A Graph Query Language based on the Monadic Second-Order Logic (Full Version)

Yunkai Lou[1], Chaokun Wang[1], Junchao Zhu[1], Hao Feng[1]
[1] Tsinghua University, Beijing 100084, China
{louyk18,zhu-jc17,fh20}@mails.tsinghua.edu.cn,chaokun@tsinghua.edu.cn

ABSTRACT

With the wide application of graphs in various fields, graph query languages have attracted more and more attention. Existing graph query languages, such as GraphQL and SoQL, mostly have similar expressive power as the first-order logic or its extended versions, and are limited when used to express various queries. In this paper, we propose a more expressive declarative graph query language named *SOGQL* to support more common queries efficiently. First, a new graph calculus for attributed graphs is proposed based on monadic second-order logic. Then, we propose a new graph data model. Specifically, a new graph algebra, which operates on graph sets, is designed, and it has seven fundamental operators such as union, filter, map, and reduce. Next, a graph query language *SOGQL* is proposed, and its grammar is carefully designed based on our graph data model. Besides, it is proved that our graph calculus and graph algebra have the same expressive power. Therefore, *SOGQL* is more expressive than existing graph query languages thanks to the power of the monadic second-order logic, and can express more common and useful queries such as maximal clique searching. Moreover, a prototype system *SOGDB* applied with *SOGQL* is implemented and the experimental results show its efficiency.

1 MOTIVATION

As graphs have been widely used in many application domains such as semantic web, finance [41], medical data [5], and road networks [18], various graph query languages have been proposed [10, 17, 23, 24, 27, 39] to extract useful information from the graphs conveniently. One of the major differences among the existing graph query languages is the basic unit. For example, GraphDB [24] and G-Log [37] are for object-oriented models, SoQL [39] and PQL [33] focus on paths, and a query language for hypernodes is proposed in [34]. However, it is complicated for these graph query languages to express some common queries such as subgraph matching [26]. Specifically, the subgraph matching query uses a pattern graph and a set of data graphs as the inputs, and the query results are a set of graphs that are the subgraphs of the data graphs and isomorphic to the pattern graph. Since the input and output of such queries are both sets of graphs, in the statements of the graph query language, the related set of graphs should be specified, and the query results are also stored in a graph set. Note that a graph has good capability in describing different structures, including simple structures such as paths and complex graph structures such as cliques. Therefore, a graph is used as the basic unit of our graph data model and graph query language, which makes it more convenient and direct to write query statements with our language.

Moreover, to the best of our knowledge, existing graph query languages are usually based on the first-order logic or its extended versions, and have limited improvement in expressive power compared with relational algebra [11, 27, 34]. As the first-order logic and its extended versions do not allow quantifiers on the set of elements, it is hard for these languages to express queries with constraints on several graphs, because a graph is actually a set of vertices and edges. For instance, it is difficult for the languages to query all the maximal cliques in a given graph, since the obtained cliques are required not to be contained by any other clique. Hence, existing graph query languages have limited ability in performing complex queries on graphs. Next, the limitation of the existing graph query languages is demonstrated with the query of maximal clique searching in more detail.

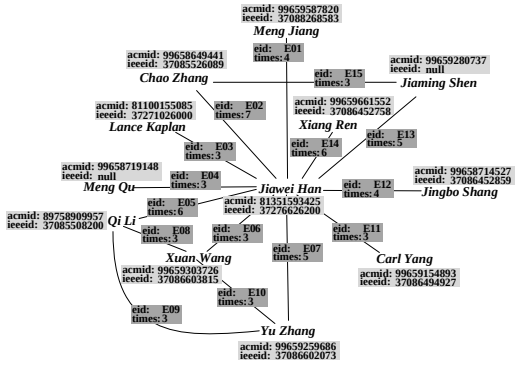
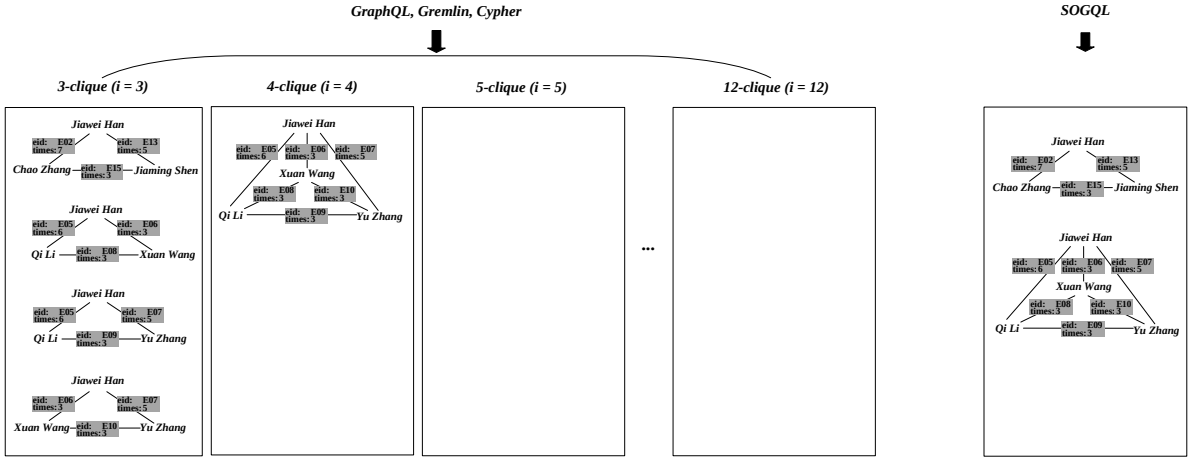


Figure 1: A snapshot of the coauthorship network consisting of the researchers coauthored with *Jiawei Han* in 2018. An edge between two researchers represents these two authors have coauthored more than two papers in 2018.

Example 1.1. A subgraph $G_c = (V_c, E_c)$ of a graph $G = (V, E)$ is called a maximal clique (abbr. *mc*) [35] iff (1) $|V_c| \geq 3$; (2) there are edges between any two vertices in G_c; (3) there is no vertex set $V_c' \supset V_c$ satisfying that the induced subgraph of V_c' on G is also a clique. Finding *mc*s is a well-known and useful query on graphs, and it has wide applications such as communitiy detection in social networks [36] and 3-D protein structure alignment [43]. However, to the best of our knowledge, most existing widely used graph query languages [4, 47], including GraphQL [27], Gremlin [38], and Cypher [21], do not have the expressive power of monadic second-order logic, and are not expressive enough to query *mc*s directly.

Given a coauthorship graph shown in Fig. 1, all the *mc*s containing *Jiawei Han* in this graph are wanted. For the existing graph query languages, an intuitive way to get the *mc*s is to search for cliques with i vertices ($3 \leq i \leq |V|$) separately, and then to pick the *mc*s from the obtained cliques.



(a) Results of GraphQL, Gremlin, and Cypher. **(b) Results of *SOGQL*.**

Figure 2: The results of searching for *mc*s with the existing graph query languages (GraphQL, Gremlin, Cypher) and our graph query language *SOGQL*. Please note that the attributes of vertices are omitted here for clarity.

In terms of GraphQL, the query for finding a clique with i vertices is as follows:

graph $P\{$
  node v_1 < Researcher name="Jiawei Han" >;
  node $v_2; \cdots;$ **node** v_i;
  edge $(v_1, v_2); \cdots;$ **edge** $(v_1, v_i); \cdots;$ **edge** (v_{i-1}, v_i);
$\}$;
for P **exhaustive in** G_c
return P

where G_c is a collection of graphs.

In terms of Gremlin, the query for a clique with i vertices is shown as follows:

G.V().match(
  __.as("v_1**").hasLabel("Researcher").has("name", "Jiawei Han"),**
  __.as("v_1**").both().as("**v_2**"),**
  $\cdots$**,**
  __.as("v_1**").both().as("**v_i**"),**
  __.as("v_2**").both().as("**v_3**"),**
  $\cdots$**,**
  __.as("v_2**").both().as("**v_i**"),**
  $\cdots$**,**
  __.as("v_{i-1}**").both().as("**v_i**"))**
  .select("v_1**",** $\cdots$**, "**v_i**")**

In terms of Cypher, the query statement is as follows:

MATCH $(v_1$**:Researcher {name: "Jiawei Han"})** -- (v_2),
  (v_1) -- $(v_3), \cdots, (v_1)$ -- (v_i),
  (v_2) -- $(v_3), \cdots, (v_2)$ -- (v_i),
  $\cdots$,
  (v_{i-1}) -- (v_i)
WHERE v_1 <> $v_2, \cdots, v_1$ <> v_i,
  v_2 <> $v_3, \cdots, v_2$ <> v_i,
  $\cdots$,
  v_{i-1} <> v_i
RETURN $v_1, \cdots, v_i$

As shown in Fig. 2a, all the cliques containing *Jiawei Han* in Fig. 1 are returned after setting i from 3 to $|V| = 12$. However, the existing graph query languages are not expressive enough to find the maximal ones among these obtained cliques. Moreover, when there are numerous vertices in G, such methods are time-consuming and impractical because a lot of possible values of i should be tried. For example, given a graph G with 100,000 vertices, suppose an *mc* in G contains 200 vertices. Then, for the existing graph query languages, i need be set from 3 to $|V| = 200$, and 198 queries are performed.

