, \ldots, K_m) \quad \mathsf{label}(t_{\text{src}}) = s \quad \mathsf{label}(t_{\text{tgt}}) = t \\ \xi = \mathsf{PK}(R_l) = R_l.K_1 \wedge \mathsf{FK}(R_l.\mathsf{fk}_s) = \mathsf{PK}(R_s) \wedge \mathsf{FK}(R_l.\mathsf{fk}_t) = \mathsf{PK}(R_t) \\ \Phi = \{l(K_1, \ldots, K_m, \mathsf{fk}_s, \mathsf{fk}_t) \rightarrow R_l(K_1, \ldots, K_m, \mathsf{fk}_s, \mathsf{fk}_t)\} \end{array}}{t_{\text{edge}} \hookrightarrow (\{R_l \mapsto (K_1, \ldots, K_m, \mathsf{fk}_s, \mathsf{fk}_t)\}, \xi, \Phi)} \text{ (Edge)}$$

$$\frac{T_1 \hookrightarrow (S_1, \xi_1, \Phi_1) \quad T_1 \hookrightarrow (S_2, \xi_2, \Phi_2)}{T_1 \uplus T_2 \hookrightarrow (S_1 \uplus S_2, \xi_1 \wedge \xi_2, \Phi_1 \cup \Phi_2)} \text{ (Set)} \qquad \frac{(T_N \uplus T_E) \hookrightarrow (S, \xi, \Phi)}{(T_N, T_E) \hookrightarrow (S, \xi, \Phi)} \text{ (Schema)}$$

Fig. 13. Rules for the InferSDT procedure. R'_l denotes the table name of R_l in the induced relational schema.

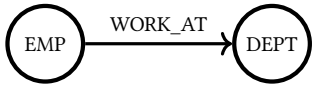
(2) **Syntax-directed transpilation:** Next, our algorithm uses the inferred database transformer and integrity constraints to transpile the Cypher query into an SQL query Q'_R that is guaranteed to be equivalent to Q_G modulo the standard Φ_{sdt}.
(3) **Checking SQL equivalence:** Finally, the algorithm computes a residual database transformer Φ_{rdt} that can be used to convert instances of Ψ'_R into instances of Ψ_R and checks equivalence between SQL queries Q_R and Q'_R modulo the residual database transformer Φ_{rdt} relating a pair of database schemas.

Discussion. An alternative approach to solving the equivalence checking problem would be to directly reason about equivalence between the graph query Q_G and the relational query Q_R. While such an approach would be more direct, we adopt the methodology illustrated in Figure 1 for several reasons. First, in order to directly verify equivalence between graph and relational database queries, we need suitable SMT encodings of *both* graphs and relations, which makes the resulting constraint solving problem harder compared to the alternative. Second, the reduction to relational equivalence checking allows us to leverage a variety of existing tools that have been developed for SQL, including testing tools [6], bounded model checkers [26], and deductive verifiers [57].

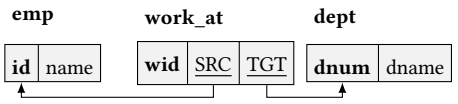
5.1 Induced Relational Schema and Standard Transformer Inference

We first discuss the InferSDT procedure for inferring the induced relational schema as well as the standard database transformer. This procedure is formalized in Figure 13 using inference rules of the form: $\Psi_G \hookrightarrow (S, \xi, \Phi_{\text{sdt}})$ where Ψ_G corresponds to elements of the graph schema (nodes, edges, and subgraphs), $\Psi_R = (S, \xi)$ is the induced relational schema for Ψ_G, and Φ_{sdt} is the standard database transformer that can be used to convert instances of Ψ_G into instances of Ψ_R.

Induced relational schema. Intuitively, the induced relational schema is the "closest" relational representation of the graph database schema $\Psi_G = (T_N, T_E)$ that represents each node and edge type as a relational table. As shown in the Node rule, for each node type $t_{\text{node}} = (l, K_1, \ldots, K_n) \in T_N$ with label l and default property key K_1, we introduce a table R_l with attributes $K_1, \ldots, K_n$ in the induced relational schema. We also use the default property key K_1 as the primary key of the corresponding table and generate an integrity constraint $\mathsf{PK}(R_l) = K_1$. Similarly, as shown in the Edge rule, for each edge type $t_{\text{edge}} = (l, t_{\text{src}}, t_{\text{tgt}}, K_1, \ldots, K_m) \in T_E$ with default property key K_1, we introduce a table R_l with attributes K_1, ..., K_m, fk_s, fk_t. Here, K_1 is also the primary key of table R_l, and fk_s, fk_t are foreign keys which reference to primary keys of the tables corresponding to source and target nodes. Thus, the integrity constraint is $\mathsf{PK}(R_l) = R_l.K_1 \wedge \mathsf{FK}(R_l.\mathsf{fk}_s) = \mathsf{PK}(R_s) \wedge \mathsf{FK}(R_l.\mathsf{fk}_t) = \mathsf{PK}(R_t)$, where R_s, R_t are the tables corresponding to the source and target nodes.

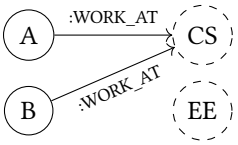


(a) Graph schema. Node labels are inside the nodes. Edge labels are above the arrows. EMP has property keys: **id**, name. DEPT has property keys: **dnum**, dname. WORK_AT has a property key: **wid**. The default property keys are in bold.



(b) Induced relational schema. Primary keys are in bold. Foreign keys are underlined and connected to references with arrows.

Fig. 14. Example of a graph schema and its induced relational schema.



(a) A graph database instance.

emp

id	name
1	A
2	B

work_at

wid	SRC	TGT
10	1	1
11	2	1

dept

dnum	dname
1	CS
2	EE

(b) The relational tables.

Fig. 15. Example graph database and its corresponding relational database over the induced relational schema.

Example 5.1. Consider the graph schema Ψ_G shown in Figure 14a. Its induced relational schema Ψ_R is visualized in Figure 14b.

Standard database transformer. The standard database transformer (SDT) is expressed in the same language as a general database transformer from Section 4.1, which transforms instances of a graph schema Ψ_G to instances of a relational schema Ψ_R. Intuitively, the standard database transformer for a graph schema Ψ_G converts each node and edge type to a separate table, and all occurrences of that element type in a graph database G become tuples in the corresponding table of the relational database. Specifically, as shown in the Node rule, for each node of type $t_{\text{node}} = (l, K_1, \ldots, K_n)$, we generate a formula $l(K_1, \ldots, K_n) \longrightarrow R_l(K_1, \ldots, K_n)$ which transforms a predicate $R_l(v_1, \ldots, v_n)$ representing a graph element to a predicate $R'_l(v_1, \ldots, v_n)$ representing a tuple in the relational database. Similarly, we generate a formula $l(K_1, \ldots, K_m, \mathsf{fk}_s, \mathsf{fk}_t) \longrightarrow R_l(K_1, \ldots, K_m, \mathsf{fk}_s, \mathsf{fk}_t)$ for each edge type in the graph schema.

Example 5.2. Consider the graph database G visualized in Figure 15a. The SDT Φ_{sdt} transforms G to the relational database shown in Figure 15b.

