Proving Cypher Query Equivalence

Lei Tang*†, Wensheng Dou*†‡§, Yingying Zheng*†, Lijie Xu*†, Wei Wang*†‡§, Jun Wei*†‡§, Tao Huang*†
*Key Lab of System Software, State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences
†University of Chinese Academy of Sciences, Beijing
‡University of Chinese Academy of Sciences, Nanjing, §Nanjing Institute of Software Technology
{tanglei20, wsdou, zhengyingying14, xulijie09, wangwei, wj, tao}@otcaix.iscas.ac.cn

***Abstract*—Graph database systems store graph data as nodes and relationships, and utilize graph query languages (e.g., Cypher) for efficiently querying graph data. Proving the equivalence of graph queries is an important foundation for optimizing graph query performance, ensuring graph query reliability, etc. Although researchers have proposed many SQL query equivalence provers for relational database systems, these provers cannot be directly applied to prove the equivalence of graph queries. The difficulty lies in the fact that graph query languages (e.g., Cypher) adopt significantly different data models (property graph model vs. relational model) and query patterns (graph pattern matching vs. tabular tuple calculus) from SQL.**

In this paper, we propose `GraphQE`, an automated prover to determine whether two Cypher queries are semantically equivalent. We design a U-semiring based Cypher algebraic representation to model the semantics of Cypher queries. Our Cypher algebraic representation is built on the algebraic structure of unbounded semirings, and can sufficiently express nodes and relationships in property graphs and complex Cypher queries. Then, determining the equivalence of two Cypher queries is transformed into determining the equivalence of the corresponding Cypher algebraic representations, which can be verified by SMT solvers. To evaluate the effectiveness of `GraphQE`, we construct a dataset consisting of 148 pairs of equivalent Cypher queries. Among them, we have successfully proven 138 pairs of equivalent Cypher queries, demonstrating the effectiveness of `GraphQE`.

I. Introduction

Graph database systems (GDBs) are designed to efficiently store and retrieve graph data. Graph storage technologies [20], [39] have developed rapidly, and many GDBs have emerged, e.g., Neo4j [35], Microsoft Azure Cosmos [1], ArangoDB [5] and Memgraph [9]. GDBs have been widely used in many applications, e.g., knowledge graphs [47], fraud detection [42], molecular and cell biology [23], and social networks [25]. Recent reports [6], [31] also show that GDBs have gained almost 1,000% popularity growth since 2013 and reached a market of around 1.7 billion dollars by 2023.

Most GDBs (e.g., Neo4j [35], ArangoDB [5] and OrientDB [43]) are built on the property graph model [11], in which graph data are stored as nodes and relationships along with their properties. Fig. 1 shows an illustrative property graph that consists of four nodes and three relationships. Specifically, each relationship is directed and describes a path from one node to another (or itself), e.g., relationship r_1 connects node n_1 and n_2. The property graph model leverages labels to shape the domain of nodes and relationships. Nodes (or relationships) that have the same label are categorized into the same set, e.g., node n_1, n_3 and n_4 are categorized by label $Person$.

GDBs adopt graph query languages (e.g., Cypher [27], Gremlin [7] and GQL [21], [32]) to define graph patterns, and utilize isomorphic [46] or homomorphic [12] graph pattern matching to effectively retrieve graph data. Among these graph query languages, Cypher is one of the most widely-used languages, and supported by 4 out of the top 10 GDBs in DB-Engines Ranking [6]. Listing 1 shows an example of a Cypher query that defines a graph pattern to retrieve the author of the book read by Alice in the property graph in Fig. 1.

Query equivalence proving is a fundamental problem in database research [14], [18], [44], and has been widely applied in detecting query optimization bugs [28], mining query rewriting rules [49], eliminating query computational overlaps [53], etc. We have witnessed the significant progress of SQL query equivalence provers [16], [17], [22], [29], [48], [49], [52], [53] for relational database systems since 2018. Initially, syntax-based approaches [16], [17], [29], [52], [53] are proposed to prove SQL query equivalence by checking the syntactic isomorphism of SQL algebraic representations (e.g., K-relations [29]). Although these approaches can prove SQL query equivalence to a certain extent, they struggle with handling SQL queries that have significantly different syntactic structures. Recently, researchers [22], [48], [49] propose semantic-based approaches to solve this problem. These approaches use U-semiring based SQL algebraic representations [16] to model the semantics of SQL queries, and construct the U-semiring expression for a SQL query (*U-expression* for short). *U-expression* is a natural number semiring expression and models a SQL query Q as an expression $u(t)$ that returns the natural number multiplicity of a tuple t in Q's query result. Semantic-based approaches solve the equivalence of SQL queries by proving the equivalence of *U-expression*s on arbitrary input tuple t based on SMT solvers [19]. The state-of-the-art semantic-based prover SQLSolver [22] has demonstrated that its effectiveness significantly surpasses that of syntax-based approaches [16], [52], [53].

Similar to SQL equivalence provers for relational database systems, graph query equivalence provers also have significant value for GDBs, and can be applied in detecting graph query optimization bugs [33], mining graph query optimization rules [30], optimizing graph queries [14], etc. However, we still

Wensheng Dou is the corresponding author.

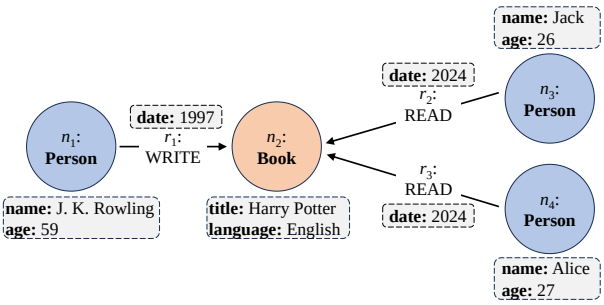

Fig. 1. An illustrative property graph. We assign a variable (e.g., n_1 and r_1) for each node and relationship for easy reference.

```
1 MATCH (reader:Person)-[:READ]->(book:Book)<-[:
      WRITE]-(writer)
2 WHERE reader.name = 'Alice'
3 RETURN writer.name
```

Listing 1. A Cypher query retrieves the author of the book read by Alice.

lack a graph query equivalence prover. Graph query languages (e.g., Cypher) employ significantly different syntaxes from SQL, and are built on the property graph model and graph pattern matching. In contrast, *U-expressions* [16] are built on the relational data model and tabular tuple calculus utilized by SQL. Therefore, we cannot directly apply *U-expressions* to model Cypher queries.

To model Cypher queries using *U-expressions* [16], we first need to transform property graphs into relational tables. For example, Cytosm [45] groups nodes and relationships in a property graph by their labels and generates an individual table for each label. Then, we can potentially use *U-expressions* on these generated tables to model some Cypher features, e.g., predicates and unions. However, some Cypher features cannot be modeled in this way due to the limitation of *U-expressions*. First, based on the tables generated by Cytosm, *U-expressions* cannot model node and relationship patterns either without specifying labels or with multiple labels, e.g., node pattern `(writer)` in Listing 1 that is not associated with any label. Second, *U-expressions* cannot model some specific Cypher features, e.g., arbitrary-length paths (e.g., `()-[*]->()`) and list variables with `UNWIND` and `COLLECT`. Third, Chu et al. [16] cannot support some common features supported by SQL and Cypher e.g., `ORDER BY`, `LIMIT` and `SKIP`.

In this paper, we propose a U-semiring based *graph-native* algebraic representation to accurately model property graphs and Cypher queries, and construct U-semiring expressions for Cypher queries (*G-expression* for short). *G-expressions* can model nodes and relationships in property graphs and complex features in Cypher queries. We then transform proving the equivalence of Cypher queries into proving the equivalence of *G-expressions* on an unspecified property graph. We further leverage the LIA* theory [22] to prove the equivalence of *G-expressions* by SMT solvers [19].

To model property graphs, we define three algebraic functions $Node(e)$, $Rel(e)$, and $Lab(e, label)$ to precisely model a graph entity e's type (i.e., node or relationship) and labels (whether e has a label $label$) in property graphs, respectively. Thus, *G-expressions* can model node and relationship patterns without specifying labels or with multiple labels. For example, *G-expression* models the node pattern `(reader:Person)` as $Node(reader) \times Lab(reader, Person)$ and the node pattern `(writer)` without specifying any label as $Node(writer)$. We further define algebraic functions to model the outgoing and incoming nodes of a relationship. Based on these algebraic functions, we construct a U-semiring based algebraic representation for Cypher's core features, including graph patterns, predicates and query results, e.g., `MATCH`, `WHERE` and `RETURN` clauses. We can further model advanced Cypher features, e.g., using uninterpreted functions in SMT to model arbitrary-length paths, and designing a divide-and-conquer proving process to handle `ORDER BY`, `LIMIT` and `SKIP`.

We implement our approach on Cypher 9 in the openCypher project [2] as `GraphQE`. To evaluate `GraphQE`, we construct a dataset `CyEqSet` with 148 pairs of equivalent Cypher queries through the following two methods. (1) We translate the equivalent SQL query pairs in Calcite [4], a widely-used open-source dataset of equivalent SQL queries [16], [22], [49], [53], to their corresponding Cypher queries, and successfully construct 79 pairs of equivalent Cypher queries. (2) We collect 36 real-world Cypher queries from popular open-source graph database benchmarks [24] and widely-used open-source Cypher projects [8], [13]. Then, we construct equivalent Cypher queries by applying three existing Cypher query rewriting rules [30], [33], and obtain 68 pairs of equivalent Cypher queries. `CyEqSet` contains both simple and complex graph patterns, e.g., optional graph patterns (`OPTIONAL`) and variable-length paths (`-[*2..3]->`). Finally, we evaluate `GraphQE` on `CyEqSet`, and `GraphQE` has successfully proven 138 pairs of them with a latency of 38 ms on average. `GraphQE` and `CyEqSet` are available at https://github.com/choeoe/GraphQe.

In summary, we make the following contributions.

- We propose a U-semiring based *graph-native* algebraic representation for property graphs and Cypher queries.
- We propose `GraphQE`, the first graph query equivalence prover for Cypher queries.
- We construct `CyEqSet`, the first dataset with 148 pairs of equivalent Cypher queries for Cypher query equivalence proving.
- We evaluate `GraphQE` on `CyEqSet`. `GraphQE` successfully proves 138 out of 148 pairs of equivalent Cypher queries.