In order to perform *mc* searching that cannot be expressed in Cypher natively, Neo4j proposes a user defined procedure *apoc.algo.cliquesWithNode(startNode, minSize)* to obtain the *mc*s with at least n vertices, but it is beyond the native expressive power of Cypher. □

Therefore, this study proposes a more expressive graph query languages named *SOGQL* (monadic-**S**econd-**O**rder-logic-based **G**raph **Q**uery **L**anguage) to support the complex queries on graphs. Specifically, a graph calculus is proposed based on the monadic second-order logic, and a new graph data model is presented. Specifically, a graph algebra is proposed and proved to have the same expressive power as our graph calculus. The seven fundamental operators and four useful additional operators of the graph algebra are presented in Table 1. Then, the grammar of our declarative graph query language is carefully designed based on our graph data model. As shown in Fig. 2b, all the *mc*s containing *Jiawei Han* in Fig. 1 can be obtained with the following *SOGQL* statement.

```
select maximal * from GS
    where Clique() and Exist x(Vertex(x) and x.name = "Jiawei Han");
```

Specifically, *Clique* and *Vertex* are formulae of the first-order logic, and their meanings can be found in Table 2.

In summary, the main contributions of this paper include:

(1) This paper brings forward a new graph calculus based on the monadic second-order logic (in Section 2). Compared with relational calculus, the graph calculus is more powerful, and it can express the relationship of "belonging to" between an item and a set.

(2) A new graph data model is proposed (in Section 3). Specifically, its basic unit is a graph, and a new graph algebra is presented (in Section 4). Seven fundamental operators are carefully designed, and they are effective for expressing various queries on graphs.

Table 1: Fundamental Operators of Our Graph Algebra and Some Additional Operators. The column *"Unary/Binary"* means the number of the processed graph sets when the operator is applied, and the column *"New Schema"* indicates whether the schema of the obtained graph set is different from any of the processed graph sets.

Operator	Sign	Description	Unary /Binary	New Schema
Fundamental Operators				
Projection	ρ	choose vertices and edges with necessary labels, and choose their attributes with the given new schema	Unary	✓
Selection	σ	obtain the subgraphs of the graphs in a graph set, and the subgraphs should satisfy the given constraints	Unary	×
Reduce	γ	obtain a new graph by putting all the graphs in a given graph set together	Unary	×
Cartesian Product	$\times$	suppose $\mathcal{G}_1$ contains n graphs $G_{11}, \cdots, G_{1n}$, and $\mathcal{G}_2$ contains m graphs $G_{21}, \cdots, G_{2m}$, $\mathcal{G}_1 \times \mathcal{G}_2$ contains mn graphs $G_{11} \cup_g G_{21}, \cdots, G_{11} \cup_g G_{2m}, \cdots, G_{1n} \cup_g G_{2m}$, where $\cup_g$ means to unite two graphs	Binary	✓
Align	ζ	align every two vertices (edges) that represent the same entity (relationship)	Unary	✓
Difference	$-$	set difference between two graphs sets	Binary	×
Map	μ	deal with the graphs in a graph set respectively	Unary	×
Additional Operators				
Join	$\bowtie$	given two graph sets $\mathcal{G}_1 = (\widehat{G}_1, \mathcal{S})$ and $\mathcal{G}_2 = (\widehat{G}_2, \mathcal{S})$, $\mathcal{G}_1 \bowtie \mathcal{G}_2 = \zeta_{h_1}(\cdots(\zeta_{h_m}(\mathcal{G}_1 \times \mathcal{G}_2))\cdots)$, after uniting a graph in $\mathcal{G}_1$ and a graph in $\mathcal{G}_2$, a vertex labeled 1.x and a vertex labeled 2.x are aligned, where $x \in \mathcal{S}.\mathcal{T}$, iff they have the same values in *all* attributes. Besides, two edges labeled 1.x and 2.x respectively are aligned iff they have the same attribute values and the same source and target vertices.	Binary	×
Intersection	$\cap$	set intersection, $\mathcal{G}_1 \cap \mathcal{G}_2 = \mathcal{G}_1 - (\mathcal{G}_1 - \mathcal{G}_2)$	Binary	×
Maximal	$\mathcal{M}$	choose the maximal graphs in a graph set $\mathcal{M}(\mathcal{G}) = \mu(\mathcal{G} \| \|\mathcal{G}' : \gamma((\mathcal{G}' \bowtie \mathcal{G}) \cap \mathcal{G}) \cap \mathcal{G}')$	Unary	×
Union	$\cup$	given two graph sets $\mathcal{G}_1 = (\widehat{G}_1, \mathcal{S})$ and $\mathcal{G}_2 = (\widehat{G}_2, \mathcal{S})$ with the same schemas, put the graphs in them into a new graph set $\mathcal{G}_1 \cup \mathcal{G}_2 = (\widehat{G} = \{G \| G \in \mathcal{G}_1 \vee G \in \mathcal{G}_2\}, \mathcal{S})$	Binary	×

(3) A new declarative graph query language named *SOGQL* is proposed based on the graph data model (in Section 5). *SOGQL* can express complex queries such as searching for all the cliques in a graph.

(4) Our proposed graph calculus is proved to have the same expressive power as our graph algebra (in Section 6). It indicates that *SOGQL* has the power of the monadic second-order logic.

(5) A prototype system *SOGDB* applied with *SOGQL* is implemented, and the results of the efficiency experiments show that *SOGDB* is more efficient than the widely-used graph database system Neo4j (in Section 7).

2 GRAPH CALCULUS

In this section, the preliminaries of our graph calculus, such as the monadic second-order logic, are briefly presented. Then, the graph calculus based on the monadic second-order logic is proposed.

2.1 Preliminaries

Zeroth-order logic (or propositional logic) refers to the formulae that consist of atomic propositions and logical connectives such as logical *not* ($\neg$), *and* ($\wedge$),*or* ($\vee$), *imply* ($\rightarrow$), as well as a set of formal "rules of proof" such as the syllogism [32]. For example, given two propositions F : "Peter is 30 years old" and M : "Peter is an adult", $F \rightarrow M$ is a formula in propositional logic.

First-order logic (abbr. FOL) introduces quantifiers and assertions into the formula, and propositions are described with variables [8]. For example, given two assertions F' : "is 30 years old" and M' : "is an adult", $\forall x(F'(x) \rightarrow M'(x))$ is a formula in FOL. Here x is an element in the domain of all human beings. In addition, the relational calculus, which is able to express queries on the relational data model [14], can be viewed as a specialization of FOL [22].

Different from commonly used zeroth-order logic and FOL, **monadic second-order logic (abbr. MSOL)** is a fragment of second-order logic, and it allows quantifiers on the set of elements [28]. That is, sets of elements can also be used as variables. For example, let S be the set of adults and x be a human. Then, $\forall S \forall x(x \in S \vee \neg(x \in S))$ is a formula in MSOL. The formula means that a human is either an adult or not an adult.

Note that MSOL is more capable and convenient for describing structures and properties in graphs [16]. For example, on an undirected graph G, formula $\exists w, x, y, z(HasEdge(G, w, x) \wedge HasEdge(G, x, y) \wedge HasEdge(G, y, z) \wedge Unique(w, x, y, z) \wedge \exists S(Subgraph(S, G) \wedge \neg(x \in S) \wedge \neg(y \in S) \wedge w \in S \wedge z \in S \wedge Connect(S)))$ means that G contains a circle with at least four vertices. In detail, w, x, y, z represent vertices, $HasEdge(G, x, y)$ represents that there is an edge between vertices x and y in G, $Unique(w, x, y, z)$ means the vertices w, x, y, z are different, *Subgraph(S, G)* means that S is a subgraph of G, and $Connect(S)$ means that graph S is connected (i.e., any two vertices, saying w and z, in S are reachable from each other). The expression is quite accurate although it looks a bit complicated. Note that such graphs are not expressible with FOL according to the Compactness Theorem [31].

In the present work, the description of a graph model often requires a set of vertices or edges. Thus, MSOL has benefits in avoiding the lengthy and complex modeling, which is more convenient than FOL.

2.2 Graph Calculus Based on MSOL

A graph is a set of vertices and edges, and many complex predicates can be specified on graphs, such as "a graph that is a connected component". However, FOL [8] is not expressive enough to make such predicates [16]. Therefore, our element graph calculus is proposed based on MSOL. The element graph calculus is called the graph calculus in this paper for simplicity.