5.2 Syntax-Directed Transpilation

Building upon the standard database transformer (SDT) introduced earlier, we now turn to the core task of translating Cypher queries into corresponding SQL queries over the induced schema. This process presents several challenges, as it involves mapping graph-based operations in Cypher to the relational model in SQL, while ensuring that the original query semantics are preserved. Specifically, Cypher includes features like pattern matching over subgraphs and optional matches, which do not have direct equivalents in SQL. Additionally, Cypher aggregates data over matched subgraphs, whereas SQL aggregates over grouped tuples. Finally, ensuring consistent mappings of graph nodes and edges across the query is critical to maintaining the integrity of references throughout the process. Our syntax-directed transpilation approach addresses these challenges by converting Cypher queries into SQL queries over the induced relational schema. The key idea is that path patterns in Cypher can be represented by relational joins in SQL. For instance, Cypher's match clauses are translated into SQL inner joins, and optional match clauses map to outer joins. This approach ensures that the pattern matching semantics of Cypher queries are faithfully represented within the relational model, maintaining the integrity of the original graph-based operations. We

$$\frac{\neg\mathsf{hasAgg}(\overline{E}) \quad \Phi_{\mathrm{sdt}}, \Psi_R \vdash C \xrightarrow{\text{clause}} \mathcal{X}, Q \quad \Phi_{\mathrm{sdt}}, \Psi_R \vdash E_i \xrightarrow{\text{expr}} E_i' \quad 1 \le i \le |\overline{E}|}{\Phi_{\mathrm{sdt}}, \Psi_R \vdash \mathsf{Return}(C, \overline{E}, \overline{k}) \xrightarrow{\text{query}} \Pi_{\rho_{\overline{k}}(\overline{E'})}(Q)} \text{ (Q-Ret)}$$

$$\frac{\mathsf{hasAgg}(\overline{E}) \quad \Phi_{\mathrm{sdt}}, \Psi_R \vdash E_i \xrightarrow{\text{expr}} E_i' \quad 1 \le i \le |\overline{E}| \quad \Phi_{\mathrm{sdt}}, \Psi_R \vdash C \xrightarrow{\text{clause}} \mathcal{X}, Q \quad \overline{A} = \mathsf{filter}(\lambda x.\neg\mathsf{IsAgg}(x), \overline{E'})}{\Phi_{\mathrm{sdt}}, \Psi_R \vdash \mathsf{Return}(C, \overline{E}, \overline{k}) \xrightarrow{\text{query}} \mathsf{GroupBy}(Q, \overline{A}, \rho_{\overline{k}}(\overline{E'}), \top)} \text{ (Q-Agg)}$$

$$\frac{\Phi_{\mathrm{sdt}}, \Psi_R \vdash Q \xrightarrow{\text{query}} Q' \quad \Phi_{\mathrm{sdt}}, \Psi_R \vdash k \xrightarrow{\text{expr}} a}{\Phi_{\mathrm{sdt}}, \Psi_R \vdash \mathsf{OrderBy}(Q, k, b) \xrightarrow{\text{query}} \mathsf{OrderBy}(Q', a, b)} \text{ (Q-OrderBy)}$$

$$\frac{\Phi_{\mathrm{sdt}}, \Psi_R \vdash Q_1 \xrightarrow{\text{query}} Q_1' \quad \Phi_{\mathrm{sdt}}, \Psi_R \vdash Q_2 \xrightarrow{\text{query}} Q_2'}{\Phi_{\mathrm{sdt}}, \Psi_R \vdash \mathsf{Union}(Q_1, Q_2) \xrightarrow{\text{query}} Q_1' \cup Q_2'} \text{ (Q-Union)} \qquad \frac{\Phi_{\mathrm{sdt}}, \Psi_R \vdash Q_1 \xrightarrow{\text{query}} Q_1' \quad \Phi_{\mathrm{sdt}}, \Psi_R \vdash Q_2 \xrightarrow{\text{query}} Q_2'}{\Phi_{\mathrm{sdt}}, \Psi_R \vdash \mathsf{UnionAll}(Q_1, Q_2) \xrightarrow{\text{query}} Q_1' \uplus Q_2'} \text{ (Q-UnionAll)}$$

Fig. 16. Translation rules for queries.

now describe a core subset of our syntax-directed transpilation rules, with further details available in the Appendix of the extended version [25].

Translating queries. The translation rules for queries, illustrated in Figure 16, use judgments of the form $\Phi_{\mathrm{sdt}}, \Psi_R \vdash Q \xrightarrow{\text{query}} Q'$, where a Cypher query Q maps to an SQL query Q' given SDT Φ_{sdt} and induced database transformer Ψ_R. Among these rules, handling Return requires particular attention, as it involves distinguishing between cases with and without aggregate functions. If there are no aggregation functions in $\overline{E}$, the Q-Ret rule produces a straightforward translation: the Cypher query $\mathsf{Return}(C, \overline{E}, \overline{k})$ becomes a simple SQL projection $\Pi_{\rho_{\overline{k}}(\overline{E'})}(Q)$. Here, Q is the translated result of the Cypher clause C, and $\overline{E'}$ represents the translated expressions of $\overline{E}$, with all attributes renamed to $\overline{k}$. In contrast, when aggregation functions appear in $\overline{E}$, the translation shifts to the Q-Agg rule, which requires generating a GroupBy query. This is necessary because SQL uses GroupBy to manage aggregation by partitioning rows based on non-aggregated columns. In the Cypher query $\mathsf{Return}(C, \overline{E}, \overline{k})$, the non-aggregation expressions $\overline{A}$ act as grouping keys, while the aggregated expressions ensure the correct computation of results for each group.

Example 5.3. Consider the following Cypher query and the SDT from Example 5.2:

$$\mathsf{Return}(\mathsf{Match}([(\mathsf{n}, \mathsf{EMP}), (\mathsf{e}, \mathsf{WORK\_AT}, \rightarrow), (\mathsf{m}, \mathsf{DEPT})], \top), [\mathtt{m.dname}, \mathsf{Count}(\mathtt{n.id})], [\mathsf{name}, \mathsf{num}])$$

Since there is a Count aggregation in the return query, we apply the Q-Agg rule to translate it to a GroupBy query in SQL. Specifically, we first apply the C-Match1 rule to translate the match clause $\mathsf{Match}([(\mathsf{n}, \mathsf{EMP}), (\mathsf{e}, \mathsf{WORK\_AT}, \rightarrow), (\mathsf{m}, \mathsf{DEPT})], \top)$ into a SQL query Q (explained in Example 5.4). Then we translate the returned Cypher expressions m.dname and Count(n.id) to their corresponding SQL expressions m.dname and Count(n.id). Among these expressions, we find the one that does not contain aggregations, namely m.dname, and use it as the grouping key for GroupBy. Since there is no filtering based on the aggregated results, the Having clause for GroupBy is not generated. Therefore, the translated SQL query is $\mathsf{GroupBy}(Q, [\mathtt{m.dname}], \rho_{[\mathsf{name},\mathsf{num}]}([\mathtt{m.dname}, \mathsf{Count}(\mathtt{n.id})]), \top)$.

Translating Clauses. Unlike translating entire queries, translating individual clauses requires tracking additional information about the node and edge variables used within the clauses. This is necessary to ensure that multiple occurrences of the same variable across different clauses are translated to refer to the same tuple in SQL. As shown in Figure 17, our translation judgments are of the form $\Phi_{\mathrm{sdt}}, \Psi_R \vdash C \xrightarrow{\text{clause}} \mathcal{X}, Q'$, meaning a Cypher clause C is translated into a SQL query Q', and $\mathcal{X}$ is the set of all used node and edge variables.

Our translation rules for the Match and OptMatch clauses are based on the observation that graph pattern matching in Cypher can be emulated using sequences of join operations in SQL.

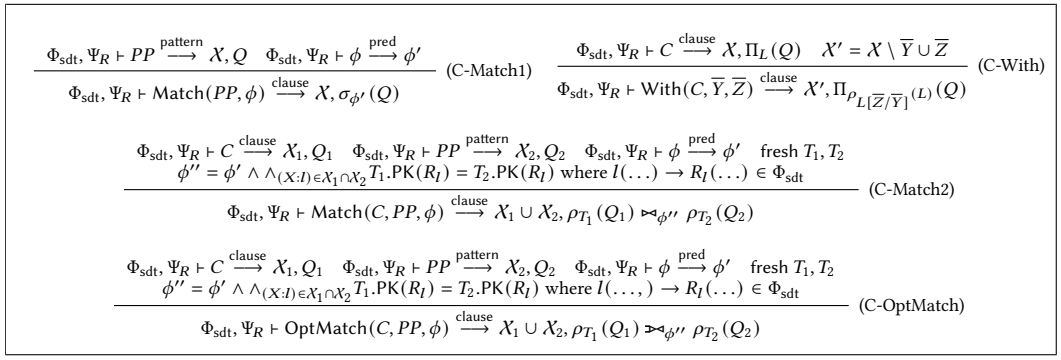


Fig. 17. Translation rules for clauses.