II. Preliminaries

In this section, we first introduce the property graph model (Section II-A). Next, we explain Cypher and its graph pattern matching mechanism (Section II-B). Finally, we introduce the U-semiring algebraic structure and how to model SQL semantics on U-semiring (Section II-C).

A. Property Graph Model

Most GDBs (e.g., Neo4j [35], JanusGraph [3], OrientDB [43] and ArangoDB [5]) adopt the property graph model [11],

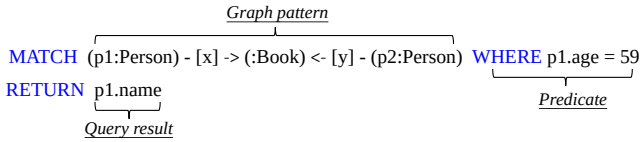

Fig. 2. The basic structure of a Cypher query.

[36] to store graph data. In the property graph model, each node or relationship contains a set of properties. Each relationship connects a node to another (or itself). The property graph model utilizes labels to categorize nodes (or relationships) into different sets, in which all nodes (or relationships) with a certain label belong to the same set.

Different graph query languages may adopt different property graph models. For example, although GQL is fully compatible with Cypher, they adopt slightly different property graph models. Cypher only supports directed relationships and requires each relationship to have only one label, while GQL further supports undirected relationships and allows a relationship to have more than one label. Cypher adopts relationship-injective semantics when mapping relationship patterns to relationships in property graphs, while GQL does not. In this paper, we mainly focus on Cypher[1], and we present the formal definition of a property graph model adopted by Cypher as follows.

Definition 1. *A property graph used by Cypher is denoted as a tuple $G = \langle N, R, \rho, \lambda, \sigma \rangle$, where*

1) *N is a finite set of nodes used in G.*
2) *R is a finite set of directed relationships used in G.*
3) *$\rho : R \rightarrow N \times N$ is a function that maps a directed relationship from R to its outgoing and incoming nodes from N.*
4) *$\lambda : N \cup R \rightarrow 2^{\mathcal{L}}$ is a labeling function that associates each node or relationship to a finite set of labels from $\mathcal{L}$ (the full set of labels). In Cypher, a node can have one or multiple labels, and a relationship can have only one label.*
5) *$\sigma : (N \cup R) \times \mathcal{K} \rightarrow Const$ is a partial function that associates a constant with a node (or relationship) and a property key from $\mathcal{K}$ (the full set of property keys).*

Take the property graph in Fig. 1 as an example. In this property graph, four nodes form a finite node set $N = \{n_1, n_2, n_3, n_4\}$, and three relationships form a finite relationship set $R = \{r_1, r_2, r_3\}$. For relationship r_1, which connects the outgoing node n_1 and incoming node n_2 (i.e., $\rho(r_1) = \langle n_1, n_2 \rangle$), has a label $WRITE$ (i.e., $\lambda(r_1) = \{WRITE\}$), and has a property key $date$ whose value is 1997 (i.e., $\sigma(r_1, date) = 1997$).

B. Cypher & Graph Pattern Matching

Cypher is developed by Neo4j [35] and has been supported by many GDBs, e.g., MemGraph [9], SAP HANA [26], and NebulaGraph [50]. Cypher is a declarative graph query language, which adopts graph pattern matching for querying nodes and relationships in property graphs.

Cypher utilizes `MATCH` clause to define a graph pattern (e.g., the graph pattern in Fig. 2). This graph pattern defines three node patterns, i.e., `(p1:Person)`, `(:Book)`, and `(p2:Person)`, and two relationship patterns, i.e., `-[x]->` and `<-[y]-`. It also contains a predicate `p1.age=59` specified in the `WHERE` clause. Cypher utilizes the `RETURN` clause to specify which parts of data should be returned. For example, `RETURN p1.name` in Fig. 2 indicates that the query result should be the values of the property $name$ of node $p1$.

Besides the above core features, Cypher also supports complex graph patterns. Cypher allows for multiple graph patterns in chained `MATCH` clauses, such as `MATCH (n1) MATCH (n2)-[]->(n3)`. The results matched by each graph pattern are combined through Cartesian product. Cypher provides `OPTIONAL MATCH` clause to define optional graph patterns that may or may not exist in a property graph. Cypher graph patterns can include variable-length paths, e.g., `(n1)-[*1..3]->(n2)` matches a path with 1 to 3 relationships. Cypher also supports sorting and aggregates on the query result, such as `ORDER BY` or `COUNT()`. Furthermore, multiple Cypher queries can be combined by `UNION ALL` clause that unions the single query results under bag semantics.

Cypher graph pattern matching. Graph pattern matching serves as the foundation of graph query languages (e.g., Cypher and Gremlin). Graph pattern matching maps node and relationship patterns to nodes and relationships in property graph, e.g., node pattern `(p1:Person)` in the Cypher query of Fig. 2 can match node n_1 in the property graph of Fig. 1. Cypher adopts isomorphism-based graph pattern matching to query property graphs as follows.

Definition 2. *Given a property graph $G = \langle N, R, \rho, \lambda, \sigma \rangle$, a graph pattern on G is defined as $G_p = \langle N_p, R_p, \rho_p, \phi_p \rangle$. Here N_p is a finite set of node patterns, R_p is a finite set of relationship patterns, $\rho_p : R_p \rightarrow N_p \times N_p$ is a function that projects a relationship pattern to its outgoing and incoming node patterns, and ϕ_p is a boolean expression constructed on a finite set of node patterns from N_p and relationship patterns from R_p. A graph pattern matching aims to find all possible mappings f_n and f_r, which satisfy the following conditions.*

1) *$f_n : N_p \rightarrow N$ maps a node pattern to a node in G.*
2) *$f_r : R_p \rightarrow R$ is an injective mapping that maps a relationship pattern to a relationship in G.*
3) *For each $r_n \in R_n$ and $r_p \in R_p$, the specified labels in r_n and r_p are a subset of $f_n(n_p)$ and $f_r(r_p)$, respectively.*
4) *f_n and f_r are structure-preserving, i.e., for each $r_p \in R_p$, if $\rho_p(r_p) = \langle n_{p1} \in N_p, n_{p2} \in N_p \rangle$, $f_r(r_p) = r \in R$ and $\rho(r) = \langle n_1, n_2 \rangle$, then $f_n(n_{p1}) = n_1$ and $f_n(n_{p2}) = n_2$.*
5) *If each node pattern $n_p \in N_p$ and relationship pattern $r_p \in R_p$ used in ϕ_p are replaced by their corresponding mapped node $f_n(n_p)$ and relationship $f_r(r_p)$ in G, ϕ_p holds True.*

[1]GQL is a new standard graph query language that extends Cypher. Although there are some differences, GQL and Cypher share many similarities in both syntaxes and semantics. It would be interesting to perform a similar study for GQL in the future.

Note that Cypher assigns variables to node and relationship patterns (e.g., variable $p1$ in node pattern `(p1:Person)`) for their references. Node or relationship patterns that share the same variable are considered to match the same graph entity. Different from other graph query languages, Cypher applies injective mapping from relationship patterns to relationships in property graphs, i.e., they adopt the so-called relationship-injective semantics [11], [35]. Specifically, in Cypher queries, different relationship patterns (which are assigned to different variables) defined within the same `MATCH` clause are not allowed to match the same relationship in the property graph. Take the query in Fig. 2 as an example. Relationship-injective semantics require that relationship patterns x and y must map to different relationships in Fig. 1, e.g.,

$$f_n(p1) \to n_1,\ f_n(p2) \to n_3,\ f_r(x) \to r_1,\ f_r(y) \to r_2$$

In contrast, relationship-injective semantics do not permit the following mapping.

$$f_n(p1) \to n_1,\ f_n(p2) \to n_1,\ f_r(x) \to r_1,\ f_r(y) \to r_1$$

C. U-semiring & U-semiring SQL Semantics

U-semiring. Unbounded semiring (U-semiring for short) [16] is a natural number semiring with unbounded summation. U-semiring extends the commutative semiring [29] with new operators ($\sum, \|\cdot\|, not(\cdot)$) and is defined as follows.

Definition 3. *U-semiring is a commutative natural number semiring denoted as* $(\mathbb{N}, 0, 1, +, \times, \|\cdot\|, not(\cdot), \sum)$*, where*

1) *The squash operator* $\|\cdot\|$ *is unary and transforms an input value into an output between 0 and 1, e.g.,* $\|0\| = 0$, $\|1 + x\| = 1$. *Squash operator is commonly used to deduplicate query results under bag semantics.*
2) *The* $not(\cdot)$ *operator is unary and reverses an input boolean value, e,g,* $not(1) = 0$, $not(0) = 1$.
3) $\sum_{t \in D} E(t)$ *takes an expression* $E(t)$ *as input and outputs a natural value. Specifically,* $\sum$ *is used to enumerate all variable (set)* t *within a given domain* D *for* $E(t)$ *and accumulate the output values.*

U-semiring also adopts semiring operator $[\phi]$ from K-relation [29] that takes a boolean expression ϕ as input, and outputs 1 for true, 0 for false. $[\phi]$ transforms a boolean value into integer for arithmetic $\times$ and $+$ under semiring semantics.

U-semiring SQL algebraic representation. Based on U-semiring, Chu et al. [16] propose an algebraic representation to model SQL queries under bag semantics and construct U-semiring expressions for SQL (*U-expression* for short). *U-expression* defines table function $R(t)$ to return the multiplicity of tuple t in table R. Then *U-expression* models a SQL query as an expression $u(t)$ that returns the multiplicity of tuple t in the query result. For example, a SQL query

```
SELECT c1 FROM R WHERE c2 = 1
```

is modeled as a *U-expression*

$$u(t) = \textstyle\sum_{t_1} [t = t_1.c1] \times R(t_1) \times [t_1.c2 = 1]$$

This *U-expression* returns the multiplicity of an arbitrary tuple t in the query result by counting all possible tuple t_1 in table R using $\sum_{t_1}$, and filters the predicate $t_1.c2 = 1$ using $[t_1.c2 = 1]$. $[t = t_1.c1]$ expresses the projection that each tuple t in the query result is the column $c1$ of a tuple t_1 in R.