In our graph calculus, elements are used to represent vertices and edges. For an element that represents a vertex, it contains the label and attributes of the vertex. For an element representing an edge, it contains not only the label and attributes, but also the source and target vertices of the edge. In the graph calculus, a graph is a set of elements that represent the vertices and edges in the graph, and a graph set is a set of graphs. Please note that the elements in a graph

can have different schemas. In an expression of our graph calculus, there can be three kinds of variables, i.e., the vertex variables, edge variables, and graph variables. These variables represent vertices, edges and graphs respectively. Usually, both vertex variables and edge variables are called element variables. Suppose t is an element variable, then, $t[1]$ represents the label of that element. In particular, if t is an edge variable, then, $t[2]$ and $t[3]$ are two elements that represent the source vertex and target vertex of this edge, respectively. For clarity, $t[label]$ is used to represent $t[1]$ for element variables, and $t[src]$ and $t[tgt]$ are used to represent $t[2]$ and $t[3]$ only for edge variables. The other parts of an element variable are the attribute values of this element. Moreover, for an undirected graph, each undirected edge is recorded as two opposite directed edges.

A graph calculus expression is expressed as: $\{G|\Phi(G)\}$ where G is a graph variable and Φ is a formula. This expression is used to obtain a new set of graphs that satisfy the constraints specified by Φ. Note that when an edge e is included in a resultant graph, e's adjacent vertices $e[src]$ and $e[tgt]$ are also included in that graph. Besides, there can be several vertex variables, edge variables and graph variables in formula Φ. A variable is said to be free unless it is quantified by $\exists$ or $\forall$.

A graph calculus formula (e.g., $\Phi(G)$) is built up out of atoms. An atom has one of the following forms:

(1) $T(p)$, where p is an element variable, and T is the type of p. T can be *Vertex* or *Edge*. $Vertex(p)$ and $Edge(p)$ mean that p is a vertex and an edge, respectively.
(2) $p[i]\theta q[j]$, where p and q are element variables, and θ is a comparison operator, including $<, \leq, =, \neq, \geq, >$. $p[i]\theta q[j]$ means that the i-th component of p and the j-th component of q satisfy the restriction of θ.
(3) $p[i]\theta c$ or $c\theta p[i]$, where p is an element variable, and c is a constant.
(4) $p \in G$, $p_c \in G$, or $p \in G_c$, where p is an element variable, p_c is a constant element that represents a stable vertex or a stable edge, G is a graph variable, and G_c is a constant graph. It means that the element corresponding to p or p_c is in the graph corresponding to G or G_c.

The first three formulae are similar to those in relational calculus [44], because it is still necessary to restrict vertices and edges on their types and attributes in the graph calculus. The last formula is used to describe the relationships between an element and a set of elements, which is never considered in relational calculus.

We build up formulae from atoms with the following rules:

(1) An atom is a formula.
(2) If Φ is a formula, then so are $\neg\Phi$ and (Φ).
(3) If Φ_1 and Φ_2 are formulae, then so are $\Phi_1 \vee \Phi_2$, $\Phi_1 \wedge \Phi_2$ and $\Phi_1 \rightarrow \Phi_2$.
(4) If $\Phi(x)$ (or $\Phi(X)$, where X is a graph variable) is a formula containing a free variable x or X, then $\exists x(\Phi(x))$ (or $\exists X(\Phi(X))$) and $\forall x(\Phi(x))$ (or $\forall X(\Phi(X))$) are also formulae.

For the sake of simplicity, formulae with clear semantics can be expressed with notations. Some commonly used notations are shown in Table 2. For instance, we explain the notation Connect(G) in detail. Connect(G) reflects that graph G is connected, and the full formula is as follows:

$$Connect(G) = \neg\exists X(InducedSub(X,G) \wedge \underbrace{①\ \exists u(Vertex(u) \wedge u \in X)}$$
$$\wedge \underbrace{②\ \exists u(Vertex(u) \wedge u \in G \wedge \neg(u \in X))} \wedge \underbrace{③\ \forall x(\forall y((Vertex(x)}$$
$$\underbrace{\wedge x \in G \wedge Vertex(y) \wedge y \in G \wedge HasEdge(G,x,y)) \rightarrow}$$
$$\underbrace{((x \in X \rightarrow y \in X) \wedge (y \in X \rightarrow x \in X)))))}.$$

In detail, for a connected graph G, there does not exist any induced subgraph X of G satisfying ① X is not empty; ② at least one vertex in G is not in X; ③ if a vertex is in X, then all its neighbors are also in X. This notation and formula cannot be utilized in the FOL expressions, because X is a variable representing a set.

Example 2.1. $\mathcal{G}_0$ is a graph set with k graphs $\{G_1, \cdots, G_k\}$. To get all the connected components of the graphs in $\mathcal{G}_0$, the expression can be written as: $\{G|\exists G'(GraphInSet(G', \mathcal{G}_0) \wedge InducedSub(G, G') \wedge \forall x, y((Vertex(x,y) \wedge x \in G' \wedge y \in G' \wedge HasEdge(G', x, y)) \rightarrow ((x \in G \rightarrow y \in G) \wedge (y \in G \rightarrow x \in G))) \wedge Connect(G))\}$.

To get all the cliques of the graphs in $\mathcal{G}_0$, the graph calculus expression is: $\{G|SizeGEQ(G,3) \wedge \exists G'(GraphInSet(G', \mathcal{G}_0) \wedge InducedSub(G, G') \wedge \forall x, y((Vertex(x,y) \wedge Unique(x,y) \wedge x \in G \wedge y \in G) \rightarrow HasEdge(G,x,y)))\}$.

The meaning of notations, e.g., *HasEdge* and *SizeGEQ*, are presented in Table 2. □

3 GRAPH DATA MODEL

In this section, a new graph data model is proposed, and its basic unit is a graph.

Definition 3.1 (Graph Schema). A graph schema (abbr. schema) is denoted as $\mathcal{S} = (VL, EL, MP, \mathcal{A}, \lambda, \mathbb{C})$, where VL, EL, and $\mathcal{A}$ are the sets of vertex labels, edge labels, and attribute names, respectively. $MP = VL \times EL \times VL$ is the set of possible 1-hop metapaths. $\lambda : (VL \cup EL) \rightarrow 2^{\mathcal{A}}$ maps each label to the list of all its related attribute names, where $2^{\mathcal{A}}$ represents the power set of $\mathcal{A}$. $\mathbb{C}$ consists of the constraints on the graph, such as the consistency constraint, integrity constraint, and pattern constraint (i.e., the vertices and edges should match the given patterns).

For simplicity, we denote $\mathcal{S}.VL \cup \mathcal{S}.EL$ by $\mathcal{S}.\mathcal{T}$, so $\mathcal{S}.\mathcal{T}$ records all the labels in $\mathcal{S}$. Besides, $\mathcal{S}.\mathcal{T}_i$ represents the i-th label in $\mathcal{S}.\mathcal{T}$, and $\mathcal{S}.\lambda_{i,j}$ represents the j-th attribute name of label $\mathcal{S}.\mathcal{T}_i$. An example of a graph schema $\mathcal{S}$ is shown in Fig. 3. It is the graph schema of a coauthorship network, and VL = {Researcher}, EL = {Coauthor}, and $\mathcal{S}.\mathcal{T}$ = {Researcher, Coauthor}. Then, $\mathcal{S}.\mathcal{T}_1$ = "Researcher", and we have λ(Researcher) = {acmid, ieeeid, name}. It means that the vertices labeled "Researcher" have three attributes, i.e., the identifier in the ACM library, the identifier in the IEEE library, and the name. For simplicity, it is recorded that $\lambda_{1,1}$ = "acmid", $\lambda_{1,2}$ = "ieeeid", $\lambda_{1,3}$ = "name". Moreover, $\mathcal{S}.MP = \{(Researcher, Coauthor, Researcher)\}$. Note that two graph schemas $\mathcal{S}$ and $\mathcal{S}'$ are the same iff they have the same VL, EL, MP, $\mathcal{A}$, and $\mathbb{C}$. Besides, $\forall l \in \mathcal{S}.\mathcal{T}$, $\mathcal{S}.\lambda(l) = \mathcal{S}'.\lambda(l)$.

Moreover, a schema $\mathcal{S}_1$ is said to contain another schema $\mathcal{S}_2$ iff $\mathcal{S}_2.VL \subseteq \mathcal{S}_1.VL$, $\mathcal{S}_2.EL \subseteq \mathcal{S}_1.EL$, $\mathcal{S}_2.MP \subseteq \mathcal{S}_1.MP$, $\mathcal{S}_2.\mathcal{A} \subseteq \mathcal{S}_1.\mathcal{A}$, and $\forall i \in \mathcal{S}_2.\mathcal{T}$, $\mathcal{S}_2.\lambda(i) \subseteq \mathcal{S}_1.\lambda(i)$.