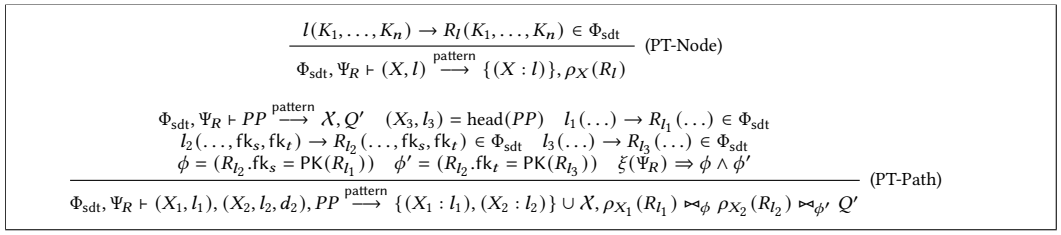


Fig. 18. Translation rules for path patterns.

Specifically, the join operations for Match are inner joins, while those for OptionalMatch are left outer joins. Intuitively, a Match clause returns no results if there is no matching pattern, mirroring the behavior of inner joins where unmatched tuples are discarded. In contrast, an OptionalMatch clause returns null for missing matches, similar to how outer joins include unmatched rows with null values.

For example, consider the clause $\mathsf{Match}(C, PP, \phi)$. The translation rule C-Match2 first translates the preceding clause C into a subquery Q_1 and the path pattern PP into another subquery Q_2. It also collects the sets of node and edge variables used in C and PP, denoted as $\mathcal{X}_1$ and $\mathcal{X}_2$, respectively. For each common variable X with label l, the translation generates a join predicate $T_1.K_1 = T_2.K_1$. This ensures that occurrences of the same variable in different parts of the clause are correctly matched by joining on their primary keys, effectively referring to the same tuple in the SQL translation.

The C-With rule handles the With clause by translating $\mathsf{With}(C, \overline{Y}, \overline{Z})$ into a renaming operation in SQL. It generates a query $\Pi_{\rho_{L[\overline{Z}/\overline{Y}]}(L)}(Q)$, which projects and renames columns, replacing the old names $\overline{Y}$ with the new names $\overline{Z}$.

Example 5.4. Consider the Cypher clause $\mathsf{Match}([(\mathtt{n}, \mathtt{EMP}), (\mathtt{e}, \mathtt{WORK\_AT}, \rightarrow), (\mathtt{m}, \mathtt{DEPT})], \top)$ and the Φ_{sdt} from Example 5.2. Based on C-Match1, we use the PT-Path rule to collect all node and edge variables in the pattern, namely $\mathcal{X} = \{(\mathtt{n} : \mathtt{EMP}), (\mathtt{e} : \mathtt{WORK\_AT}), (\mathtt{m} : \mathtt{DEPT})\}$, and translate it to a SQL query $\rho_{\mathtt{n}}(\mathtt{emp}) \bowtie_{\mathtt{n.id=e.SRC}} \rho_{\mathtt{e}}(\mathtt{work\_at}) \bowtie_{\mathtt{e.TGT=m.dnum}} \rho_{\mathtt{m}}(\mathtt{dept})$. Thus, the Cypher clause is translated to $\sigma_{\top}(\rho_{\mathtt{n}}(\mathtt{emp}) \bowtie_{\mathtt{n.id=e.SRC}} \rho_{\mathtt{e}}(\mathtt{work\_at}) \bowtie_{\mathtt{e.TGT=m.dnum}} \rho_{\mathtt{m}}(\mathtt{dept}))$.

Example 5.5. Consider the Cypher clause $\mathsf{OptMatch}(C, PP, \phi)$ where C is the Match clause in Example 5.4, $PP = [(\mathtt{m}, \mathtt{DEPT})]$ and the Φ_{sdt} from Example 5.2. Based on C-Match1 and C-OptMatch, we know $\mathcal{X}_1 = \{(\mathtt{n} : \mathtt{EMP}), (\mathtt{e} : \mathtt{WORK\_AT}), (\mathtt{m} : \mathtt{DEPT})\}$ and $\mathcal{X}_2 = \{(\mathtt{m} : \mathtt{DEPT})\}$. C and PP are translated to $Q_1 = \sigma_{\top}(\rho_{\mathtt{n}}(\mathtt{emp}) \bowtie_{\mathtt{n.id=e.SRC}} \rho_{\mathtt{e}}(\mathtt{work\_at}) \bowtie_{\mathtt{e.TGT=m.dnum}} \rho_{\mathtt{m}}(\mathtt{dept}))$ and $Q_2 = \sigma_{\phi}(\rho_{\mathtt{m}}(\mathtt{dept}))$, respectively. Since there is a shared node $(\mathtt{m}, \mathtt{DEPT})$ between C and PP, and the primary key of dept is dnum, the Cypher clause is translated to $\rho_{T_1}(Q_1) \rightarrowtriangle_{T_1.\mathtt{dnum}=T_2.\mathtt{dnum}} \rho_{T_2}(Q_2)$.

Algorithm 2 Algorithm for inferring the residual of database transformers and SDT's

1: **procedure** ReduceToSQL($\Psi_R, Q_R, \Psi'_R, Q'_R, \Phi, \Phi_{\text{sdt}}$)
Input: Database transformer Φ, standard database transformer Φ_{sdt}
Output: $\top$ for equivalence or $\bot$ indicating failure
2: $\sigma \leftarrow \{P_1 \mapsto P_0 \mid P_1(\ldots) \rightarrow P_0(\ldots) \in \Phi_{\text{sdt}}\}$
3: $\Phi_{\text{rdt}} \leftarrow \Phi[\sigma]$
4: **return** CheckSQL($\Psi_R, Q_R, \Psi'_R, Q'_R, \Phi_{\text{rdt}}$)

Translating patterns. Figure 18 illustrates the rules for translating patterns, using judgments of the form $\Phi_{\text{sdt}}, \Psi_R \vdash PP \overset{\text{pattern}}{\longrightarrow} \mathcal{X}, Q'$. This notation indicates that a Cypher pattern PP translates to an SQL query Q', with all node and edge variables in the pattern represented by $\mathcal{X}$.

The translation of patterns follows an inductive structure, with two key rules. The base case handles a single node pattern using the PT-Node rule, which maps the node variable X to its corresponding table R'_l in the relational schema derived from SDT Φ_{sdt}, renaming it to X. The inductive case, represented by the PT-Path rule, addresses more complex patterns where a new node (X_2, l_2) expands an existing sub-pattern PP. The rule identifies the connecting node (X_3, l_3) in PP and determines the appropriate joins. It locates the tables $R'_{l_1}, R'_{l_2}, R'_{l_3}$ corresponding to nodes X_1, X_2, and X_3, and constructs join predicates to connect the foreign keys fk_s and fk_t in the edge table R'_{l_2} to the primary keys in the source and target node tables, respectively. This approach aligns with the observation that Cypher's pattern matching naturally corresponds to a series of SQL join operations.