U-expression can also model complex SQL features. For subqueries combined by `UNION`, *U-expression* recursively models the subqueries and combines them using semiring operator $+$. For tables joined together, *U-expression* multiplies table functions, e.g., $R(t_1) \times S(t_2)$.

By using *U-expression*s, proving the equivalence of SQL queries is transformed into proving the equivalence of *U-expression*s. To prove the equivalence of *U-expression*s, UDP [16] formalizes *U-expression*s using axioms, creating a canonical form that allows for isomorphism checking between *U-expression*s. Additionally, SQLSolver [22] leverages LIA* theory to eliminate $\sum$ in *U-expression*s, modeling them as first-order logical expressions that are verified by SMT solvers.

III. Overview

To prove the equivalence of Cypher queries, we first formalize the problem of Cypher query equivalence under bag semantics in Section III-A. We then explain the basic idea of modeling the Cypher semantics based on U-semiring in Section III-B. Finally, we introduce the workflow of our Cypher equivalence prover in Section III-C.

A. Problem Formulation

Cypher queries return tabular results under bag semantics that allow for duplicates. Two Cypher queries Q_1 and Q_2 are equivalent if their results consist of the same tuples, and the multiplicity of any tuple t in their results is equal. Additionally, if Cypher queries return ordered results (e.g., queries with `ORDER BY` clauses), any two tuples t_1 and t_2 appear in the same order in the results of both queries Q_1 and Q_2.

For two Cypher queries Q_1 and Q_2 with `ORDER BY` clauses, if their corresponding sub-queries $Q_1^{'}$ and $Q_2^{'}$ without `ORDER BY` clauses are equivalent, and Q_1 and Q_2 sort the query results returned by $Q_1^{'}$ and $Q_2^{'}$ according to the same `ORDER BY` expressions, we can say that Q_1 and Q_2 are equivalent.

Therefore, the Cypher query equivalence under the ordered bag semantics can be defined as follows.

Definition 4. *If two Cypher queries* Q_1 *and* Q_2 *are equivalent, they need to satisfy the following conditions.*

1) *The results of* Q_1 *and* Q_2 *contain the same tuples.*
2) *The multiplicity of any tuple* t *in the results of* Q_1 *and* Q_2 *is equal.*
3) *If* Q_1 *or* Q_2 *contains the* `ORDER BY` *clauses, their corresponding sub-queries* $Q_1^{'}$ *and* $Q_2^{'}$ *without* `ORDER BY` *clauses are equivalent, and* Q_1 *and* Q_2 *sort the query results returned by* $Q_1^{'}$ *and* $Q_2^{'}$ *according to the equivalent* `ORDER BY` *expressions.*

B. Basic Idea

Inspired by the *U-expression*s for modeling SQL queries, we propose an approach to model a Cypher query as a *G-expression* $g(t)$, which takes a tuple t in the Cypher query

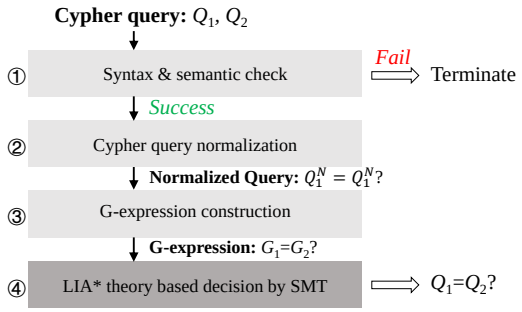

Fig. 3. The workflow of GraphQE.

result as input and outputs its multiplicity. Then two Cypher queries Q_1 and Q_2 are considered semantically equivalent if their corresponding *G-expression*s $g_1(t)$ and $g_2(t)$ always output the same value for any tuple t on an unspecified property graph.

To model a Cypher query based on U-semiring, we first assign a variable e_i to represent any property graph entity for each node pattern and relationship pattern in a graph pattern $G_p = \langle N_p, R_p, \rho_p, \phi_p \rangle$. We then model the Cypher query as a *G-expression* based on these variables. Finally, we can calculate the multiplicity of a tuple t in the Cypher query result by enumerating all possible graph entities $(e_1, \ldots, e_n)$ over an unspecified property graph G, which satisfy graph pattern G_p.

For example, given a simple Cypher query `MATCH (n1)-[r]->(n2) WHERE n1.age=59 RETURN n1`, we assign three variables e_1, e_2, e_3 to node pattern `(n1)`, relationship pattern `-[r]->`, and node pattern `(n2)`, respectively. Then, we model the Cypher query into a *G-expression* as

$$g(t) = \sum_{e_1,e_2,e_3} [t = e_1] \times Node(e_1) \times Rel(e_2) \times Node(e_3) \times [in(e_2) = e_1] \times [out(e_2) = e_3] \times [e_1.age = 59]$$

in which $Node(e)$, $Rel(e)$, $in(e)$ and $out(e)$ denote whether e is a node entity, relationship entity, e's incoming node and outgoing node, respectively. $g(t)$ uses the unbounded summation (Σ) to enumerate all graph entities (e_1, e_2, e_3) over an unspecified property graph G and calculates the multiplicity of tuple t in the Cypher query result using $[t = e_1]$.

Note that $g(t)$ returns a natural number for any tuple t, and can illustrate the bag semantics of Cypher queries. Based on this idea, we further construct *G-expression*s for the predicates and sorting of a Cypher query and a set of complex Cypher features, e.g., the arbitrary-length paths (`(n1)-[*]->(n2)`). The problem of proving Cypher query equivalence is then transformed into that of proving the equivalence of their corresponding *G-expression*s, i.e., $g_1(t)$ and $g_2(t)$.

C. Workflow of GraphQE

We propose an approach to model Cypher queries based on our idea in Section III-B and implement it as GraphQE. GraphQE takes a pair of Cypher queries Q_1 and Q_2 as input and returns whether they are equivalent. As shown in Fig. 3, GraphQE consists of four stages.

① Syntax & semantic check. The input Cypher queries may contain syntax or semantic errors. Syntax errors occur when Cypher queries violate the Cypher grammar and semantic errors occur when Cypher queries use illegal operations. To ensure the correctness of the input Cypher queries, we perform both syntax and semantic checks, discarding any queries that contain errors. Specifically, for a given Cypher query, we first attempt to construct an Abstract Syntax Tree (AST) using a Cypher grammar parser that is built on the openCypher grammar [2]. Queries that fail to generate ASTs are considered to violate the Cypher grammar, and the proving process is terminated. For syntactically correct Cypher queries, we adopt a semantic check based on their ASTs. We check the following cases: (1) Incorrect variable reference. Undefined variable references in `WHERE` clause cause semantic errors. (2) Incorrect relationship labels. In property graphs, each relationship can have only one label. Relationship patterns that share the same variable but define different labels can lead to semantic errors. Similar to syntax checking, for queries that fail the semantic check, we terminate the proving process.

② Cypher query normalization. Since some complex Cypher queries (e.g., those with variable-length paths) are difficult to be directly modeled and some Cypher queries can be isomorphic after simple query transforms, we apply a rule-based Cypher query normalization to simplify these queries into forms that we can support. Specifically, we define a group of normalization rules on the Cypher ASTs to handle built-in functions, complex features and reform the ASTs. We will further discuss our rule-based Cypher query normalizations in Section V.

③ *G-expression* construction. For normalized Cypher queries Q_1^N and Q_2^N, we model them as U-semiring Cypher expressions (*G-expression* for short) $G_1 = g_1(t)$ and $G_2 = g_2(t)$, which model the multiplicity of tuple t in their query results. By utilizing *G-expression*s, we transform the problem of proving Cypher query equivalence into proving the equivalence of $g_1(t)$ and $g_2(t)$. We will further discuss our *G-expression*s in Section IV.

④ Decision procedure. Once Cypher queries are modeled into *G-expression*s, we prove the equivalence of *G-expression*s by proving that $\exists t.g_1(t) \neq g_2(t)$ is unsatisfiable using the Z3 SMT solver [19]. Since SMT solvers cannot handle the unbounded summations (i.e., $\sum$) in *G-expression*s, we leverage the LIA* based algorithm proposed by Ding et al. [22] to eliminate the unbounded summations and construct first-order logical expressions for proving their satisfiability by Z3.

IV. G-EXPRESSION

To model Cypher queries as *G-expression*s, we first model the property graph model under semiring semantics (Section IV-A). Then we model core and advanced Cypher features, and construct *G-expression*s for Cypher queries (Section IV-B). Finally, we illustrate the decision procedure of proving the equivalence of *G-expression*s (Section IV-C).

A. Modeling Property Graph

Property graph entities are stored as nodes and relationships under set semantics, so that the multiplicity of each property graph entity is one. Therefore, we only need to distinguish whether a property graph entity is a node or a relationship. For an arbitrary property graph entity e in an unspecified property graph $G = \langle N, R, \rho, \lambda, \sigma \rangle$, we define $Node(e)$ and $Rel(e)$ functions to model nodes and relationships as follows.

$$Node : N \cup R \to \mathbb{B} = \{1, 0\},\ Rel : N \cup R \to \mathbb{B} = \{1, 0\}$$

$$Node(e) = \begin{cases} 1 & \text{if } e \in N \\ 0 & \text{otherwise} \end{cases} \quad Rel(e) = \begin{cases} 1 & \text{if } e \in R \\ 0 & \text{otherwise} \end{cases}$$

Each relationship in a property graph connects two nodes from its outgoing node to its incoming node. For a relationship e in an unspecified property graph $G = \langle N, R, \rho, \lambda, \sigma \rangle$, we define $out(e)$ and $in(e)$ functions to express a relationship e's outgoing and incoming nodes, respectively, as follows.

$$out : R \to N,\ in : R \to N$$

Specifically, if e_1 and e_2 are the outgoing and incoming nodes of e, respectively, i.e., $\rho(e) = \langle e_1, e_2 \rangle$, then

$$out(e) = e_1,\ in(e) = e_2$$

To express the multiplicity of relationships' outgoing and incoming nodes, we use U-semiring operator $[\cdot]$ on $out(e)$ and $in(e)$. For example, $[out(e) = e_1]$ returns 1 if e_1 is the outgoing node of relationship e, and 0 otherwise. Note that, $out(e)$ and $in(e)$ are always used together with $Rel(e)$ to ensure e is a complete relationship in a property graph.