Definition 3.2 (Graph Instance). A graph instance (abbr. instance) over graph schema S is denoted as $\mathcal{I} = (V, E, \mathcal{T}, \alpha, \tau)$, where V and E are the sets of vertices and edges, respectively. $\mathcal{T} = \mathcal{S}.VL \cup \mathcal{S}.EL$

Table 2: Some notations of FOL and MSOL expressions used in this paper. Specifically, the *"Notation"* column shows the notations, and their meanings are presented in the *Meaning* column. Moreover, the corresponding formulae of the notations are presented in the *Formula* column, and the last column reflects whether the notations can be used in FOL and MSOL expressions.

Notation	Meaning	Formula	FOL/MSOL
Equal(t_1, t_2) (or $t_1 = t_2$)	two elements t_1 and t_2 are the same	$t_1[1] = t_2[1] \wedge \cdots \wedge t_1[m] = t_2[m]$ where m is the number of the components in t_1	Both
Vertex($v_1, \cdots, v_n$)	n elements representing vertices	$\text{Vertex}(v_1) \wedge \cdots \wedge \text{Vertex}(v_n)$	Both
Edge($e_1, \cdots, e_n$)	n elements representing edges	$\text{Edge}(e_1) \wedge \cdots \wedge \text{Edge}(e_n)$	Both
Complete($v_1, \cdots, v_n$)	vertices $v_1, \cdots, v_n$ form a complete graph	$\exists e_1, \cdots, e_{n(n-1)}\,(Edge(e_1, \cdots, e_{n(n-1)}) \wedge e_1[src] = v_1 \wedge e_1[tgt] = v_2 \wedge e_2[src] = v_2 \wedge e_2[tgt] = v_1 \wedge e_3[src] = v_1 \wedge e_1[tgt] = v_3 \wedge \cdots \wedge e_{n(n-1)}[src] = v_n \wedge e_{n(n-1)}[tgt] = v_{n-1})$	Both
Unique($t_1, \cdots, t_n$)	any two of the n elements are different	$\neg Equal(t_1, t_2) \wedge \cdots \wedge \neg Equal(t_1, t_n) \wedge \neg Equal(t_2, t_3) \wedge \cdots \wedge \neg Equal(t_2, t_n) \wedge \cdots \wedge \neg Equal(t_{n-1}, t_n)$	Both
Clique()	the constraints that a clique should satisfy	$\exists v_1, v_2, v_3(Vertex(v_1, v_2, v_3) \wedge Unique(v_1, v_2, v_3)) \wedge \forall u, v((Vertex(u, v) \wedge Unique(u, v)) \rightarrow Complete(u, v))$	Both
HasEdge(G, v_1, v_2)	there is an edge between v_1 and v_2 in G	$\exists t(Edge(t) \wedge t \in G \wedge Equal(v_1, t[src]) \wedge Equal(v_2, t[tgt]))$	MSOL
SizeGEQ(G, n)	there are at least n vertices in G	$\exists v_1, \cdots, v_n(Vertex(v_1, \cdots, v_n) \wedge v_1 \in G \wedge \cdots \wedge v_n \in G \wedge Unique(v_1, \cdots, v_n))$	MSOL
GraphEqual(G_1, G_2)	two graphs G_1 and G_2 are the same	$\forall t(t \in G_1 \rightarrow t \in G_2) \wedge \forall t(t \in G_2 \rightarrow t \in G_1)$	MSOL
GraphInSet($G, \mathcal{G}$)	graph G is in a graph set $\mathcal{G}$, and $\mathcal{G}$ contains k graphs $G_1 \cdots, G_k$	$GraphEqual(G, G_1) \vee \cdots \vee GraphEqual(G, G_k)$	MSOL
Subgraph(X, G)	graph X is a subgraph of graph G	$\exists t(t \in X) \wedge \forall t(t \in X \rightarrow t \in G)$	MSOL
InducedSub(X, G)	graph X is an induced subgraph of graph G	$Subgraph(X, G) \wedge \forall e_x((Edge(e_x) \wedge e_x \in G) \wedge e_x[src] \in X \wedge e_x[tgt] \in X) \rightarrow e_x \in X)$	MSOL
Connect(G)	graph G is connected	$\neg\exists X(InducedSub(X, G) \wedge \exists u(Vertex(u) \wedge u \in X) \wedge \exists u(Vertex(u) \wedge u \in G \wedge \neg(u \in X)) \wedge \forall x(\forall y((Vertex(x) \wedge x \in G \wedge Vertex(y) \wedge y \in G \wedge HasEdge(G, x, y)) \rightarrow ((x \in X \rightarrow y \in X) \wedge (y \in X \rightarrow x \in X)))))$	MSOL

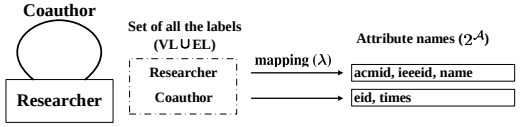


Figure 3: The graph schema $\mathcal{S}$ of the network in Fig. 1.

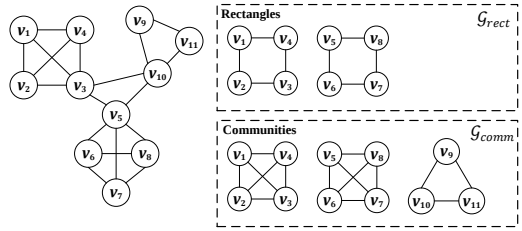


Figure 4: An Example of Graph Sets.

is the set of labels. $\alpha : (V \cup E) \rightarrow \mathcal{T}$ maps vertices and edges to their labels. $\tau : (V \cup E) \rightarrow dom(a_1) \times \cdots \times dom(a_k)$ maps a vertex or an edge to the list of its attribute values, where $a_1, \cdots, a_k$ are the attribute names of the input vertex or edge p (i.e., $\{a_1, \cdots, a_k\} = S.\lambda(\alpha(p))$), and $dom(a_i)$ is the domain of attribute a_i.

Fig. 1 is an example of a graph instance $\mathcal{I}$ over the graph schema in Fig. 3, and there are totally 12 vertices representing researchers in the graph instance. Moreover, there are 15 edges labeled "Coauthor", and each edge represents that the two vertices adjacent to it have coauthored at least three papers in 2018. Besides, $\mathcal{T}$ = {Researcher, Coauthor}. Denote the vertex representing *Jiawei Han* by v, and then we have $\alpha(v)$ = "Researcher", and $\tau(v)$ = ("81351593425", "37276626200", "Jiawei Han").

Definition 3.3 (Graph). A graph $G = (\mathcal{S}, \mathcal{I})$ consists of a graph schema $\mathcal{S}$, and a graph instance $\mathcal{I}$ over $\mathcal{S}$.

Example 3.4. Given the graph schema $\mathcal{S}$ in Fig. 3 and the graph instance $\mathcal{I}$ in Fig. 1, $G = (\mathcal{S}, \mathcal{I})$ is a graph, which stores the coauthorships of *"Jiawei Han"* in 2018. □

Definition 3.5 (Graph Set). A graph set is a set of graphs, and is denoted as $\mathcal{G} = (\widehat{G}, \mathcal{S})$, where $\widehat{G} = \{G_1, G_2, \cdots, G_k\}$ contains the graphs in $\mathcal{G}$, and $\mathcal{S} = G_1.\mathcal{S} = G_2.\mathcal{S} = \cdots = G_k.\mathcal{S}$ is the schema of the graphs.

Please note that the graphs in a graph set should always have similar semantics. For example, each graph in graph set $\mathcal{G}_1$ represents a rectangle. For another more practical example, each graph in graph set $\mathcal{G}_2$ models some people and their relationships in medical records. More examples of graph sets are provided in Example 3.6. For simplicity, the schema of a graph is sometimes omitted, if it is not to be used in the rest of this paper. In that condition, when we mention the vertices and edges in a graph G, we refer to the vertices and edges in $G.\mathcal{I}$.

Example 3.6. As shown in Fig. 4, graph G consists of 11 vertices, and its schema is denoted as $\mathcal{S}_1$. Then, $\mathcal{G} = (\{G\}, \mathcal{S}_1)$ is a graph set consisting of G. If rectangles in $\mathcal{G}$ are queried, graph set $\mathcal{G}_{\text{rect}}$, which contains all the rectangles in $\mathcal{G}$, can be obtained, and the schemas of the graphs in $\mathcal{G}_{\text{rect}}$ are all $\mathcal{S}_1$. Besides, if the communities of graphs in $\mathcal{G}$ is queried, graph set $\mathcal{G}_{\text{comm}}$, which consists of the three communities in G, is obtained. □

Since the basic unit of our graph data model is a graph, the operators of our graph data model are proposed in Section 4 to deal with sets of graphs.