Example 5.6. Given the standard database transformer Φ in Example 5.2, let us focus on the path pattern $[(\mathtt{n}, \mathtt{EMP}), (\mathtt{e}, \mathtt{WORK\_AT}, \rightarrow), (\mathtt{m}, \mathtt{DEPT})]$. According to the PT-Path rule, we first need to apply the PT-Node rule to get the variables $\{(\mathtt{m} : \mathtt{DEPT})\}$ and query $\rho_{\mathtt{m}}(\mathtt{DEPT})$ from the node pattern $(\mathtt{m}, \mathtt{DEPT})$. Based on these results, we can use the PT-Path rule to collect all variables $\mathcal{X} = \{(\mathtt{n} : \mathtt{EMP}), (\mathtt{e} : \mathtt{WORK\_AT}), (\mathtt{m} : \mathtt{DEPT})\}$ and obtain the SQL query $\rho_{\mathtt{n}}(\mathtt{emp}) \bowtie_{\mathtt{n.id=e.SRC}} \rho_{\mathtt{e}}(\mathtt{work\_at}) \bowtie_{\mathtt{e.TGT=m.dnum}} \rho_{\mathtt{m}}(\mathtt{dept})$.

Theorem 5.7 (Soundness of translation). *Let Ψ_G be a graph schema and Q be a Cypher query over Ψ_G. Let Ψ_R be the induced relational schema of Ψ_G, and Φ_{sdt} be the standard database transformer from Ψ_G to Ψ_R. If $\Phi_{\text{sdt}}, \Psi_R \vdash Q \overset{\text{query}}{\longrightarrow} Q'$, then Q' is equivalent to Q modulo Φ_{sdt}, i.e., $Q \simeq_{\Phi_{\text{sdt}}} Q'$.*

Theorem 5.8 (Completeness of translation). *Let Ψ_G be a graph schema and Ψ_R be the induced relational schema of Ψ_G. Given any Cypher query Q over Ψ_G accepted by the grammar shown in Figure 9, there exists a SQL query Q' over Ψ_R such that $\Phi_{\text{sdt}}, \Psi_R \vdash Q \overset{\text{query}}{\longrightarrow} Q'$.*

5.3 Reduction to SQL Equivalence Checking

The final step of our algorithm utilizes the transpiled query to reduce our original problem to checking equivalence between a pair of SQL queries. As shown in Algorithm 2, the ReduceToSQL procedure first infers the residual database transformer Φ_{rdt} through a simple syntactic substitution: Since every clause of the SDT is of the form $P_1(\ldots) \rightarrow P_0(\ldots)$, we can obtain the residual transformer simply by substituting every occurrence of P_1 in Φ by P_0. Finally, since the residual transformer Φ_{rdt} specifies how to convert instances of the induced schema to those of the desired schema, we can use an existing tool for checking SQL equivalence by utilizing Φ_{rdt}. As stated by the following theorems, the original Cypher query is equivalent to the given SQL query if and only if Q_R and Q'_R are equivalent modulo Φ_{rdt}.

Table 1. Statistics of Cypher and SQL queries in the benchmarks. Sizes are the number of AST nodes.

Dataset	#	SQL Size				Cypher Size				Transformer Size			
		Min	Max	Avg	Med	Min	Max	Avg	Med	Min	Max	Avg	Med
StackOverflow	12	15	74	32.5	28	20	149	54.9	41	1	6	3.3	4
Tutorial	26	5	76	25.8	22	12	77	31.2	28	1	17	8.3	5
Academic	7	27	59	46.9	54	45	121	75.0	66	5	7	6.7	7
VeriEQL	60	15	143	42.2	39	29	174	65.8	61	1	10	6.7	10
Mediator	100	9	63	18.6	13	20	114	33.8	28	1	11	5.1	4
GPT-Translate	205	5	143	28.2	25	12	171	46.0	38	1	17	5.9	5
Total	410	5	143	28.2	25	12	174	45.7	38	1	17	5.9	5

Theorem 5.9 (Soundness). *Let CheckSQL*$(\Psi_R, Q, \Psi'_R, Q', \Phi_{\text{rdt}})$ *be a sound procedure for equivalence checking of SQL queries* Q, Q' *over relational schemas* Ψ_R, Ψ'_R *connected by RDT* Φ_{rdt}*. Given a Cypher query* Q_G *over graph schema* Ψ_G*, a SQL query* Q_R *over relational schema* Ψ_R*, and their database transformer* Φ*, if* CheckEquivalence$(\Psi_G, Q_G, \Psi_R, Q_R, \Phi)$ *returns* $\top$*, it holds that* $Q_G \simeq_\Phi Q_R$*.*

Theorem 5.10 (Completeness). *Let CheckSQL*$(\Psi_R, Q, \Psi'_R, Q', \Phi_{\text{rdt}})$ *be a complete procedure for equivalence checking of SQL queries* Q, Q' *over schemas* Ψ_R, Ψ'_R *connected by RDT* Φ_{rdt}*. Given a Cypher query* Q_G *over graph schema* Ψ_G*, a SQL query* Q_R *over relational schema* Ψ_R*, and their database transformer* Φ*, if* $Q_G \simeq_\Phi Q_R$*, then* CheckEquivalence$(\Psi_G, Q_G, \Psi_R, Q_R, \Phi)$ *returns* $\top$*.*

6 Evaluation

In this section, we describe three experiments to evaluate Graphiti. Because Graphiti's verification methodology reduces the Cypher-SQL equivalence checking problem to pure SQL, our results depend on what backend is used for SQL equivalence checking. Thus, in our first experiment, we evaluate Graphiti using the VeriEQL [26] bounded model checker as its backend, and in our second experiment, we use a deductive verifier called Mediator [57] as the backend. Finally, we also evaluate the quality of Graphiti's transpilation results.

Benchmarks. We evaluate Graphiti on 410 pairs of SQL and Cypher queries (see Table 1) from the following sources:

- `StackOverflow`: We identified 12 StackOverflow posts where users inquire about translating a SQL query to Cypher or vice versa. All of these posts contain a description of the schemas and SQL/Cypher queries that are intended to be semantically equivalent.
- `Tutorial`. We identified 26 tutorial examples, including from the official Neo4j guide [38], that explain how a SQL query can be implemented using the Cypher query language.
- `Academic`. We identified 7 examples from academic papers [4, 32] that contain relational queries and their corresponding version in Cypher.
- `VeriEQL`. We collected 60 benchmarks from the VeriEQL paper [26] by randomly sampling 20 queries from each of its three datasets and asking various people with at least 3 months of Cypher experience to write an equivalent Cypher query.
- `Mediator`. We collected 100 benchmarks from the Mediator evaluation set [57]. Each Mediator benchmark, consisting of an SQL query pair (Q_1, Q_2) over schemas $\Psi_1$and Ψ_2, was translated into pairs (G_1, Q_2) and (G_2, Q_1) where G_1, G_2 are Cypher queries over graph schemas Ψ'_1 and Ψ'_2, and Ψ_1 and Ψ_2 are the induced relational schemas for Ψ'_1 and Ψ'_2.
- `GPT-Translate`. Given that large language models like GPT are increasingly used by people for coding and transpilation tasks, we included GPT-generated Cypher queries to assess Graphiti's ability to detect errors in automated translations. Specifically, we used GPT to transpile SQL queries from the previous five categories, yielding an additional 205 benchmarks.

Table 2. Results of bounded equivalence checking.

Dataset	#	# Non-Equiv	Avg Checked Bound	Avg Refutation Time (s)
StackOverflow	12	1	9.2	0.6
Tutorial	26	1	7.7	56.2
Academic	7	1	2.5	5.4
VeriEQL	60	4	7.2	8.5
Mediator	100	0	33.2	N/A
GPT-Translate	205	27	18.7	25.9
Total	410	34	19.6	23.4

Database transformers. Since the induced schema of graph databases may differ from the schema of relational databases, Graphiti requires a database transformer to describe the relationship between the graph and relational schemas. To evaluate Graphiti across all pairs of SQL and Cypher queries, one of the authors constructed a database transformer for each query pair based on their schema descriptions. We observe that writing these database transformers is not difficult. As shown in Table 1, each database transformer consists of an average of 5.9 rules, which takes approximately one minute to write.