Property graphs utilize labels to categorize nodes (relationships) into different node (relationship) sets. To model labels of each property graph entity, we define a function $Lab(e, label)$ to represent that a property graph entity e has a label $label$. Specifically, for an arbitrary graph entity e in an unspecified property graph $G = \langle N, R, \rho, \lambda, \sigma \rangle$, we define $Lab(e, label)$ as follows.

$$Lab : (N \cup R) \times \mathcal{L} \to \mathbb{B} = \{1, 0\}$$

Specifically, for the labels of e, denoted as $\lambda(e)$, we have

$$Lab(e, label) = \begin{cases} 1 & \text{if } label \in \lambda(e) \\ 0 & \text{otherwise} \end{cases}$$

Property graph entities have a set of properties. We model each property of a property graph entity by defining $x.key$ to access the property value of a given property name key. For example, we use $e.p_1$ to access the value of e's property name p_1, and $[e.p_1 = v_1]$ returns 1 if v_1 is the value of $e.p_1$ and 0 otherwise.

B. Modeling Cypher Query Features

Based on the predefined functions in Section IV-A, we model Cypher queries as U-semiring based Cypher expressions (*G-expression* for short). Fig. 4 shows the Cypher fragments we support currently. We first model core features of Cypher queries, e.g., graph patterns, predicates and query results. We then model a set of advanced Cypher features, e.g., join of graph patterns.

$$\begin{aligned}
q \in \text{Query} &::= m\ r \mid q_1 \text{ UNION } q_2 \mid q_1 \text{ UNION ALL } q_2 \\
m \in \text{Match} &::= \text{MATCH } p_1, \ldots, p_n \\
&\mid \text{OPTIONAL MATCH } p_1, \ldots, p_n \\
&\mid m \text{ WHERE } b \mid m \text{ WITH } b_1, \ldots b_n \\
r \in \text{Return} &::= b \mid \text{DISTINCT } r \mid r \text{ ORDER BY } b \\
&\mid r \text{ LIMIT } b \mid r \text{ SKIP } b \\
p \in \text{GraphPattern} &::= n \mid n\ c \\
n \in \text{Node} &::= (v : l_1 : \ldots : l_n\ \{a_1 \ldots a_n\}) \\
v \in \text{Variable} &::= \text{string} \\
l \in \text{Label} &::= \text{string} \\
a \in \text{Property} &::= \text{string} \\
c \in \text{Relationship} &::= p\text{-}[v : l_1 : \ldots : l_n\ \{a_1 \ldots a_n\} * i_1 \ldots i_2]\text{->}n \\
&\mid p\text{<-}[v : l_1 : \ldots : l_n\ \{a_1 \ldots a_n\} * i_1 \ldots i_2]\text{-}n \\
&\mid p\text{-}[v : l_1 : \ldots : l_n\ \{a_1 \ldots a_n\} * i_1 \ldots i_2]\text{-}n \\
i \in \text{Constant} &::= \text{integer} \\
b \in \text{Expression} &::= b_1 = b_2 \mid b_1 \neq b_2 \\
&\mid \text{NOT } b \mid b \text{ IS NULL} \mid +b \mid -b \mid (b) \\
&\mid b_1 \text{ AND } b_2 \mid b_1 \text{ OR } b_2 \\
&\mid b_1 > b_2 \mid b_1 < b_2 \mid b_1 \geq b_2 \mid b_1 \leq b_2 \\
&\mid \text{TRUE} \mid \text{FALSE} \mid \text{EXISTS } q \\
&\mid v \mid v.a \mid f(b) \mid \text{agg}(b) \\
&\mid * \mid b \text{ AS } s \mid p_1, p_2 \mid i \mid \text{UNWIND } b \\
agg &::= \text{COLLECT} \mid \text{COUNT} \mid \text{SUM} \mid \text{MAX} \mid \text{MIN} \mid \text{AVG} \\
s \in \text{Alias} &::= \text{string} \\
f \in \text{UDF} &::= \text{string}
\end{aligned}$$

Fig. 4. Cypher fragments supported by `GraphQE`.

1) Modeling the Core Features of Cypher Queries: Cypher provides several core features to form a basic query for graph pattern matching, including graph patterns, predicates, and query results, as shwon in Fig. 2.

Modeling Cypher graph patterns. Cypher graph patterns include node and relationship patterns. We model each node or relationship pattern as an algebraic expression, which outputs 1, if an arbitrary graph entity e satisfies this pattern; otherwise outputs 0.

A complete node pattern in a Cypher query consists of a variable that refers to this node pattern, labels and properties with their values. To model a node pattern, we assign an arbitrary graph entity e to it and construct an algebraic expression using predefined functions. Specifically, we use $Node(e)$ to represent that property graph element e has the type of node, use $Lab(e, label)$ to represent e has label $label$, and use $[e.key = value]$ to represent the value of its property key is $value$. Furthermore, we use AND (i.e., $\times$) operations to join multiple labels and multiple properties. For example, a node pattern `(n:l1:l2{p1:v1})` is assigned a variable n, and has two labels $l1$ and $l2$ and a property $p1$ with its value $v1$. We can model this node pattern as $Node(e) \times Lab(e, l1) \times Lab(e, l2) \times [e.p1 = v1]$. Note that the variable (e.g., `n`), the labels (e.g., `:l1:l2`), and the properties (e.g., `{p1:v1}`) can be omitted in a node pattern. For example, `()` is the simplest node pattern, and we only use $Node(e)$ to model it.

A complete relationship pattern in a Cypher query consists

of a variable that refers to this relationship pattern, labels and properties with their values, and its outgoing and incoming node patterns. To model a relationship pattern, we assign an arbitrary graph entity e to it and construct an algebraic expression using predefined functions. Specifically, we use $Rel(e)$ to represent that property graph pattern e has the type of relationship, use $Lab(e, label)$ to represent e has label $label$, use $[e.key = value]$ to represent the value of its property key is $value$, and use $[out(e) = e_1]$ and $[in(e) = e_2]$ to represent e's outgoing node pattern is e_1 and incoming node pattern is e_2, respectively. For multiple properties, we use AND (i.e., $\times$) operations to join them. However, different from node patterns, for multiple labels in a relationship pattern, we use OR (i.e., $+$) operations instead of AND (i.e., $\times$) operations to connect them, because a relationship can have only one label. For example, a relationship pattern `(n1)-[r:l1|l2{p1:v1}]->(n2)` is assigned a variable r, has two labels $l1$ and $l2$, a property $p1$ with its value $v1$, and its outgoing and incoming node patterns are assigned variables $n1$ and $n2$, respectively. We can model this relationship pattern as $Rel(r) \times [out(r) = n2] \times [in(r) = n1] \times (Lab(r, l1) + Lab(r, l2)) \times [r.p1 = v1]$.

Cypher adopts relationship-injective semantics for the relationship patterns that require relationship patterns with different variables to match different relationships in the property graph. To model it, we construct not equal predicates to each pair of relationships defined within the same `MATCH` clause. For example, for a graph pattern `MATCH ()-[r1]->()-[r2]->()`, we first assign e_1 and e_2 to relationship pattern `()-[r1]->()` and `()-[r2]->()`, respectively. Then we construct expression $not([e_1 = e_2])$ for restricting that the two relationship patterns must match the different relationships. Note that relationships defined in different `MATCH` clauses are not restricted by relationship-injective semantics, e.g., $r1$ and $r2$ in `MATCH ()-[r1]->() MATCH ()-[r2]->()`.

Modeling predicates. Predicates define filtering conditions in a Cypher query. We leverage the semiring operator $[\cdot]$ to model predicates. For example, we model the predicate in Fig. 2 using $[e.age = 59]$, which returns 1 if the property age of entity e is 59, and 0 otherwise. Multiple predicates are connected by $\times$ for `AND` operations and $+$ for `OR` operations. For example, we model `WHERE p1.age>29 OR p1.age<59` under semiring semantics as $\|[e_1.age > 29] + [e_1.age < 59]\|$.

Modeling query results. Cypher queries return tabular results that consist of a set of tuples, each of which has a set of column values. We model each tuple in the result table using $t.col_i$ and construct the projections from graph entities to the value of $t.col_i$. For example, in Fig. 2, the tuple in the query result has one column, which projects an arbitrary graph entity e_1 for node pattern $p1$ to the value of its property $name$. Thus, we model this query result using $[t.col_1 = e_1.name]$.

2) Modeling the Advanced Cypher Features: Cypher is highly expressive and provides a set of advanced features or clauses to enrich its expressiveness. As shown in Table I, we model a set of advanced Cypher features and construct *G-expression*s for them.

Intermediate results. The `WITH` clause creates intermediate results in a query. It generates an intermediate table as the input of the subsequent clause. For each intermediate table, we introduce an intermediate variable t'_i to model the multiplicity of tuples in the table in the same way as `RETURN`. For Cypher queries with the structure: `MATCH` p_1 `WITH` $i_1, i_2...$ `MATCH` p_2 `RETURN` $e_1, e_2...$, we model it as $g(t) = \sum_{t', \vec{e_1}}(E_2 \times [t.col_1 = e_1] \times ... \times \sum_{\vec{e_2}}(E_1 \times [t'.col_1 = i_1] \times ...))$, where E_1 represents the *G-expression* for p_1, E_2 represents the *G-expression* for p_2, and t' is the intermediate variable. Furthermore, temporary variables are also used to model Cypher nested subqueries, such as `MATCH (n) WHERE EXISTS MATCH (n1) WHERE n1.p1 > 100 RETURN n`.