4 GRAPH ALGEBRA

In this section, the graph algebra is proposed, and it consists of seven fundamental operators (i.e., projection, selection, reduce, cartesian product, align, difference, and map) and more addition operators (e.g., union, join, intersection, and maximal). These operators are building blocks of our graph algebra as well as graph data model, and

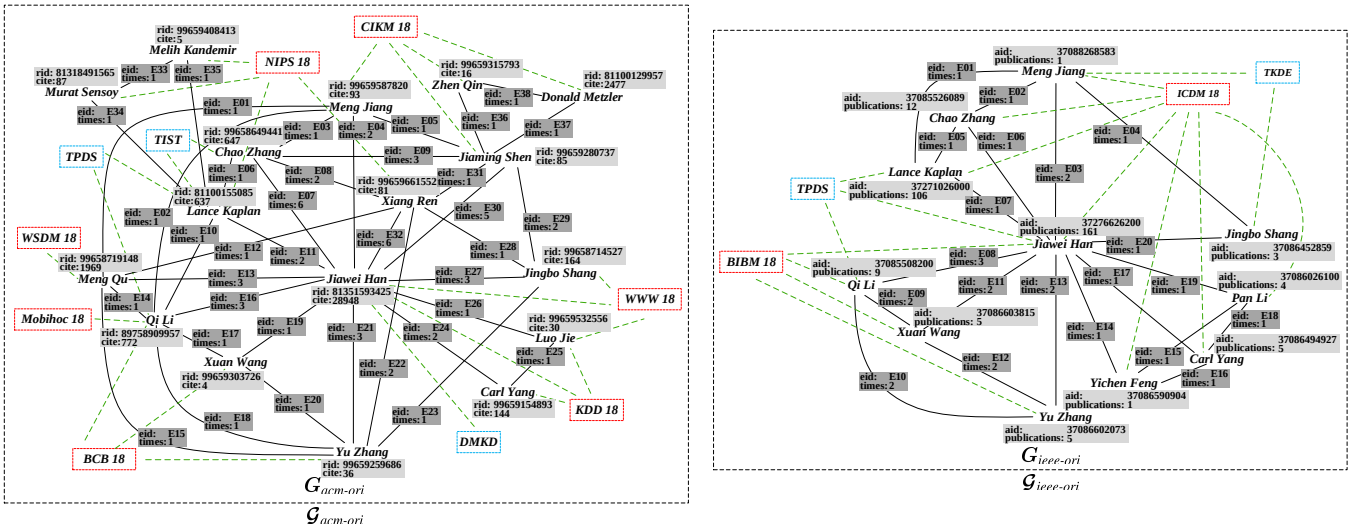


Figure 5: Two graph sets $\mathcal{G}_{acm\text{-}ori}$ and $\mathcal{G}_{ieee\text{-}ori}$ that consist of the snapshots $G_{acm\text{-}ori}$ and $G_{ieee\text{-}ori}$ respectively.

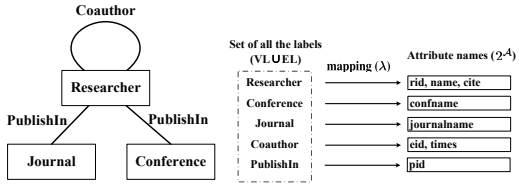


Figure 6: The graph schema $\mathcal{S}_{acm\text{-}ori}$ of $\mathcal{G}_{acm\text{-}ori}$.

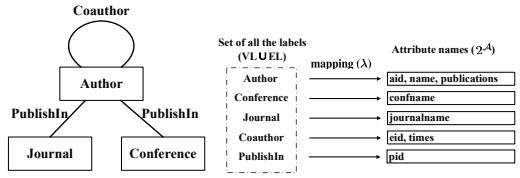


Figure 7: The graph schema $\mathcal{S}_{ieee\text{-}ori}$ of $\mathcal{G}_{ieee\text{-}ori}$.

are illustrated with the example of searching for *mc*s on the ACM and IEEE graphs. Specifically, ACM and IEEE are two famous organizations for researchers in the field of computer science, and we generate two snapshots (i.e., $G_{acm\text{-}ori}$ and $G_{ieee\text{-}ori}$) that store some publications and coauthorships of the authors in 2018 in the ACM and IEEE libraries, respectively.

Two graph sets $\mathcal{G}_{acm\text{-}ori}$ and $\mathcal{G}_{ieee\text{-}ori}$ that consist of $G_{acm\text{-}ori}$ and $G_{ieee\text{-}ori}$ respectively are shown in Fig. 5, and their graph schemas are shown in Fig. 6 and Fig. 7, respectively. In an ACM graph, there are three types of vertices and two types of edges. Specifically, a vertex can represent a researcher, a conference, or a journal. An edge (labeled *"Coauthor"*) between two researchers represents that these two researchers have coauthored ACM papers, and the edge has an attribute named *"times"* that records the number of ACM papers these two researchers have coauthored in 2018. An edge (labeled *"PublishIn"*) between a researcher and a conference (or a journal) indicates that the researcher has published at least one paper in the conference (or the journal) in 2018. For clarity, the vertices labeled *"Conference"* and *"Journal"* are marked with red and blue dashed lines, respectively, and the edges labeled *"PublishIn"* are marked with green dashed lines. Moreover, the attribute *"cite"* of a researcher represents the citation count of this researcher. Besides, it is similar for the IEEE graph set $\mathcal{G}_{ieee\text{-}ori}$. Specifically, the attribute *"publications"* of an *"Author"* refers to the number of papers the author have published.

Then, it is an interesting problem to find groups of researchers that have coauthored frequently according to $\mathcal{G}_{acm\text{-}ori}$ and $\mathcal{G}_{ieee\text{-}ori}$. We take *"Jiawei Han"*, who is a famous researcher in the area of data mining, as an example, and attemp to find out all the frequently-coauthored groups including him in 2018 by searching for all the *mc*s containing *"Jiawei Han"*. Besides, in this paper, two researchers are considered to coauthor frequently if they coauthored more than two papers in one year. Therefore, for each two researchers in an obtained frequently-coauthored group, they should have coauthored more than two papers in 2018. Note that for simplicity, only the coauthorships and publications in the snapshots (i.e., $G_{acm\text{-}ori}$ and $G_{ieee\text{-}ori}$) are considered in the rest of this paper.

4.1 Fundamental Operators

In this subsection, the fundamental operators of our graph algebra is proposed. For the convenience of introducing the proposed fundamental operators, a sample query, i.e., finding the *mc*s containing *"Jiawei Han"*, is used throughout this part.

4.1.1 Projection. Firstly, because the researchers and the coauthorships are focused on in the query, some vertices and attributes (e.g., the conference and journal vertices, and attribute *"cite"* of researchers) can be ignored. Therefore, the projection operator is proposed to remove the unconcerned labels and attributes. The definition of the projection operator is as follows.

Definition 4.1 (projection, ρ). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$, where $\widehat{G} = \{G_1, \cdots, G_k\}$, and a new schema $\mathcal{S}_{new}$ contained by $\mathcal{S}$, the projection operator projects the graphs in $\mathcal{G}$ according to $\mathcal{S}_{new}$ to get a new graph set. Specifically, $\rho_{\mathcal{S}_{new}}(\mathcal{G}) = (\widehat{G}', \mathcal{S}_{new})$, where $\widehat{G}' = \{G'_1, \cdots, G'_k\}$, and each graph G'_i ($i \in \{1, \cdots, k\}$) is obtained

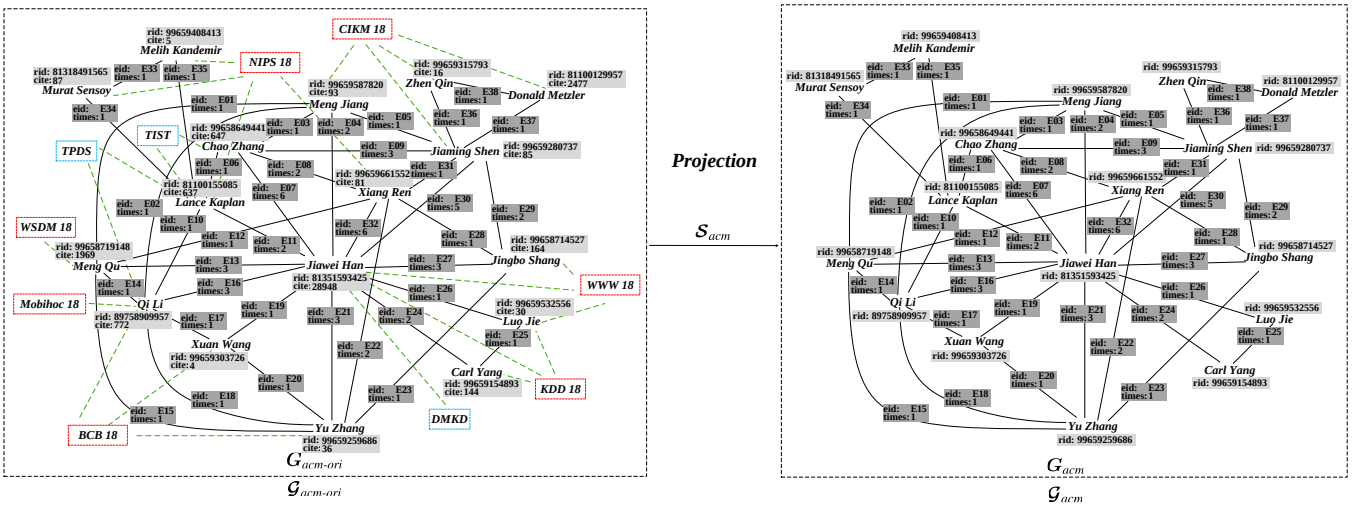


Figure 8: Removing the unconcerned vertices and attributes with the projection operator.