Machine configuration. All of the experiments are conducted on a laptop with an Intel Core i7-8750H processor and 32GB physical memory running the Debian 12 operating system.

6.1 Evaluation of Graphiti with BMC Backend

In this section, we present the results of the evaluation in which we use the VeriEQL bounded model checker as Graphiti's SQL equivalence checking backend. VeriEQL is a bounded model checker that requires a hyperparameter specifying the size bound of symbolic tables. However, since it is difficult to estimate a suitable bound a priori, we set a 10-minute time limit and gradually increase the bound until either a counterexample is found or the time-limit is reached. For each refuted benchmark, Graphiti uses the relational counterexamples produced by VeriEQL to construct a counterexample over the graph schema.

The results of this evaluation are presented in Table 2. Here, the column labeled "# Non-Equiv" shows the number of benchmarks proven to be *not* equivalent, and the last column shows the average time to find a counterexample. The column labeled "Avg Checked Bound" shows the average size of symbolic tables (measured in terms of the number of rows) when the 10 minute time limit is reached. As shown in Table 2, Graphiti refutes equivalence for 34 out of the 410 benchmarks, taking an average of 23.4 seconds to find a counterexample. For the remaining 376 benchmarks, Graphiti performs bounded verification, demonstrating that there is no counterexample for database instances with symbolic tables of average size 19.6.

Uncovered bugs. We have manually inspected all 34 bugs uncovered by Graphiti and confirmed that all counterexamples produced by the tool correspond to true positives. As expected, GPT-generated queries have a higher probability of being incorrect compared to the human-written queries. In particular, Graphiti finds a counterexample to equivalence for 13% of the queries transpiled by GPT. This experiment shows that GPT may introduce semantic bugs when converting SQL to Cypher, and Graphiti can effectively identify these bugs. As developers increasingly rely on large language models for assistance with coding-related tasks, we believe this demonstrates Graphiti 's practical value for developers relying on LLM-generated queries.

Perhaps more surprisingly, Graphiti also finds incorrect translations among benchmarks in the `StackOverflow, Tutorial, Academic`, and `VeriEQL` categories, all of which involve queries

Table 3. Results of full equivalence verification.

Dataset	#	# Supported	# Verified	# Unknown	Avg Time (s)
StackOverflow	12	1	1	0	1.0
Tutorial	26	1	1	0	0.2
Academic	7	0	0	0	N/A
VeriEQL	60	0	0	0	N/A
Mediator	100	100	77	23	20.5
GPT-Translate	205	94	73	21	23.5
Total	410	196	152	44	16.8

transpiled by experts. Most surprisingly, Graphiti uncovers a bug in an example from a Neo4j tutorial [38] that is intended to help developers familiar with SQL to learn Cypher. This tutorial contains several pairs of SQL and Cypher queries that are intended to be equivalent, but, for one of these examples, Graphiti finds a counterexample on which the query results are actually different. We refer the interested reader to the Appendix of the extended version [25] for a case study explaining some of the uncovered bugs, including the example from the Neo4j tutorial.

False negatives. Since VeriEQL is a bounded model checker, benchmarks that are not refuted by Graphiti within the 10-minute time limit *may* still contain bugs. To assess how frequently this occurs, we sampled 50 pairs of queries that were not refuted by Graphiti and manually inspected whether the translation was correct. For 48 of the 50 manually-inspected benchmarks, we found that the translation is indeed correct, but for 2 benchmarks (both from the GPT-Translate category), the translation is incorrect but Graphiti fails to find a counterexample within the 10 minute time limit. Thus, while Graphiti with the VeriEQL backend does not provide theoretical soundness guarantees, we find that it is useful for finding bugs and has a low chance of missing incorrect translations (4% according to our manual inspection results).

Key finding: Using a bounded model checker backend, Graphiti identifies 27 bugs among 205 SQL queries transpiled to Cypher using GPT. More surprisingly, among the 205 manually-written query pairs that are meant to be equivalent, Graphiti also identifies 7 inconsistencies, including 3 benchmarks from the wild and 4 benchmarks from manual translations.

6.2 Evaluation of Graphiti with Deductive Verifier

In this section, we present the results of a second experiment wherein we evaluate Graphiti with Mediator as its backend. As mentioned earlier, Mediator is an SMT-based deductive verifier for reasoning about SQL applications over different schemas. Unlike VeriEQL, Mediator can perform full-fledged verification; however, it supports a limited subset of SQL without aggregations or outer joins. Additionally, unlike VeriEQL, Mediator cannot disprove equivalence by generating counterexamples. Hence, when using Graphiti with Mediator as its backend, Graphiti can either prove equivalence or it returns "Unknown". For performing this experiment, we also use a time limit of 10 minutes per benchmark.

The results of this experiment are summarized in Table 3. As shown in the "# Supported" column, about half of the benchmarks (196 out of 410) fall inside the fragment of SQL supported by Mediator, so we conduct our evaluation on this subset. Overall, Graphiti can verify 77.6% of these 196 benchmarks, with an average running time of 16.8 seconds. Since the syntax-directed transpilation performed by Graphiti takes negligible time, most of the verification time is dominated by SMT queries for discharging the generated verification conditions.

Qualitative analysis. We manually inspected 44 benchmarks that cannot be verified by Graphiti with the Mediator backend. Among these 44 benchmarks, two of them are in fact *refuted* by

Table 4. Execution time of transpiled and manually-written SQL queries.

Dataset	#	Avg Exec Time (s)		% Transpiled Faster	% Trans Slower (1x, 1.1x]	% Trans Slower (1.1x, 1.2x]	% Trans Slower (1.2x, +∞)
		Transpiled	Manual				
`StackOverflow`	12	1.9	1.8	41.7%	8.3%	50.0%	0.0%
`Tutorial`	26	2.3	1.7	19.2%	11.5%	46.2%	23.1%
`Academic`	7	2.4	3.2	71.4%	0.0%	14.3%	14.3%
Total	45	2.2	2.0	33.3%	8.9%	42.2%	15.6%

Graphiti with VeriEQL, so they should *not* be verified. Among the 42 remaining benchmarks, Mediator fails to complete verification within the 10 minute time limit for 14 of these, and, for the final 28 benchmarks, it terminates but returns "Unknown". To gain insight about failure cases, note that Mediator needs to infer an *inductive bisimulation invariant* between the two queries [57]. However, for queries involving long join chains, the corresponding bisimulation invariant can be complex. This can either lead to expensive SMT queries, thereby causing time-outs, or the required invariant might fall outside the inference capabilities of Mediator.

Key finding: Among the 196 SQL queries supported by Mediator, Graphiti can prove equivalence between roughly 80% of (Cypher, SQL) query pairs using the Mediator backend.

6.3 Evaluation of Transpilation

While the primary goal of this work is to enable checking equivalence between graph and relational queries, an additional benefit of our method is that it can transpile graph database queries to relational queries over the induced schema. To assess the practical effectiveness, we conduct an experiment to evaluate how well our method transpiles graph queries into *efficient* SQL queries.

Efficiency of transpilation. First, we evaluate how long Graphiti takes to transpile each Cypher query to a SQL query over the induced schema. Graphiti can successfully transpile all 410 queries, and the average, median, and maximum transpilation times are 6.3, 3.0, and 180.2 milliseconds respectively. Hence, we can conclude that transpilation is very fast in practice.