However, temporary variables may lead to semantic loss. For example, for the Cypher query `MATCH (p) WITH DISTINCT p.name AS name RETURN name`, we construct the following *G-expression*s for it.

$$g(t) = \sum_{t'}[t = t'] \times \| \sum_{e_1} Node(e_1) \times [t' = e_1.name] \|$$

The *G-expression*s cannot be correctly solved by SMT solvers since t' lost the uniqueness of the query results created by `WITH DISTINCT p.name`. Therefore, we propose a normalization rule to eliminate temporary variables. Specifically, we remove temporary variable t' from the unbounded summation if it can be represented by other variables using $[t' = t_1]$. After normalization, Q's *G-expression* becomes

$$g(t) = \| \sum_{e_1} Node(e_1) \times Lab(e_1, Person) \times [t = e_1.name] \|$$

Intermediate variables can cause predicates to deviate from their intended scopes, e.g., the $[e_1.dept = e_2.dept]$ in

$$g(t) = \sum_{e_1}[t = e_1] \times Node(e_1) \times [e_1.dept = e_2.dept] \times \| \sum_{e_2} Node(e_2) \|$$

To address this, we propose a normalization rule that moves the predicate to the correct summation body as follows.

$$g(t) = \sum_{e_1}[t = e_1] \times Node(e_1) \times \| \sum_{e_2} Node(e_2) \times [e_1.dept = e_2.dept] \|$$

Aggregate. `GraphQE` models aggregates using intermediate variables. For example, for aggregate `SUM` in the Cypher query segment `MATCH (n:Person) RETURN SUM(n.age)`, `GraphQE` creates variable t' and model it as

$$g(t) = [t = \sum_{t'}[t' = e_1.name] \times Node(e_1) \times Lab(e_1, Person) \times e_1.sal]$$

`GraphQE` does not model the concrete semantics of `COLLECT` but represents it using an uninterpreted function in the SMT solver.

Unwinding. The `UNWIND` clause transforms a list into individual rows. The list for unwinding can be a constant list created by `WITH` or collected from `COLLECT`. As shown in Table I, we model unwinding on a constant list by modeling the concatenation of each element in it using $+$. Unwinding a collected list will break it into individual rows and remove the aggregates, e.g., `UNWIND(COLLECT(n.name))` is directly modeled into `n.name`.

TABLE I
MODELING ADVANCED CYPHER FEATURES BY *G-expression*S.

Cypher feature	**Query example**	***G-expression***
Intermediate results	`MATCH (n) WITH n.name AS name RETURN name`	$\sum'_t([t = t'] \times \sum_{t'}([t' = e.name] \times Node(e)))$
Aggregate	`MATCH (n:Person) RETURN SUM(n.age)`	$g(t) = [t = \sum_{t'}[t' = e_1.name] \times Node(e_1) \times Lab(e_1, Person) \times e_1.sal$
Unwinding	`WITH [{c1:0, c2:1},{c1:2, c2:3}] AS tmp UNWIND tmp AS tmpRow`	$([tmpRow.c1 = 0] \times [tmpRow.c2 = 1]) + ([tmpRow.c1 = 2] \times [tmpRow.c2 = 3])$
Arbitrary-length path	`()-[*]->()`	$Rel(e) \times UNBOUNDED(e) \times [out(e) = e_1] \times [in(e) = e_2]$
Sorting with truncation	`WITH x.name ORDER BY x.age RETURN 1`	$[t = e.name] \times [asc(t) = e.age] \times [limit(t) = 1]$
Natural join	`MATCH...`(q_1) `MATCH...`(q_2)	$G(q_1) \times G(q_2)$
Left outer join	`MATCH...` (q_1) `OPTIONAL MATCH...` (q_2)	$G(q_1) \times G(q_2) + G(q_1) \times not(G(q_2)) \times isNULL(G(q_2))$
Union all	`MATCH...` (q_1) `UNION ALL MATCH...` (q_2)	$G(q_1) + G(q_2)$
Union	`MATCH...` (q_1) `UNION MATCH...` (q_2)	$\|\|G(q_1) + G(q_2)\|\|$

Arbitrary-length path. Cypher uses `(n1)-[*]->(n2)` to retrieve all paths of any length between two nodes $n1$ and $n2$. In this case, $n1$ and $n2$ are fixed and the relationships in the paths must satisfy the same predicate. Therefore, we treat arbitrary-length path patterns (i.e., `()-[*]->()`) as a special kind of relationship pattern, and assign a relationship variable e to this pattern, in which e_1 and e_2 represent its outgoing and incoming node patterns, respectively. Then, we define a function $UNBOUNDED(e)$ on e that outputs 1 if e is a combination of any number of relationships. Finally, we construct *G-expression* term $Rel(e) \times UNBOUNDED(e) \times [out(e) = e_1] \times [in(e) = e_2]$ to model this pattern. Besides, labels specified on the arbitrary-length path can also be supported through this approach. For example, we model `-[*:KNOWS]->` as $UNBOUNDED(e) \times Lab(e, KNOWS)$.

Sorting with truncation. Cypher sorts and truncates the query results using the `ORDER BY...LIMIT...SKIP...` fragments. To model `ORDER BY` $o_1, ..., o_n$, we follow an idea that a condition on the query result can be represented as the same condition on all the tuples within the query result. Therefore, we treat the $o_1, ..., o_n$ as conditions on all the tuples in the query result. We define function $order(t, i)$ on an arbitrary tuple t in the query result to represent o_i. Then, we construct $[order(t, i) = o_i]$ to model the value of o_i. Finally, we use $\times$ to connect all the $[order(t, i) = o_i]$, i.e., $[order(t, 1) = o_1] \times \cdots \times [order(t, n) = o_n]$. To model `LIMIT` l and `SKIP` s, we also treat them as conditions on all the tuples in the query result. We define functions $limit(t)$ and $skip(t)$ that represent the limiting and skipping condition on all the tuple t in the Cypher query result. Then, we model `LIMIT` l and `SKIP` s as $[limit(t) = l]$ and $[skip(t) = s]$.

For the `ORDER BY...LIMIT...SKIP...` fragments among Cypher subqueries, we consider the following cases: (1) single `ORDER BY` is ignored since the order it specifies will not be guaranteed by the following clauses. (2) For `ORDER BY` followed by `LIMIT` and `SKIP`, we cannot directly model it into a single *G-expression*. Instead, we design a divide-and-conquer based approach that individually check the equivalence of their subqueries. For example, for the equivalent Cypher queries in Listing 2, we divide each query into subqueries, i.e., $Q1$ into $Q1'$ and $Q1''$, $Q2$ into $Q2'$ and $Q2''$. Then we check the equivalence of each pair of subqueries, i.e., $\langle Q1', Q2' \rangle$, and $\langle Q1'', Q2'' \rangle$.

```
1 Q1:   MATCH (n1) WITH n1 ORDER BY n1.p1 LIMIT 1
2       MATCH (n1)-[]->(n2) RETURN n2
3 Q1':  MATCH (n1) WITH n1 ORDER BY n1.p1 LIMIT 1
4 Q1'': MATCH (n1) MATCH (n1)-[]->(n2) RETURN n2
5
6 Q2:   MATCH (n1) WITH n1 ORDER BY n1.p1 LIMIT 1
7       MATCH (n2)<-[]-(n1) RETURN n2
8 Q2':  MATCH (n1) WITH n1 ORDER BY n1.p1 LIMIT 1
9 Q2'': MATCH (n1) MATCH (n2)<-[]-(n1) RETURN n2
```

Listing 2. Equivalent Cypher queries with `ORDER BY...LIMIT...` within their subqueries.

Product/Concatenation. Cypher graph patterns defined in multiple `MATCH` are joined through Cartesian product, and we recursively construct *G-expression*s (denoted as $G(s_i)$ in Table I) for each `MATCH` and connect them using $\times$. Cypher graph patterns are left outer joined by `OPTIONAL MATCH`, and we model the graph patterns that can be NULL using uninterpreted function $isNULL$ in the SMT solver. In the similar way, we use $+$ for the concatenation of recursively constructed *G-expression*s of Cypher subqueries in `UNION ALL` and use an extra $||\cdot||$ for `UNION`.

C. *Proving the Equivalence of* G-expression

By constructing *G-expression*s for Cypher queries, we transform the problem of proving the equivalence of Cypher queries into proving the equivalence of *G-expression*s. We first map the returned elements across the two Cypher queries. This process prevents query inequality caused by difference in the order of returned elements. For example, consider two Cypher queries as follows.

Q_1: `MATCH (n1)-[r]->(n2) RETURN n1, n2`
Q_2: `MATCH (n1)<-[r]-(n2) RETURN n1, n2`

In Q_1, the returned element $n1$ should be mapped to the returned element $n2$ in Q_2, and should not be affected by their orders. To achieve this, we pair the returned elements in both queries according to their types. Specifically, node or relationship variables are mapped to those of the same type, expressions are mapped to expressions of the same type, and references to variable properties are mapped to references with the same name for variables of the same type. If no successful mappings are found, we will remove these conditions and map again. If the numbers of elements returned by the two

Cypher queries are inconsistent, then the two queries can only be equivalent if they both return empty results on any property graph. Therefore, we directly prove whether the two queries satisfy this case.

To prove the equivalence of *G-expression* $g_1(t)$ and $g_2(t)$, we prove that $\exists t.g_1(t) \neq g_2(t)$ is unsatisfiable through Z3 SMT solver. Since SMT solvers cannot handle unbounded summations, we leverage LIA* theory based algorithm [22], [37] to eliminate unbounded summations. LIA* formula extends Linear Integer Arithmetic to represent unbounded summations, allowing for reasoning unbounded summations using SMT solvers. The algorithm replaces each unbounded summations with an integer value v_i and finds an equisatisfiable formula to represent the unbounded summations.

For example, given two *G-expression*s

$$g_1(t) = \sum_{e_1}[t = e_1.name] \times Node(e_1) \times ([e_1.age < 10] + [e_1.age > 20])$$
$$g_2(t) = \sum_{e_1}[t = e_1.name] \times Node(e_1) \times [e_1.age < 10] + \sum_{e_2}[t = e_2.name] \times Node(e_2) \times [e_2.age > 20]$$

the algorithm replaces the unbounded summations in $g_1(t)$ and $g_2(t)$ and models them as v_1 and $v_2 + v_3$. It finds an equisatisfiable formula of $\exists v_1, v_2, v_3.v_1 \neq v_2 + v_3 \wedge (v_1, v_2, v_3) = \lambda_1(1, 0, 1) + \lambda_2(0, 1, 0)$, which can be proved unsatisfiable by SMT solvers since such λ_1 and λ_2 do not exist.

V. Rule-based Cypher Query Normalization

Cypher provides some complex features that cannot be directly modeled by the approach in Section IV (e.g., variable-length paths (`()-[*1..2]->()`) and `RETURN *`. However, we observe that these features can be represented by the combinations of features in Section IV. Therefore, we propose a group of normalization rules that transform queries containing these complex features into equivalent queries using only the features we have modeled. Specifically, each normalization rule traverses the Cypher AST, matches specific query fragments, and transforms them into simplified equivalent Cypher fragments.