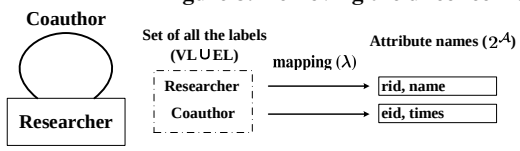


Figure 9: The graph schema $\mathcal{S}_{acm}$ of the ACM graph after the unconcerned labels and attributes are removed.

by removing from G_i the attributes not in $\mathcal{S}_{new}$, and the vertices and edges that no longer have corresponding vertices and edges respectively in $\mathcal{S}_{new}$.

Example 4.2. As shown in Fig. 8, $\mathcal{G}_{acm\text{-}ori}$ contains the graph of the ACM library in 2018, and its schema $\mathcal{S}_{acm\text{-}ori}$ is shown in Fig. 6. During the process of *mc* searching, because the *"Conference"* vertices, *"Journal"* vertices, the *"PublishIn"* edges, and the *"cite"* attribute of the researchers are unconcerned, a new schema is constructed to remove these unnecessary information. The new schema $\mathcal{S}_{acm}$ is shown in Fig. 9. After the projection operator is applied, a new graph set $\mathcal{G}_{acm}$ is obtained. Specifically, in the graph G_{acm} in $\mathcal{G}_{acm}$, the vertices labeled *"Conference"* or *"Journal"*, and edges labeled *"PublishIn"* are removed, since their corresponding vertices and edges are removed in $\mathcal{S}_{acm}$. Moreover, researchers no longer have attribute *"cite"*. □

Note that in Sections 4.1.1–4.1.3, we focus on the ACM graph, and show how to use the proposed operators to process the ACM graph. Meanwhile, the IEEE graph is processed in the similar methods, and the details are not presented.

4.1.2 Selection. After the unconcerned vertices and attributes are removed, we focus on finding the *mc*s in the obtained graph set. It is noted that the vertices in an *mc* containing *"Jiawei Han"* should all be the 1-hop neighbors of *"Jiawei Han"* or *"Jiawei Han"* himself, and these 1-hop neighbors are called the candidate vertices. Meanwhile, the other vertices are called the impossible vertices. Therefore, to further prune the impossible vertices, the selection operator is utilized to obtain the induced subgraph of the candidate vertices, and this graph contains all the vertices and edges that may exist in the cliques of *"Jiawei Han"*. This induced subgraph is also called the 1-hop-neighbor graph of *"Jiawei Han"*.

Definition 4.3 (selection, σ). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$ and an FOL expression d, the selection operator returns a graph set $(\sigma_d(\mathcal{G}) = (\widehat{G}_0, \mathcal{S}))$ that contains all the subgraphs of graphs in $\mathcal{G}$, and the subgraphs satisfy the constraint in d.

Please note that the notations that can be utilized in the FOL expressions, such as *Vertex* and *Complete* in Table 2, can be used in the constraint expression d. Besides, when an edge is selected to exist in a subgraph, the adjacent vertices of that edge are also selected.

Example 4.4. Given $\mathcal{G}_{acm}$ obtained in Fig. 8, in the *mc*s containing *"Jiawei Han"*, there are two types of edges: (1) the edges between *"Jiawei Han"* and his 1-hop neighbors; (2) the edges between two 1-hop neighbors of *"Jiawei Han"*. Therefore, constraint $d =$ *"*$d_- \vee d_\triangle$*"* is set, where

$$d_- = \exists v_1, v_2 (Vertex(v_1, v_2) \wedge v_1.name = \text{"Jiawei Han"} \wedge Unique(v_1, v_2) \wedge Complete(v_1, v_2)) \wedge \neg \exists v_1, v_2, e_1, e_2, x (Unique(v_1, v_2, e_1, e_2, x)),$$

and d_- is used to find the first type of edges. Specifically, $Complete(v_1, v_2)$ requires two vertices v_1 and v_2 to have two opposite edges between them and form a complete graph. $Unique(v_1, v_2, e_1, e_2, x)$ means that there are only four components in the obtained graph, i.e., two vertices v_1, v_2 and two opposite edges e_1, e_2 between them, and x represents a vertex or an edge. Besides, we set

$$d_\triangle = \exists v_1, v_2, v_3 (Vertex(v_1, v_2, v_3) \wedge v_1.name = \text{"Jiawei Han"} \wedge Unique(v_1, v_2, v_3) \wedge Complete(v_1, v_2, v_3)) \wedge \neg \exists v_1, v_2, v_3, e_1, \cdots, e_6, x (Unique(v_1, v_2, v_3, e_1, \cdots, e_6, x))$$

to find all the triangles containing the second type of edges. Then, the selection operator is applied, and a new graph set $\mathcal{G}_{1\text{-}hop}$ is obtained as shown in Fig. 10. The graphs in $\mathcal{G}_{1\text{-}hop}$ contain all the 1-hop neighbors of *"Jiawei Han"* and the edges that may appear in the resultant cliques. The edges of the first type are stored in $G_{n_1}, \cdots, G_{n_{12}}$, while the second type of edges are stored in $G_{\triangle_1}, \cdots, G_{\triangle_{20}}$. Meanwhile, the impossible vertices and the edges adjacent to these vertices are

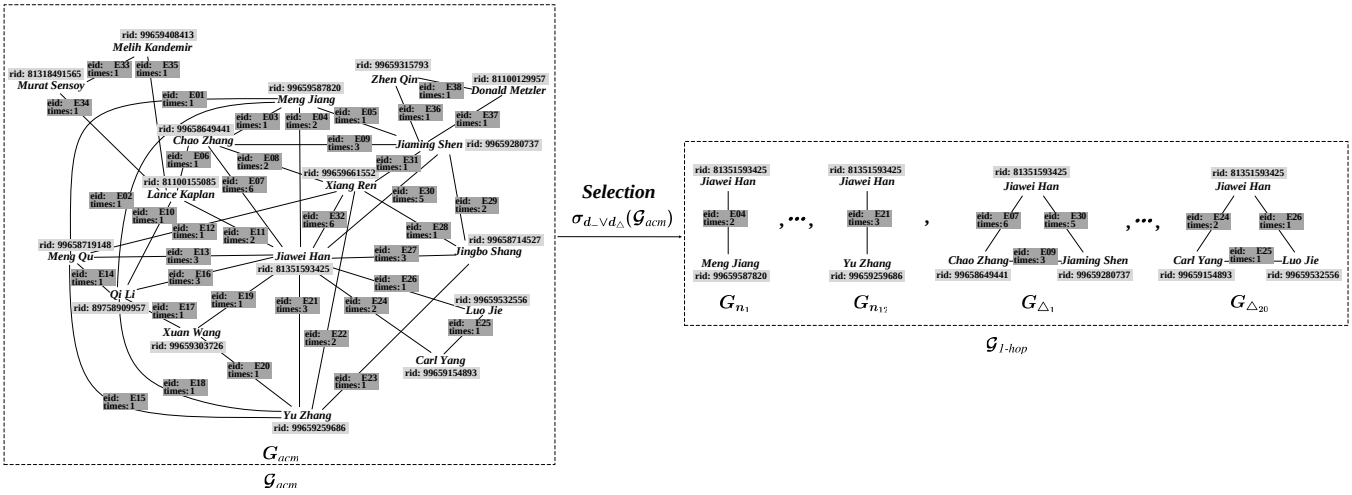


Figure 10: Finding the induced subgraph of the 1-hop neighbors of *"Jiawei Han"*.

pruned. Next, the problem is to combine these graphs into a new big graph, and then find the *mcs* . □

4.1.3 Reduce. The reduce operator is proposed to merge the graphs in a graph set into a new graph, and it is defined as follows.