Quality of transpiled queries. Next, we also set out to evaluate the quality of the transpiled queries by comparing the execution time of manually written SQL queries against Graphiti's transpilation result. However, performing this evaluation is challenging for two reasons: First, we only have access to the "ground truth" Cypher and SQL queries for some benchmark categories. Second, we do not have database instances that these queries are meant to be executed on. To deal with the first challenge, we conduct this evaluation only on those benchmarks from the `StackOverflow`, `Tutorial`, and `Academic` categories for which we are given the original Cypher query and its SQL equivalent (or vice versa). To deal with the second challenge, we generate mock database instances and assess query efficiency on them. For each benchmark with SQL query Q_R over schema Ψ_R and Cypher query Q_G, we use Graphiti to transpile Q_G into SQL query Q'_R over the induced schema Ψ'_R. We then create databases R and R' over Ψ_R and Ψ'_R, respectively, ensuring $\Phi_{\text{rdt}}(R') = R$. To account for execution time variability, we start with 10,000 tuples in each table of R and iteratively increase the table size by 10x, up to 1 million, choosing the largest size where manually written SQL queries run within 10 seconds. This approach results in 1 million tuples for 36 benchmarks and between 10,000 and 1 million for the remaining 9 benchmarks. Finally, we measure the execution times of Q'_R on R' and Q_R on R. Table 4 summarizes the results: for 33.3% of the benchmarks, the transpiled queries are faster than the manually written queries. For the remaining benchmarks, 8.9% exhibit a slowdown of no more than 1.1x, 42.2% exhibit a slowdown between 1.1x and 1.2x, and 15.6% exceed 1.2x. These results indicate that using Graphiti to perform automated transpilation

could be useful in real-world scenarios that necessitate graph-to-relational query conversion, such as for legacy systems, resource-constrained environments, or data integration.

Key finding: Graphiti can transpile Cypher queries to SQL queries in milliseconds. The execution time of transpiled SQL queries is faster than manually-written queries on 33.3% of benchmarks and within 1.2x slowdown on 51.1% of benchmarks.

7 Related Work

In this section, we discuss prior work that is mostly related to our techniques for equivalence verification between Cypher and SQL queries.

Automated reasoning for SQL. Despite the undecidability of checking equivalence between SQL queries [51], there has been much prior work on automated reasoning for relational queries. We can categorize existing work into three classes. The first class targets a decidable subset of SQL and proposes decision procedures for that subset. Examples of work in this category include [2, 5, 21]. Approaches in the second category propose sound but incomplete algorithms for an undecidable subset of SQL; examples of work in this space include [10, 12, 63, 64]. The third category performs bounded verification to find bugs in SQL queries; examples include Cosette [11], Qex [53], and VeriEQL [26]. There is also prior work on verifying relational database applications that involve both queries and updates [57]. Our proposed approach reduces the verification problem between graph and relational queries to checking equivalence between a pair of relational queries; as such, it can leverage any future advances in this area.

Migration between database instances. There is prior work on migrating data between different schemas, including [18, 29, 34, 55, 59–61]. The most related to this paper is Dynamite [59], which automates data migration between graph and relational databases. However, Dynamite is only useful for migrating the *contents* of the database and cannot be used for transpiling queries. There are also prior papers that address the query transpilation problem in the context of SQL [13, 14, 58]. While our notion of database transformer is inspired by prior work [37, 59], to the best of our knowledge, this paper is the first to formalize the transpilation procedure from Cypher to SQL queries and leverage it for formal equivalence checking.

Data representation refactoring. There is a related line of work on *data representation refactoring*, which aims to refactor programs or specifications from one data representation to another [8, 9, 15, 20, 30, 43–45, 56]. For instance, Solidare [43] refactors smart contracts between different ADTs. QBS [8] converts Java programs operating over collections to SQL queries. Since graph and relational schemas can be viewed as different data representations, the query transpilation problem in this work can be viewed as a form of data representation refactoring. However, none of the existing techniques addresses the query transpilation problem between graph and relational data. Additionally, this work can be viewed as presenting a novel methodology for verifying data representation refactoring: Rather than directly going from representation R to representation R', our idea is to introduce an auxiliary representation R'' that simplifies the problem by enabling syntax-directed translation. To the best of our knowledge, such a methodology based on a layer of indirection has not previously been explored in this context.

Graph database query languages. There has been a long line of prior work on graph databases and semantic foundations of graph query languages [1, 16, 19, 22, 23, 48, 52]. The unifying insight behind many of such works is that a graph database schema may be viewed as a graphical representation of the Entity-Relationship Diagram (ER Diagram) of a relational database schema. In part due to this unifying insight, these graph query languages are both semantically and syntactically similar

to Cypher. Because of this similarity, we believe our proposed methodology can be adapted fairly easily to graph database query languages other than Cypher.

Testing database queries. Another related line of related work focuses on testing database queries. Work in this space includes differential testing [27, 35, 62] and metamorphic testing [7, 28, 33, 46] to detect bugs in database management systems, mutation-based testing to grade programming assignments involving SQL queries [6], and provenance-based techniques for explaining wrong SQL queries [36]. In contrast, Graphiti transpiles graph database queries into equivalent SQL queries and uses existing automated reasoning tools for SQL equivalence checking. Thus, Graphiti is complementary and can benefit from advances in testing SQL queries.

Transpiling Cypher queries. There are a few existing tools that can translate Cypher queries to queries in SQL-like languages [31, 50]. The most relevant to this paper is OpenCypherTranspiler [31], which first transforms a Cypher query into a logical plan and renders it as a relational query. However, it supports only a limited subset of Cypher and lacks soundness guarantees for translated queries.[5] In contrast, Graphiti ensures soundness during transpilation and supports a broader subset of Cypher queries. Another tool, Kuzu [50], can execute graph queries on relational databases with a Cypher interface. It compiles Cypher queries into an intermediate representation similar to a relational database's logical plan. However, Kuzu does not transpile Cypher into SQL but instead directly executes the intermediate representation on the database.

8 Limitation

The current version of Graphiti is focused on a specific subset of SQL and Cypher, which does not yet cover all modern features, such as variable-length pattern matching in Cypher. However, considering the lack of prior research on reasoning about equivalence between graph and relational database queries, we believe our selected fragments offer a strong foundation for advancing this area of study. While some SQL and Cypher queries lie outside the scope of our current subset, evaluations on a diverse set of benchmarks, including real-world queries, demonstrate that this subset is expressive enough for practical use cases. Future work can further extend the transpilation rules and backend equivalence verifiers to support additional features.

9 Conclusion and Future Work

In this paper, we proposed automated reasoning techniques between graph and relational database queries. Specifically, we first proposed a formal definition of equivalence between graph and relational queries and used it as the basis of a correct-by-construction transpilation strategy for converting Cypher queries to SQL queries over a so-called *induced relational schema*. We then showed how our translation approach can be used to check equivalence between graph and SQL queries over *arbitrary* schema by leveraging existing automated reasoning techniques for SQL. We have also evaluated our implementation, Graphiti, on equivalence checking tasks involving real-world Cypher and SQL queries and showed that Graphiti can be useful for (a) uncovering subtle bugs in Cypher queries that are meant to be equivalent to a reference SQL implementation, and (b) verifying full equivalence between Cypher and SQL queries.

Looking ahead, we plan to explore the development of a graphical interface for specifying database transformers between graph and relational databases, inspired by prior work on schema mapping visualization [47]. This interface would aim to further reduce the manual effort required by users to verify equivalence between graph and relational queries, providing a more intuitive and user-friendly approach. We see this as a promising avenue for future research.

[5]The detailed evaluation of OpenCypherTranspiler can be found in the Appendix of the extended version [25].

Acknowledgments

We would like to thank the anonymous reviewers for their insightful comments and feedback on this paper. This work was conducted in research groups supported by Natural Sciences and Engineering Research Council of Canada (NSERC) Discovery Grant and National Science Foundation (NSF) awards CCF-1762299, CCF-1918889, CNS-1908304, CCF-1901376, CNS-2120696, CCF-2210831, and CCF-2319471.

Data-Availability Statement

The software that implements the techniques described in Section 5 and supports the evaluation results reported in Section 6 is available on Zenodo [24].

References

[1] AGE. 2024. Apache AGE. https://age.apache.org.