Table II shows the normalization rules to transform complex features. Rule ① transforms an undirected relationship in a Cypher query to the union of relationships with both directions. This is because Cypher does not support querying undirected edges. Instead, undirected edges are parsed as two directed edges in opposite directions. Rule ② transforms variable-length paths into the union of all the lengths. Rule ③ to ⑤ aims to fill the omitted but determined parts of a Cypher query, e.g., `RETURN *`. Rule ⑥ converts primary key equivalences into variable equivalences based on the integrity constraints of the graph database.

We normalize a Cypher query by sequentially applying these normalization rules round by round until no rule is successfully applied. Only one rule is applied per round to avoid conflicts. Specifically, ⑤ is applied after ②, ③ and ④, since ② and ④ can create anonymous node or relationship patterns and copy the existing patterns that should be standardized by ⑤. Since ⑤ assign variables to anonymous graph patterns, it is applied after ③. We apply ⑥ after ⑤, because we should not replace the variable names of node or relationship patterns across subqueries since they are indeed different.

VI. Soundness and Completeness

Soundness. We now show that `GraphQE`, our approach for checking the equivalence of Cypher queries, is sound.

Theorem 1. Let Q be a simple Cypher query, which is defined by the Cypher fragments shown in Fig. 4, excluding arbitrary-length path, built-in functions, `ORDER BY`, `LIMIT`, and `SKIP`. Let $g(t)$ be the *G-expression* representation of Q. Then, for any tuple t, $g(t)$ returns the multiplicity of t in the evaluation of Q over a property graph G under Cypher bag semantics.

Proof sketch. According to Definition 2, Cypher queries adopt graph pattern matching that finds all maps from each node/relationship pattern ($N_p \cup R_p$ in G_p) to structure-preserving node/relationship in a property graph ($N \cup R$ in G) and satisfy condition ϕ_p. *G-expression* $g(t)$ uses algebraic functions and semiring operations to translate each Cypher feature in Q into natural number semiring semantics, which computes the multiplicity of each tuple t by counting all the combinations of property graph elements matched in G. For example, given a simple Cypher query Q: `MATCH (n1)-[r]->(n2) RETURN n1`, `GraphQE` maps the graph pattern in Q into the product of algebraic functions as $Node(n1) \times Rel(r) \times Node(n2) \times [in(r) = n1] \times [out(r) = n2]$ on semiring semantics that is structure preserving. Then, `GraphQE` maps the graph pattern matching in Q into $\sum_{n1,r,n2}$ that enumerates all the combinations of property graph elements and calculates the multiplicity of tuple t in Q's query result using $[t = n1]$. Other simple Cypher features are modeled in the similar way. Therefore, $g(t)$ returns the multiplicity of any tuple t in Q's query result. □

Theorem 2. Given two Cypher queries Q_1 and Q_2 and their corresponding translated *G-expression*s $g_1(t)$ and $g_2(t)$, if $g_1(t)$ and $g_2(t)$ are equivalent, then Q_1 and Q_2 are equivalent.

Proof sketch. We prove this theorem by examining three different cases.

1) Q_1 and Q_2 do not contain arbitrary-length paths, built-in functions, sorting and truncation. According to Theorem 1, $g_1(t)$ and $g_2(t)$ return the multiplicities of arbitrary tuple t in their query results. If the SMT solver proves $\exists t.g_1(t) \neq g_2(t)$ is unsatisfiable, then for an arbitrary tuple t, the multiplicities of t in the query results of Q_1 and Q_2 are the same. Thus, Q_1 and Q_2 are equivalent.
2) Q_1 and Q_2 contain arbitrary-length paths, built-in functions, sorting and truncation outside subqueries. `GraphQE` models these features as uninterpreted functions in $g_1(t)$ and $g_2(t)$. The equivalence of $g_1(t)$ and $g_2(t)$ implies that Q_1 and Q_2 use these features in the same way. This implies that the equivalence of $g_1(t)$ and $g_2(t)$ is the sufficient condition for the equivalence of Q_1 and Q_2.

TABLE II
NORMALIZATION RULES USED FOR DE-SUGARING COMPLEX CYPHER QUERIES.

No.	Normalization rule	Original query	Normalized query
①	Eliminating undirected relationship pattern	`MATCH (n1)-[]-(n2) RETURN n1.name`	`MATCH (n1)-[]->(n2) RETURN n1.name UNION ALL MATCH (n1)<-[]-(n2) RETURN n1.name`
②	Rewriting variable-length path	`MATCH (n1)-[*1..2]->(n2) RETURN n1`	`MATCH (n1)-[]->(n2) RETURN n1 UNION ALL MATCH (n1)-[]->()-[]->(n2) RETURN n1`
③	Rewriting `RETURN *`	`MATCH (x)-[z]->()-[y]->() RETURN *`	`MATCH (x)-[z]->()-[y]->() RETURN x, y, z`
④	Eliminating redundant clause	`MATCH (x) WITH x.name AS name RETURN name`	`MATCH (x) RETURN x.name`
⑤	Standardizing variable	`MATCH (person)-[]->(book) RETURN person`	`MATCH (n1)-[r1]->(n2) RETURN n2`
⑥	ID equality simplification	`MATCH (n1), (n2) WHERE id(n1)=id(n2) RETURN n2`	`MATCH (n1) RETURN n1`

3) Q_1 and Q_2 contain sorting and truncation within subqueries. GraphQE divides Q_1 and Q_2 into subqueries $Q_1^1 \dots Q_1^m$ and $Q_2^1 \dots Q_2^n$. Then, GraphQE requires $m = n$ and individually proves each Q_1^i and Q_2^i are equivalent, which forms a sufficient condition for the equivalence of Q_1 and Q_2. □

Completeness. GraphQE does not ensure completeness. That said, even if two Cypher queries are equivalent, we cannot ensure their corresponding translated $g_1(t)$ and $g_2(t)$ are equivalent. The incompleteness of GraphQE is caused by the following reasons. (1) GraphQE does not model all Cypher features, e.g., regular functions and CALL. (2) GraphQE does not support handling all cases for `ORDER BY...LIMIT...SKIP...` fragments in subqueries. Our divide-and-conquer based approach forms a sufficient but not necessary condition proving the equivalence of Cypher queries. (3) GraphQE utilizes uninterpreted functions to model arbitrary-length paths and built-in functions, which only form a sufficient but not necessary condition for proving the equivalence of Cypher queries. (4) GraphQE relies on the algorithm proposed by Ding et al [22] to prove the equivalence of *G-expression*s that is also incomplete.

VII. EVALUATION

We implement GraphQE with around 5000 lines of Java code. We use Antlr4 [40] to parse Cypher queries into Cypher Abstract Syntax Trees (ASTs) according to the grammar from openCypher [2]. We then transform the ASTs into graph relational algebra [30], [38]. We construct a *G-expression* for each Cypher query based on its graph relational algebra. The decision procedure is implemented based on the LIA* construction algorithm in SQLSolver [22] and utilizes Microsoft Z3 [19] to verify the equivalence of *G-expression*s.

To demonstrate the effectiveness of GraphQE, we address the following two research questions:

- **RQ1:** How effective is GraphQE's proving capability?
- **RQ2:** How is the performance of GraphQE on proving Cypher query equivalence?

To answer **RQ1**, we construct a dataset of 148 equivalent Cypher query pairs as CyEqSet and test the verification capability of GraphQE on it. To the best of our knowledge, GraphQE is the first equivalence prover for Cypher queries. Therefore, we cannot compare GraphQE with other provers. However, we analyze the verification result of GraphQE on all the Cypher features in CyEqSet.

To answer **RQ2**, we calculate the verification latency of GraphQE on each test case in CyEqSet and analyze why certain cases can have an extremely low or high latency.

A. Dataset Construction

Currently, there are no open-source datasets of equivalent Cypher queries available for evaluating our approach. Therefore, we construct a dataset, CyEqSet, of equivalent Cypher queries to evaluate GraphQE. Specifically, we construct CyEqSet through two approaches: (1) translating the open-source equivalent SQL query pairs from Calcite [4] into equivalent Cypher query pairs and (2) constructing equivalent Cypher query pairs by rewriting Cypher queries using existing equivalent rewriting rules.

Translation of SQL Calcite dataset. Calcite dataset [4] that contains 232 pairs of equivalent SQL queries is widely applied for evaluating SQL query equivalence [16], [17], [22], [49], [52], [53]. Researchers have proposed SQL-to-Cypher translating tools, e.g., the openCypherTranspiler [10]. However, these tools only support limited Cypher features, resulting in poor effectiveness when translating the Calcite dataset. Therefore, we manually translate each SQL query pair in Calcite dataset into a Cypher query pair and check the equivalence of the obtained Cypher query pairs based on the principles of Cytosm [45].

However, since the syntax of Cypher is significantly different from SQL, the following cases cannot be translated from SQL queries to Cypher queries. (1) We discard SQL queries having sorting operations inside and outside subclauses, since Cypher does not guarantee that the sorting of subqueries will be preserved. (2) We discard SQL queries adopting operations on the results of UNION or UNION ALL, which is not supported by Cypher. (3) We discard SQL queries containing GROUP BY that cannot be represented as DISTINCT in

TABLE III
PROVED QUERY PAIRS BY GRAPHQE.

Project	Query pairs	Proved
Calcite-Cypher	80	73
LDBC	13	13
Cypher-for-gremlin	23	23
Graphdb-benchmarks	32	29
Total	148	138

Cypher. Finally, we obtain 77 equivalent Cypher query pairs from Calcite in total.

Equivalent query transformation through rewriting rules. Since SQL queries do not have some specific Cypher features, e.g., arbitrary-length paths, the collected dataset via translating SQL queries cannot cover these Cypher features. Therefore, to enrich and expand our dataset, we first collect real-world Cypher queries from open-source projects, i.e., LDBC-snb-interactive-v1-impls [24] (a GDB benchmark), Cypher-for-gremlin [8] (a Cypher query plugin), and Graphdb-benchmarks [13] (a GDB benchmark), and we obtain 36 Cypher queries. Then, we apply three exiting Cypher rewriting rules on these Cypher queries as follows. (1) **Renaming variables**, which renames the variable names of node and relationship patterns. (2) **Reversing path direction**, which reverses the relationship patterns without changing their incoming and outgoing node patterns. (3) **Splitting graph pattern**, which splits multiple relationship patterns in a Cypher graph pattern.