Definition 4.5 (Reduce, γ). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$, $\gamma(\mathcal{G}) = \mathcal{G}_0 = (\widehat{G}_0, \mathcal{S})$, where $\widehat{G}_0 = \{G_0\}$ and G_0 is a graph generated by putting all the graphs in $\mathcal{G}$ together. Meanwhile, if two vertices (or edges) have the same labels and attributes, they are merged.

Example 4.6. As shown in Fig. 11, the 32 graphs in $\mathcal{G}_{1\text{-}hop}$ are put into the same graph G_a, and the vertices and the edges with the same labels and attributes are merged. For example, the vertex representing *"Jiawei Han"* in $G_{n_1}, \cdots, G_{\triangle_{20}}$ are merged in G_a. □

4.1.4 Cartesian Product. After the reduce operator is applied, the 1-hop-neighbor graph of *"Jiawei Han"* on the ACM graph is obtained, and is stored in $\mathcal{G}_a$. With the similar method, that on the IEEE graph is obtained and stored in $\mathcal{G}_i$. Then, the graphs in these two graph set should be put into the same graph, and then we can check whether two researchers have coauthored frequently based on their ACM papers and IEEE papers at the same time. However, these two graph sets have different schemas, because the researchers and coauthorships can have different labels and attributes in ACM and IEEE libraries. Therefore, the cartesian product operator is proposed to combine two graph sets with different schemas.

Before the formal definition of the cartesian product operator is presented, the definition of the union of graphs is given first. The union of two graphs G_a and G_i is denoted as $G_a \cup_g G_i$, and its task is to combine two graphs and generate a new graph. Let $AddPre(\{l_1, \cdots, l_k\}, x) = \{{}^x l_1, \cdots, {}^x l_k\}$ be a method to add a prefix x to each label or attribute in a list $\{l_1, \cdots, l_k\}$, where ${}^x l_j = x + \text{"."} + l_j$. Given $G_a = (\mathcal{S}_a, \mathcal{I}_a)$ from $\mathcal{G}_a$ and $G_i = (\mathcal{S}_i, \mathcal{I}_i)$ from $\mathcal{G}_i$, let $G = G_a \cup_g G_i = (\mathcal{S}, \mathcal{I})$. Then, we have $\mathcal{S}.X = AddPre(\mathcal{S}_a.X, a) \cup AddPre(\mathcal{S}_i.X, i)$, $X \in \{VL, EL, MP\}$, where $AddPre(\mathcal{S}_a.MP, a)$ means to add a prefix a to each label in $S_a.MP$. Note that the prefixes *"a"* and *"i"* are from $\mathcal{G}_a$ and $\mathcal{G}_i$, respectively. For the obtained graph G, it satisfies: (1) $\mathcal{S}.\mathcal{A} = \mathcal{S}_a.\mathcal{A} \cup \mathcal{S}_i.\mathcal{A}$; (2) $\forall l \in \mathcal{S}_a.\mathcal{T}, \mathcal{S}.\lambda({}^a l) = \mathcal{S}_a.\lambda(l), \forall l \in \mathcal{S}_i.\mathcal{T}, \mathcal{S}.\lambda({}^i l) = \mathcal{S}_i.\lambda(l)$; (3) $\mathcal{I}.V = \mathcal{I}_a.V \cup \mathcal{I}_i.V$; (4) $\mathcal{I}.E = \mathcal{I}_a.E \cup \mathcal{I}_i.E$; (5) $\mathcal{I}.\mathcal{T} = AddPre(\mathcal{I}_a.\mathcal{T}, a) \cup AddPre(\mathcal{I}_i.\mathcal{T}, i)$; (6) $\forall t \in (\mathcal{I}_a.V \cup \mathcal{I}_a.E), \mathcal{I}_a.\alpha(t) = l \rightarrow \mathcal{I}.\alpha(t) = {}^a l$, and $\forall t \in (\mathcal{I}_i.V \cup \mathcal{I}_i.E), \mathcal{I}_i.\alpha(t) = l \rightarrow \mathcal{S}.\alpha(t) = {}^i l$; (7) $\mathcal{I}.\tau = \mathcal{I}_a.\tau \cup \mathcal{I}_i.\tau$.

Definition 4.7 (cartesian product, $\times$). Given two graph sets $\mathcal{G}_1 = (\widehat{G}_1, \mathcal{S}_1)$ and $\mathcal{G}_2 = (\widehat{G}_2, \mathcal{S}_2)$. $\mathcal{G}_1 \times \mathcal{G}_2 = \mathcal{G}_0 = (\widehat{G}_0, \mathcal{S}_0)$, where $\widehat{G}_0 = \{G_{1_i} \cup_g G_{2_j} | G_{1_i} \in \widehat{G}_1, G_{2_j} \in \widehat{G}_2\}$.

Example 4.8. As shown in Fig. 12, $\mathcal{G}_a$ and $\mathcal{G}_i$ are two graph sets that contain the 1-hop-neighbor graphs of *"Jiawei Han"* in ACM and IEEE graphs, respectively. The schema of $\mathcal{G}_a$ is shown in Fig. 9, and the schema of $\mathcal{G}_i$ is shown in Fig. 13. It is noted that a researcher in the ACM graph is called an author in the IEEE graph, and the identification of an author in the IEEE graph is stored in the attribute named *"aid"* rather than *"rid"* in the ACM graph. Then, the cartesian product operator is utilized to combine these two graph sets to consider the coauthorships between researchers in both two organizations, and the resultant graph set $\mathcal{G}_{combine}$ is shown in Fig. 12. Note that $G_{combine}$ consists of G_a and G_i. Moreover, the schema $\mathcal{S}_{combine}$ of $\mathcal{G}_{combine}$ is presented in Fig. 14. □

4.1.5 Align. After the 1-hop-neighbor graphs of *"Jiawei Han"* on ACM and IEEE are united with the cartesian product operator, there are two vertices named *"Jiawei Han"*, because *"Jiawei Han"* has published papers in both ACM and IEEE publications. At the same time, there are also two corresponding vertices for some other researchers (e.g., *"Meng Jiang"*). Then, these vertices representing the same researcher should be aligned. After the vertices representing the same researchers are aligned, there may be two edges between two researchers. For example, after the two vertices representing *"Jiawei Han"* and two vertices representing *"Meng Jiang"* are aligned respectively, there are two edges between the aligned vertices. These two edges contain the numbers of the coauthored ACM papers and IEEE papers respectively, and the edges can also be aligned to store the number of coauthored ACM and IEEE papers in one edge. For the above reasons, the align operator is proposed.

Definition 4.9 (align, ζ). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$, $\zeta_h(\mathcal{G}) = \mathcal{G}_0 = (\widehat{G}_0, \mathcal{S}_0)$, where h is the alignment condition in the form of

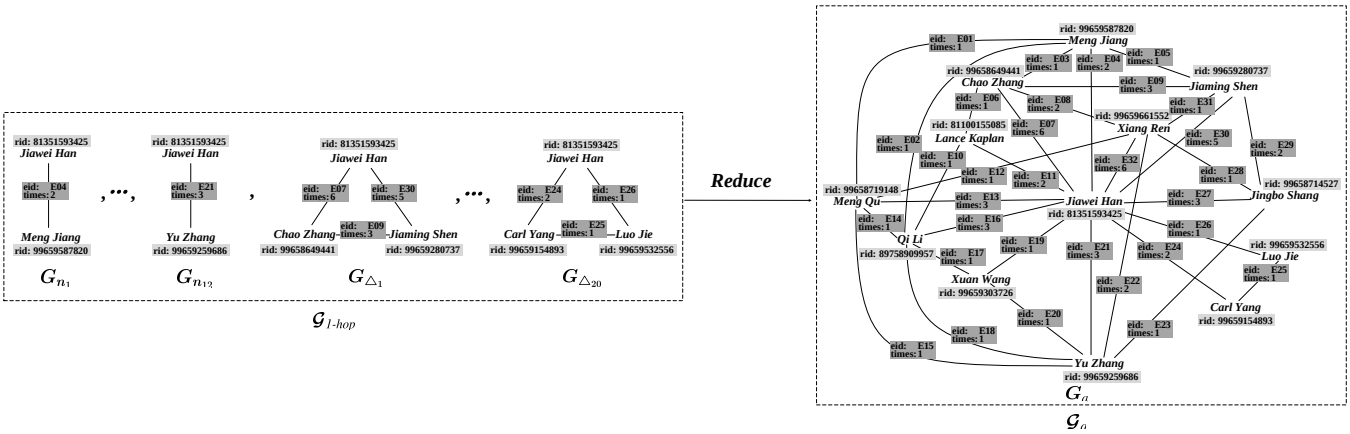


Figure 11: Generating the 1-hop-neighbor graph of "*Jiawei Han*" with the reduce operator.