[2] Alfred V. Aho, Yehoshua Sagiv, and Jeffrey D. Ullman. 1979. Equivalences Among Relational Expressions. *SIAM J. Comput.* 8, 2 (1979), 218–246. doi:10.1137/0208017

[3] Amazon. 2024. What's the Difference Between a Graph Database and a Relational Database? https://aws.amazon.com/compare/the-difference-between-graph-and-relational-database/?nc1=h_ls.

[4] Abdelkrim Boudaoud, Houari Mahfoud, and Azeddine Chikh. 2022. Towards a Complete Direct Mapping from Relational Databases to Property Graphs. In *Proceedings of the International Conference on Model and Data Engineering (MEDI)*. 222–235. doi:10.1007/978-3-031-21595-7_16

[5] Ashok K. Chandra and Philip M. Merlin. 1977. Optimal Implementation of Conjunctive Queries in Relational Data Bases. In *Proceedings of the ACM Symposium on Theory of Computing (STOC)*. 77–90. doi:10.1145/800105.803397

[6] Bikash Chandra, Bhupesh Chawda, Biplab Kar, K. V. Maheshwara Reddy, Shetal Shah, and S. Sudarshan. 2015. Data generation for testing and grading SQL queries. *VLDB J.* 24, 6 (2015), 731–755. doi:10.1007/S00778-015-0395-0

[7] Tsong Yueh Chen, S. C. Cheung, and Siu-Ming Yiu. 2020. Metamorphic testing: a new approach for generating next test cases. *CoRR* abs/2002.12543 (2020). arXiv:2002.12543 https://arxiv.org/abs/2002.12543

[8] Yanju Chen, Yuepeng Wang, Maruth Goyal, James Dong, Yu Feng, and Isil Dillig. 2022. Synthesis-powered optimization of smart contracts via data type refactoring. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 560–588. doi:10.1145/3563308

[9] Alvin Cheung, Armando Solar-Lezama, and Samuel Madden. 2013. Optimizing database-backed applications with query synthesis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. ACM, 3–14. doi:10.1145/2491956.2462180

[10] Shumo Chu, Brendan Murphy, Jared Roesch, Alvin Cheung, and Dan Suciu. 2018. Axiomatic Foundations and Algorithms for Deciding Semantic Equivalences of SQL Queries. *Proc. VLDB Endow.* 11, 11 (2018), 1482–1495. doi:10.14778/3236187.3236200

[11] Shumo Chu, Chenglong Wang, Konstantin Weitz, and Alvin Cheung. 2017. Cosette: An Automated Prover for SQL. In *Proceedings of the Biennial Conference on Innovative Data Systems Research (CIDR)*. http://cidrdb.org/cidr2017/papers/p51-chu-cidr17.pdf

[12] Shumo Chu, Konstantin Weitz, Alvin Cheung, and Dan Suciu. 2017. HoTTSQL: proving query rewrites with univalent SQL semantics. In *Proceedings of the SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 510–524. doi:10.1145/3062341.3062348

[13] Carlo Curino, Hyun Jin Moon, Alin Deutsch, and Carlo Zaniolo. 2013. Automating the database schema evolution process. *VLDB J.* 22, 1 (2013), 73–98. doi:10.1007/s00778-012-0302-x

[14] Carlo Curino, Hyun Jin Moon, and Carlo Zaniolo. 2008. Graceful database schema evolution: the PRISM workbench. *Proc. VLDB Endow.* 1, 1 (2008), 761–772. doi:10.14778/1453856.1453939

[15] Benjamin Delaware, Clément Pit-Claudel, Jason Gross, and Adam Chlipala. 2015. Fiat: Deductive Synthesis of Abstract Data Types in a Proof Assistant. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL)*. 689–700. doi:10.1145/2676726.2677006

[16] Alin Deutsch, Nadime Francis, Alastair Green, Keith Hare, Bei Li, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Wim Martens, Jan Michels, Filip Murlak, Stefan Plantikow, Petra Selmer, Oskar van Rest, Hannes Voigt, Domagoj Vrgoc, Mingxi Wu, and Fred Zemke. 2022. Graph Pattern Matching in GQL and SQL/PGQ. In *International Conference on Management of Data (SIGMOD)*. 2246–2258. doi:10.1145/3514221.3526057

[17] Ronald Fagin, Laura M. Haas, Mauricio A. Hernández, Renée J. Miller, Lucian Popa, and Yannis Velegrakis. 2009. Clio: Schema Mapping Creation and Data Exchange. In *Conceptual Modeling: Foundations and Applications - Essays in Honor of John Mylopoulos (Lecture Notes in Computer Science, Vol. 5600)*. 198–236. doi:10.1007/978-3-642-02463-4_12

[18] Yu Feng, Ruben Martins, Jacob Van Geffen, Isil Dillig, and Swarat Chaudhuri. 2017. Component-based synthesis of table consolidation and transformation tasks from examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 422–436. doi:10.1145/3062341.3062351

[19] Nadime Francis, Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Mats Rydberg, Petra Selmer, and Andrés Taylor. 2018. Cypher: An Evolving Query Language for Property Graphs. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 1433–1445. doi:10.1145/3183713.3190657

[20] Xi Ge, Quinton L. DuBose, and Emerson R. Murphy-Hill. 2012. Reconciling manual and automatic refactoring. In *International Conference on Software Engineering (ICSE)*. 211–221. doi:10.1109/ICSE.2012.6227192

[21] Todd J. Green. 2011. Containment of Conjunctive Queries on Annotated Relations. *Theory Comput. Syst.* 49, 2 (2011), 429–459. doi:10.1007/S00224-011-9327-6

[22] Harry Halpin and James Cheney. 2011. Dynamic Provenance for SPARQL Updates Using Named Graphs. In *Workshop on the Theory and Practice of Provenance (TaPP)*. doi:10.1145/2567948.2577357

[23] Olaf Hartig and Jorge Pérez. 2018. Semantics and Complexity of GraphQL. In *Proceedings of the World Wide Web Conference on World Wide Web (WWW)*. 1155–1164. doi:10.1145/3178876.3186014

[24] Yang He, Ruijie Fang, Işıl Dillig, and Yuepeng Wang. 2025. Artifact Evaluation Graphiti: Bridging Graph and Relational Database Queries. doi:10.5281/zenodo.15043281

[25] Yang He, Ruijie Fang, Isil Dillig, and Yuepeng Wang. 2025. Graphiti: Bridging Graph and Relational Database Queries. arXiv:2504.03182

[26] Yang He, Pinhan Zhao, Xinyu Wang, and Yuepeng Wang. 2024. VeriEQL: Bounded Equivalence Verification for Complex SQL Queries with Integrity Constraints. *Proc. ACM Program. Lang.* 8, OOPSLA1 (2024), 1071–1099. doi:10.1145/3649849

[27] Ziyue Hua, Wei Lin, Luyao Ren, Zongyang Li, Lu Zhang, Wenpin Jiao, and Tao Xie. 2023. GDsmith: Detecting Bugs in Cypher Graph Database Engines. In *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 163–174. doi:10.1145/3597926.3598046

[28] Yuancheng Jiang, Jiahao Liu, Jinsheng Ba, Roland HC Yap, Zhenkai Liang, and Manuel Rigger. 2024. Detecting Logic Bugs in Graph Database Management Systems via Injective and Surjective Graph Query Transformation. In *Proceedings of the International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3623307

[29] Zhongjun Jin, Michael R. Anderson, Michael J. Cafarella, and H. V. Jagadish. 2017. Foofah: Transforming Data By Example. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*. 683–698. doi:10.1145/3035918.3064034

[30] Shadaj Laddad, Conor Power, Mae Milano, Alvin Cheung, and Joseph M. Hellerstein. 2022. Katara: synthesizing CRDTs with verified lifting. *Proc. ACM Program. Lang.* 6, OOPSLA2 (2022), 1349–1377. doi:10.1145/3563336

[31] Jerry Liang. 2025. openCypher Transpiler. https://github.com/microsoft/openCypherTranspiler.