We construct equivalent Cypher query pairs by applying these rules on each real-world Cypher queries. Note that some Cypher queries can be successfully rewrited by more than one rewriting rule, resulting in multiple equivalent Cypher query pairs. Finally, we generated 68 equivalent Cypher query pairs.

In total, we construct `CyEqSet` with 148 equivalent Cypher query pairs from the above two approaches. `CyEqSet` is representative since it covers various of Cypher query fragments: `MATCH`, `OPTIONAL MATCH`, `RETURN`, `WHERE`, `WITH`, `UNION`, `UNION ALL`, `UNWIND`, `LIMIT`, `SKIP`, `ORDER BY`, `ASC`, `DESC`, and aggregate functions (`SUM`, `COUNT`, `AVG`, `DISTINCT`, `MIN`, `MAX`, `COLLECT`). `CyEqSet` also covers advanced or complex Cypher features, including all the features in Table I along with special value (e.g., null, false) and queries that always return empty results.

To test the effectiveness of `GraphQE` on proving non-equivalent Cypher queries, we construct a dataset `CyNeqSet` containing 148 non-equivalent Cypher query pairs. Specifically, we mutate the equivalent Cypher query pairs in `CyEqSet` by randomly applying one of the following mutation rules: 1) changing the direction of a path, 2) changing the property values or labels of some variables, 3) changing `UNION ALL` to `UNION` or vice versa, 4) changing the value of `LIMIT` or `ORDER BY` and 5) removing or adding `DISTINCT`. We further manually confirmed that each pair of Cypher queries in `CyEqSet` is not equivalent.

B. Verification Result

Table III shows the evaluation result of `GraphQE` on `CyEqSet`. Specifically, out of 148 pairs of equivalent Cypher queries in `CyEqSet`, `GraphQE` can successfully prove the equivalence of 138 query pairs. On the dataset translated from Calcite, we proved 73 out of 80 (89%) equivalent Cypher query pairs. On the dataset constructed from real-world projects, we proved 65 out of 68 (approximately 96%) equivalent Cypher query pairs. We further analyze the failed cases and identify the reasons for these failed cases.

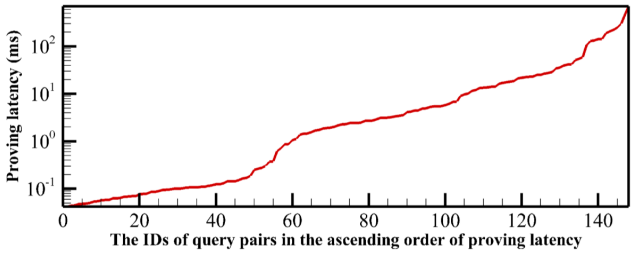

Fig. 5. The proving latency of `GraphQE`.

- **Sorting and truncation.** `GraphQE` cannot prove equivalent queries that contain inconsistent numbers of `ORDER BY...LIMIT...SKIP...` fragments within subqueries due to the limitation of our divide-and-conquer proving approach in Section IV-B. 2 cases failed due to this reason.
- **Aggregate.** `GraphQE` cannot eliminate the intermediate variables in nested aggregates and aggregate computations, e.g., `COUNT(SUM(n))` and `SUM(n)/COUNT(n)`. 4 cases failed due to this reason.
- **Uninterpreted function.** `GraphQE` utilizes uninterpreted functions to model `COLLECT` aggregate and built-in functions, which cannot express their concrete semantics. 4 cases failed due to this reason.

`GraphQE` has successfully proved 138 out of 148 pairs of equivalent Cypher queries, demonstrating its effectiveness.

We evaluate `GraphQE` on `CyNeqSet`, and check whether `GraphQE` can prove the non-equivalence of these pairs in `CyNeqSet`. The experimental results show that `GraphQE` proves all test cases in `CyNeqSet` to be non-equivalent.

C. Performance

To evaluate the performance of `GraphQE`, we conduct tests about `GraphQE`'s proving latency on a platform equipped with an Intel Core i5-11300 processor and 16GB of RAM.

The average latency for proving Cypher query equivalence across all test cases in `CyEqSet` is 38ms. Fig. 5 shows the distribution of proving latency for `GraphQE`. 90% of the test cases are proved by `GraphQE` in less than 100ms. Notably, 75 out of 148 Cypher query pairs are verified with an exceptionally low latency of 10ms. Only 2 Cypher query pairs require over 500ms latency for proving.

The Cypher query pair with the highest proving latency is due to the `ORDER BY...LIMIT...` fragments within subqueries. We use the divide-and-conquer based approach that proves their equivalence by dividing the Cypher queries into subqueries by `ORDER BY...LIMIT...` fragments. Each

subquery is then individually proved for equivalence, leading to significantly higher proving latency than other cases.

GraphQE can efficiently prove the equivalence of Cypher queries with a latency of 38ms on average.

VIII. Discussion

In this section, we first discuss the limitations of GraphQE. Then, we discuss how to extend GraphQE for supporting other graph query languages, e.g., GQL [21], [32], Gremlin [7] and SPARQL [41].

A. Limitations

Although GraphQE is effective in proving the equivalence of Cypher queries, GraphQE is unable to adequately support some Cypher features, which necessitate the development of new approaches in the future.

- GraphQE cannot prove the equivalence of Cypher queries containing nested aggregates and aggregation computations, e.g., `SUM(SUM(n))` and `SUM(n)/COUNT(n)`, because their corresponding *G-expression*s contain intermediate variables that cannot be eliminated through our approach.
- GraphQE cannot prove the equivalence of Cypher queries containing different number of `ORDER BY...LIMIT...SKIP...` fragments, because our divide-and-conquer proving process requires that two equivalent Cypher queries have the same number of `ORDER BY...LIMIT...SKIP...` fragments.
- GraphQE cannot model the concrete semantics of built-in functions and user-defined functions, and cannot support some Cypher features, e.g., `YIELD` in `CALL`.

B. Extending GraphQE to other Graph Query Languages

Although GraphQE is designed for Cypher queries, it can be extended to other graph query languages, e.g., GQL [21], [32], Gremlin [7] and SPARQL [41]. Next, we introduce how to extend GraphQE to support these graph query languages.

Supporting GQL. GQL is a new standard graph query language that extends Cypher and is fully compatible with Cypher. We can easily extend GraphQE to support GQL from the following aspects. First, GraphQE needs to support the property graph model adopted by GQL. GraphQE needs to remove its relationship-injective semantics when constructing *G-expression*s for GQL. GraphQE needs to allow a relationship to be specified with more than one label. Second, GraphQE can support undirected relationships using uninterpreted function. Third, GraphQE needs to model the new introduced features in GQL, e.g., `EXCEPT`, `FILTER` and `FOR`.

Supporting Gremlin. Gremlin is designed as a functional query language on property graphs. For example, the Gremlin query $g.V().hasLabel(\text{“}person\text{”}).has(\text{“}name\text{”}, \text{“}Alice\text{”}).out(\text{“}knows\text{”}).values(\text{“}name\text{”})$ finds all people that Alice knows. The semantic difference between Gremlin and Cypher is minimal. Gremlin does not use relationship-injective semantics and allows a relationship to have more than one label. GraphQE needs to remove its relationship-injective semantics while constructing *G-expression*s and allows a relationship to be specified with more than one label. Then, GraphQE needs to model the Gremlin fragments as *G-expression*s.

Supporting SPARQL. SPARQL is a graph query language designed for RDF graphs [51]. SPARQL defines graph patterns as triples $\langle Subject, Predicate, Object\rangle$ that match paths from $Subject$ to $Object$ through $Predicate$. We can model SPARQL queries on an RDF graph in the same way of modeling Cypher queries, since an RDF graph can be treated as a special kind of property graphs [11]. GraphQE needs to fix the gap between the RDF graph model and property graph model, and build *G-expression*s for SPARQL fragments.

IX. Related Work

SQL query equivalence provers. Researchers have proposed syntax-based [16], [17], [52], [53] and semantics-based [22], [48], [49] SQL equivalence provers. Syntax-based approaches prove the equivalence of SQL queries by checking the isomorphism of SQL algebraic representations. For example, SPES [53] proposes tree-based SQL algebraic representations to model SQL relaional algebras, and adopts Z3 SMT solver to prove their equivalence. However, these approaches cannot handle equivalent SQL queries that have significantly different syntactic structures. Semantics-based approaches model SQL queries using semiring expressions, and transform them into first-order logic expressions that can be verified by SMT solvers. However, these approaches cannot be directly used to prove Cypher query equivalence.

Equivalent graph queries and their applications. Equivalent rewriting of graph queries is essential in both bug detection of GDBs [33], [34], [54] and query optimization [15], [30]. Kamm et al. [34] test Gremlin-supported GDBs via constructing equivalent Gremlin queries as test oracles through equivalent query rewriting. Jiang et al. [33] use equivalent rewriting to test Cypher-supported GDBs. However, these rewriting rules are manually defined. Our work aims to propose an automated graph query equivalence prover.

X. Conclusion

Query equivalence proving is a fundamental problem in database research. However, we still lack a graph query equivalence prover for the emerging graph databases. In this paper, we propose GraphQE, the first Cypher query equivalence prover, to determine whether two Cypher queries are semantically equivalent. We model Cypher queries as *G-expression*s, and then prove their equivalence by using constraint solvers. We further construct a dataset that contains 148 pairs of equivalent Cypher queries. We evaluate GraphQE on this dataset, and GraphQE has proved 138 pairs of equivalent Cypher queries, demonstrating the effectiveness of GraphQE.

Acknowledgments

This work was partially supported by National Natural Science Foundation of China (62072444), Major Project of ISCAS (ISCAS-ZD-202302), Basic Research Project of ISCAS (ISCAS-JCZD-202403), and Youth Innovation Promotion Association at Chinese Academy of Sciences (Y2022044).

REFERENCES

[1] "Azure cosmos db." [Online]. Available: https://azure.microsoft.com/en-us/services/cosmos-db/

[2] "Opencypher documentation." [Online]. Available: https://opencypher.org/

[3] "Janusgraph: Official documentation," 2019. [Online]. Available: https://docs.janusgraph.org

[4] "Calcite test suite," 2021. [Online]. Available: "https://github.com/georgia-tech-db/Qizhou2020spes/blob/main/testData"

[5] "ArangoDB: The native multi-model database," "https://www.arangodb.com/", 2023.