[32] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast In-Memory SQL Analytics on Relationships between Entities. *CoRR* abs/1602.00033 (2016). arXiv:1602.00033 http://arxiv.org/abs/1602.00033

[33] Qiuyang Mang, Aoyang Fang, Boxi Yu, Hanfei Chen, and Pinjia He. 2024. Testing Graph Database Systems via Equivalent Query Rewriting. In *Proceedings of the International Conference on Software Engineering (ICSE)*. doi:10.1145/3597503.3639200

[34] Ruben Martins, Jia Chen, Yanju Chen, Yu Feng, and Isil Dillig. 2019. Trinity: An Extensible Synthesis Framework for Data Science. *Proc. VLDB Endow.* 12, 12 (2019), 1914–1917. doi:10.14778/3352063.3352098

[35] William M. McKeeman. 1998. Differential Testing for Software. *Digit. Tech. J.* 10, 1 (1998), 100–107.

[36] Zhengjie Miao, Sudeepa Roy, and Jun Yang. 2019. Explaining Wrong Queries Using Small Examples. In *Proceedings of the International Conference on Management of Data (SIGMOD)*. 503–520. doi:10.1145/3299869.3319866

[37] Renée J. Miller, Laura M. Haas, and Mauricio A. Hernández. 2000. Schema Mapping as Query Discovery. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*. 77–88. http://www.vldb.org/conf/2000/P077.pdf

[38] Neo4j. 2024. *Cypher for SQL Users.* https://neo4j.com/docs/getting-started/cypher-intro/cypher-sql/ Accessed: 2024-11-07.

[39] Neo4j. 2024. Neo4j Docs. https://neo4j.com/docs/getting-started/cypher-intro/cypher-sql/#_return_customers_without_existing_orders.

[40] Neo4j. 2024. Transition from Relational to Graph Database. https://neo4j.com/docs/getting-started/appendix/graphdb-concepts/graphdb-vs-rdbms.

[41] NLM. 2024. SemMedDB Database Details - Version 2.0. https://lhncbc.nlm.nih.gov/ii/tools/SemRep_SemMedDB_SKR/dbinfo20.html.

[42] Stack Overflow. 2024. How could i use this SQL on cypher(neo4j). https://stackoverflow.com/questions/43438214/how-could-i-use-this-sql-on-cypherneo4j.

[43] Shankara Pailoor, Yuepeng Wang, and Isil Dillig. 2024. Semantic Code Refactoring for Abstract Data Types. *Proc. ACM Program. Lang.* 8, POPL (2024), 816–847. doi:10.1145/3632870

[44] Shankara Pailoor, Yuepeng Wang, Xinyu Wang, and Isil Dillig. 2021. Synthesizing data structure refinements from integrity constraints. In *Proceedings of the International Conference on Programming Language Design and Implementation (PLDI)*. 574–587. doi:10.1145/3453483.3454063

[45] Kia Rahmani, Kartik Nagar, Benjamin Delaware, and Suresh Jagannathan. 2021. Repairing serializability bugs in distributed database programs via automated schema refactoring. In *ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*. 32–47. doi:10.1145/3453483.3454028

[46] Manuel Rigger and Zhendong Su. 2020. Finding bugs in database systems via query partitioning. *Proc. ACM Program. Lang.* 4, OOPSLA (2020), 211:1–211:30. doi:10.1145/3428279

[47] George G. Robertson, Mary Czerwinski, and John E. Churchill. 2005. Visualization of mappings between schemas. In *Proceedings of the Conference on Human Factors in Computing Systems (CHI)*. 431–439. doi:10.1145/1054972.1055032

[48] Marko A. Rodriguez. 2015. The Gremlin graph traversal machine and language (invited talk). In *Proceedings of the Symposium on Database Programming Languages (DBPL)*. 1–10. doi:10.1145/2815072.2815073

[49] StackOverflow. 2024. Comparison of relational databases and graph databases. https://stackoverflow.com/questions/13046442/comparison-of-relational-databases-and-graph-databases.

[50] Kuzu Team. 2025. Kuzu. https://kuzudb.com/.

[51] Boris A Trakhtenbrot. 1950. Impossibility of an algorithm for the decision problem in finite classes. In *Doklady Akademii Nauk SSSR 70*. 569–572.

[52] Oskar van Rest, Sungpack Hong, Jinha Kim, Xuming Meng, and Hassan Chafi. 2016. PGQL: a property graph query language. In *Proceedings of the International Workshop on Graph Data Management Experiences and Systems (GRADES)*. 7. doi:10.1145/2960414.2960421

[53] Margus Veanes, Nikolai Tillmann, and Jonathan de Halleux. 2010. Qex: Symbolic SQL Query Explorer. In *International Conference on Logic for Programming, Artificial Intelligence, and Reasoning (LPAR)*. 425–446. doi:10.1007/978-3-642-17511-4_24

[54] Chad Vicknair, Michael Macias, Zhendong Zhao, Xiaofei Nan, Yixin Chen, and Dawn Wilkins. 2010. A comparison of a graph database and a relational database: a data provenance perspective. In *Proceedings of the Annual Southeast Regional Conference*. 42. doi:10.1145/1900008.1900067

[55] Chenglong Wang, Alvin Cheung, and Rastislav Bodík. 2017. Synthesizing highly expressive SQL queries from input-output examples. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 452–466. doi:10.1145/3062341.3062365

[56] Meng Wang, Jeremy Gibbons, Kazutaka Matsuda, and Zhenjiang Hu. 2013. Refactoring pattern matching. *Sci. Comput. Program.* 78, 11 (2013), 2216–2242. doi:10.1016/J.SCICO.2012.07.014

[57] Yuepeng Wang, Isil Dillig, Shuvendu K. Lahiri, and William R. Cook. 2018. Verifying equivalence of database-driven applications. *Proc. ACM Program. Lang.* 2, POPL (2018), 56:1–56:29. doi:10.1145/3158144

[58] Yuepeng Wang, James Dong, Rushi Shah, and Isil Dillig. 2019. Synthesizing database programs for schema refactoring. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 286–300. doi:10.1145/3314221.3314588

[59] Yuepeng Wang, Rushi Shah, Abby Criswell, Rong Pan, and Isil Dillig. 2020. Data Migration using Datalog Program Synthesis. *Proc. VLDB Endow.* 13, 7 (2020), 1006–1019. doi:10.14778/3384345.3384350

[60] Navid Yaghmazadeh, Christian Klinger, Isil Dillig, and Swarat Chaudhuri. 2016. Synthesizing transformations on hierarchically structured data. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*. 508–521. doi:10.1145/2908080.2908088

[61] Navid Yaghmazadeh, Xinyu Wang, and Isil Dillig. 2018. Automated Migration of Hierarchical Data to Relational Tables using Programming-by-Example. *Proc. VLDB Endow.* 11, 5 (2018), 580–593. doi:10.1145/3187009.3177735

[62] Yingying Zheng, Wensheng Dou, Yicheng Wang, Zheng Qin, Lei Tang, Yu Gao, Dong Wang, Wei Wang, and Jun Wei. 2022. Finding bugs in Gremlin-based graph database systems via Randomized differential testing. In *Proceedings of the SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*. 302–313. doi:10.1145/3533767.3534409

[63] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Jinpeng Wu. 2022. SPES: A Symbolic Approach to Proving Query Equivalence Under Bag Semantics. In *International Conference on Data Engineering (ICDE)*. 2735–2748. doi:10.1109/ICDE53745.2022.00250

[64] Qi Zhou, Joy Arulraj, Shamkant B. Navathe, William Harris, and Dong Xu. 2019. Automated Verification of Query Equivalence Using Satisfiability Modulo Theories. *Proc. VLDB Endow.* 12, 11 (2019), 1276–1288. doi:10.14778/3342263.3342267

Received 2024-11-14; accepted 2025-03-06