[6] "Graph database ranking," "https://db-engines.com/en/ranking/graph+dbms", 2023.

[7] "Gremlin: The graph traversal machine and language," https://tinkerpop.apache.org/gremlin.html, 2023.

[8] "Cypher for Gremlin," https://github.com/opencypher/cypher-for-gremlin, 2024.

[9] "Memgraph documentation," 2024. [Online]. Available: https://memgraph.com/docs/memgraph/introduction

[10] "openCypherTranspiler: An open source project for transpiling cypher queries," https://github.com/microsoft/openCypherTranspiler, 2024.

[11] R. Angles, M. Arenas, P. Barceló, A. Hogan, J. Reutter, and D. Vrgoč, "Foundations of modern query languages for graph databases," *ACM Computing Surveys (CSUR)*, vol. 50, no. 5, pp. 1–40, 2017.

[12] P. Barceló Baeza, "Querying graph databases," in *Proceedings of ACM SIGMOD-SIGACT-SIGAI Symposium On Principles of Database Systems (PODS)*, 2013, pp. 175–188.

[13] M. Brandizi, A. Singh, and K. Hassani-Pak, "Getting the best of linked data and property graphs: Rdf2neo and the KnetMiner use case," in *Proceedings of International Conference Semantic Web Applications and Tools for Life Sciences (SWAT4LS)*, 2018.

[14] A. K. Chandra and P. M. Merlin, "Optimal implementation of conjunctive queries in relational databases," in *Proceedings of the ACM Symposium on Theory of Computing (STOC)*, 1977, pp. 77–90.

[15] S. Choudhury, L. Holder, G. Chin, P. Mackey, K. Agarwal, and J. Feo, "Query optimization for dynamic graphs," *arXiv preprint arXiv:1407.3745*, 2014.

[16] S. Chu, B. Murphy, J. Roesch, A. Cheung, and D. Suciu, "Axiomatic foundations and algorithms for deciding semantic equivalences of SQL queries," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 11, no. 11, pp. 1482–1495, 2018.

[17] S. Chu, K. Weitz, A. Cheung, and D. Suciu, "Hottsql: Proving query rewrites with univalent sql semantics," vol. 52, no. 6, 2017, pp. 510–524.

[18] S. Cohen, W. Nutt, and A. Serebrenik, "Rewriting aggregate queries using views," in *Proceedings of the ACM Symposium on Principles of Database Systems (SIGMOD-SIGACT-SIGART)*, 1999, pp. 155–166.

[19] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *Proceedings of International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*, 2008, pp. 337–340.

[20] R. De Virgilio, "Smart rdf data storage in graph databases," in *IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. IEEE, 2017, pp. 872–881.

[21] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels, F. Murlak, S. Plantikow, P. Selmer, O. van Rest, H. Voigt, D. Vrgoč, M. Wu, and F. Zemke, "Graph pattern matching in gql and SQL/PGQ," in *Proceedings of the ACM International Conference on Management of Data (SIGMOD)*, 2022, pp. 2246–2258.

[22] H. Ding, Z. Wang, Y. Yang, D. Zhang, Z. Xu, H. Chen, R. Piskac, and J. Li, "Proving query equivalence using linear integer arithmetic," *Proceedings of the ACM on Management of Data (SIGMOD)*, vol. 1, no. 4, pp. 1–26, 2023.

[23] B. A. Eckman and P. G. Brown, "Graph data management for molecular and cell biology," *IBM Journal of Research and Development*, vol. 50, no. 6, pp. 545–560, 2006.

[24] O. Erling, A. Averbuch, J. Larriba-Pey, H. Chafi, A. Gubichev, A. Prat, M.-D. Pham, and P. Boncz, "The LDBC social network benchmark: Interactive workload," in *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2015, pp. 619–630.

[25] W. Fan, Y. M. 0001, Q. Li, Y. He, Y. E. Zhao, J. Tang, and D. Yin, "Graph neural networks for social recommendation," in *Proceedings of the World Wide Web Conference (WWW)*, 2019, pp. 417–426.

[26] F. Färber, S. K. Cha, J. Primsch, C. Bornhövd, S. Sigg, and W. Lehner, "Sap hana database: Data management for modern business applications," *ACM Sigmod Record*, vol. 40, no. 4, pp. 45–51, 2012.

[27] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, "Cypher: An evolving query language for property graphs," in *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2018, pp. 1433–1445.

[28] R. A. Ganski and H. K. Wong, "Optimization of nsted sql queries revisited," *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, vol. 16, no. 3, pp. 23–33, 1987.

[29] T. J. Green, G. Karvounarakis, and V. Tannen, "Provenance semirings," in *Proceedings of ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, 2007, pp. 31–40.

[30] J. Hölsch and M. Grossniklaus, "An algebra and equivalences to transform graph patterns in Neo4j," in *Proceedings of the Workshops of the EDBT/ICDT Joint Conference (EDBT/ICDT)*, 2016.

[31] IMARC Group, "Graph database market size, share, growth report 2024-32," www.imarcgroup.com, 2024.

[32] *Information technology — Database languages — GQL*, International Organization for Standardization (ISO) Std. ISO/IEC FDIS 39 075:2024, 2024.

[33] Y. Jiang, J. Liu, J. Ba, R. H. C. Yap, Z. Liang, and M. Rigger, "Detecting logic bugs in graph database management systems via injective and surjective graph query transformation," in *Proceedings of IEEE/ACM International Conference on Software Engineering (ICSE)*, 2023, pp. 531–542.

[34] M. Kamm, M. Rigger, C. Zhang, and Z. Su, "Testing graph database engines via query partitioning," in *Proceedings of ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, 2023, pp. 140–149.

[35] C. Kemper, *Beginning Neo4j*. Springer, 2015.

[36] M. Kim and J. Leskovec, "Multiplicative attribute graph model of real-world networks," *Internet Mathematics*, vol. 8, no. 1-2, pp. 113–160, 2012.

[37] M. Levatich, N. Bjørner, R. Piskac, and S. Shoham, "Solving using approximations," in *International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2020, pp. 360–378.

[38] J. Marton, G. Szárnyas, and D. Varró, "Formalising opencypher graph queries in relational algebra," in *Proceedings of the Advances in Databases and Information Systems (ADBIS)*, 2017, pp. 182–196.

[39] S. A. T. Mpinda, L. C. Ferreira, M. X. Ribeiro, and M. T. P. Santos, "Evaluation of graph databases performance through indexing techniques," *International Journal of Artificial Intelligence & Applications (IJAIA)*, vol. 6, no. 5, pp. 87–98, 2015.

[40] T. Parr, "The definitive ANTLR4 reference," *The Definitive ANTLR4 Reference*, pp. 1–326, 2013.

[41] J. Pérez, M. Arenas, and C. Gutierrez, "Semantics and complexity of SPARQL," *ACM Transactions on Database Systems (TODS)*, vol. 34, no. 3, pp. 1–45, 2009.

[42] Y. Ren, H. Zhu, J. Zhang, P. Dai, and L. Bo, "EnsemFDet: An ensemble approach to fraud detection based on bipartite graph," in *Proceedings of International Conference on Data Engineering (ICDE)*, 2021, pp. 2039–2044.

[43] D. Ritter, L. Dell'Aquila, A. Lomakin, and E. Tagliaferri, "Orientdb: A nosql, open source mmdms." in *Proceedings of the British International Conference on Databases (BICOD)*, 2021, pp. 10–19.

[44] Y. Sagiv and M. Yannakakis, "Equivalences among relational expressions with the union and difference operators," *Journal of the ACM (JACM)*, vol. 27, no. 4, pp. 633–655, 1980.

[45] B. A. Steer, A. Alnaimi, M. A. Lotz, F. Cuadrado, L. M. Vaquero, and J. Varvenne, "Cytosm: Declarative property graph queries without data migration," in *Proceedings of International Workshop on Graph Data-management Experiences and Systems (GRADES)*, 2017, pp. 1–6.

[46] J. R. Ullmann, "An algorithm for subgraph isomorphism," *Journal of the ACM (JACM)*, vol. 23, no. 1, pp. 31–42, 1976.

[47] J. Wang, K. Wang, J. Li, J. Jiang, Y. Wang, J. Mei, and S. Li, "Accelerating epidemiological investigation analysis by using nlp and knowledge reasoning: A case study on covid-19," in *American Medical Informatics Association Annual Symposium (AMIA)*, 2020.

[48] S. Wang, S. Pan, and A. Cheung, "Qed: A powerful query equivalence decider for sql," in *Proceedings of International Conference on Very Large Data Bases (PVLDB)*, 2024, pp. 3602–3614.

[49] Z. Wang, Z. Zhou, Y. Yang, H. Ding, G. Hu, D. Ding, C. Tang, H. Chen, and J. Li, "WeTune: Automatic discovery and verification of query rewrite rules," in *Proceedings of ACM International Conference on Management of Data (SIGMOD)*, 2022, pp. 94–107.

[50] M. Wu, X. Yi, H. Yu, Y. Liu, and Y. Wang, "Nebula graph: An open source distributed graph database," *CoRR*, vol. abs/2206.07278, 2022.

[51] M. Wylot, M. Hauswirth, P. Cudré-Mauroux, and S. Sakr, "Rdf data storage and query processing schemes: A survey," *ACM Computing Surveys*, vol. 51, no. 4, 2018.

[52] Q. Zhou, J. Arulraj, S. Navathe, W. Harris, and D. Xu, "Automated verification of query equivalence using satisfiability modulo theories," vol. 12, no. 11, 2019, pp. 1276–1288.

[53] Q. Zhou, J. Arulraj, S. B. Navathe, W. Harris, and J. Wu, "Spes: A symbolic approach to proving query equivalence under bag semantics," in *IEEE International Conference on Data Engineering (ICDE)*, 2022, pp. 2735–2748.

[54] Z. Zhuang, P. Li, P. Ma, W. Meng, and S. Wang, "Testing graph database systems via graph-aware metamorphic relations," *Proceedings of the VLDB Endowment (PVLDB)*, vol. 17, no. 4, pp. 836–848, 2023.