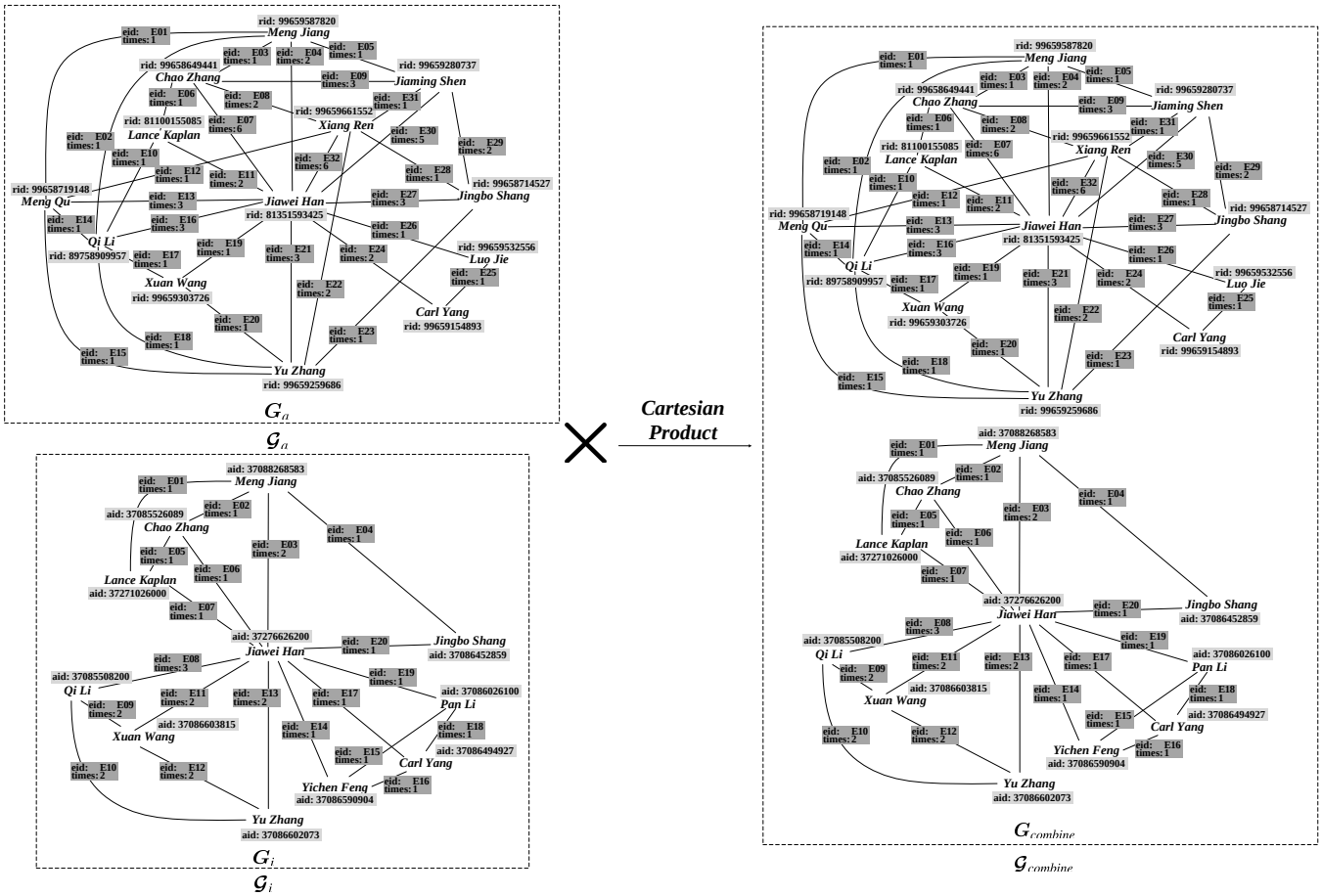


Figure 12: Combining the 1-hop-neighbor graphs of "*Jiawei Han*" in the ACM graph and IEEE graph into the same graph set with the cartesian product operator.

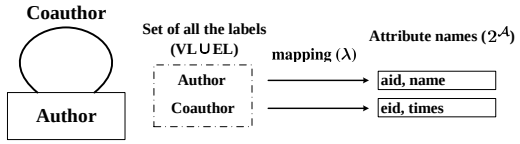


Figure 13: The graph schema $\mathcal{S}_{ieee}$ of G_i in Fig. ??.

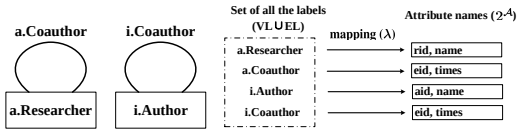


Figure 14: The schema $\mathcal{S}_{combine}$ of the graph set $\mathcal{G}_{combine}$ obtained by the cartesian product operator.

"$l_1, l_2 \Rightarrow l_0 || attr_{i_1} = attr_{j_1}$ as $a_1, \cdots, attr_{i_k} = attr_{j_k}$ as a_k", l_1 and l_2 are two labels in $\mathcal{G}$, l_0 is the new label of the vertices and edges labeled l_1 and l_2 originally, and $attr_{i_1}, \cdots, attr_{i_k}$ are k attribute names of l_1 while $attr_{j_1}, \cdots, attr_{j_k}$ are those of l_2. Moreover, a_1 is the new name of attribute $attr_{i_1}$ (of l_1) and $attr_{j_1}$ (of l_2), $\mathcal{S}_0.\mathcal{T} = \mathcal{S}.\mathcal{T} \cup \{l_0\} - \{l_1, l_2\}$, l_1 and l_2 in $\mathcal{S}_0.MP$ are changed to be l_0, and $\mathcal{S}_0.\lambda(l_0) = \{a_1, \cdots, a_k\} \cup AddPre(\mathcal{S}.\lambda(l_1) - \{attr_{i_1}, \cdots, attr_{i_k}\}, l_1) \cup AddPre(\mathcal{S}.\lambda(l_2) - \{attr_{j_1}, \cdots, attr_{j_k}\}, l_2)$.

Specifically, for each graph in $\mathcal{G}$, if two vertices labeled l_1 and l_2 respectively satisfy the alignment condition, and can only be aligned with each other according to the condition h (uniqueness constraint), they are aligned and their attributes are merged into the new vertex. Two edges labeled l_1 and l_2 respectively can be aligned iff they satisfy the alignment condition, have the same adjacent vertices (adjacency constraint), and each of these two edges can only be aligned with the other edge (uniqueness constraint). For a vertex or an edge labeled l_1 or l_2 that cannot be aligned with any other vertex or edge, its label is also set l_0, and the values of the unknown attributes are set "NULL" in default.

Example 4.10. Given $\mathcal{G}_{combine}$ obtained in Fig. 12, the vertices that represent the same researchers are firstly aligned. Specifically, we perform $\mathcal{G}_{alignv} = \zeta_{h_v}(\mathcal{G}_{combine})$, where

$$h_v = a.Researcher, i.Author \Rightarrow Researcher \,\|\, name = name \text{ as } name.$$

Then, a vertex labeled *"a.Researcher"* and a vertex labeled *"i.Author"* are aligned if they have the same names and satisfy the uniqueness constraint.

Next, two edges are aligned if their adjacent researchers are the same, and the expression is $\mathcal{G}_{aligne} = \zeta_{h_e}(\mathcal{G}_{alignv})$, where

$$h_e = a.Coauthor, i.Coauthor \Rightarrow Coauthor.$$

In alignment condition h_e, the constraints after "‖" are omitted, and it means that every two edges labeled *"a.Coauthor"* and *"i.Coauthor"* respectively can be aligned as long as they have the same adjacent vertices and satisfy the uniqueness constraint.

After these two steps of alignment, the vertices and edges are aligned. Note that the attribute names of the aligned vertices labeled *"Researcher"* are *"name"*, *"a.Researcher.rid"*, and *"i.Author.aid"*, and those of the aligned edges labeled *"Coauthor"* are *"a.Coauthor.eid"*, *"a.Coauthor.times"*, *"i.Coauthor.eid"*, and *"i.Coauthor.times"*. In order to make the attribute names simpler and more concise, we use the align operator again to rename the attribute names. The expression is $\mathcal{G}_{align} = \zeta_{h_{re}}(\zeta_{h_{rv}}(\mathcal{G}_{aligne}))$, where

$$\begin{aligned} h_{rv} = {} & Researcher, Researcher \Rightarrow Researcher \,\| \\ & name = name \text{ as } name, \\ & a.Researcher.rid = a.Researcher.rid \text{ as } a.rid, \\ & i.Author.aid = i.Author.aid \text{ as } i.aid. \end{aligned}$$

$$\begin{aligned} h_{re} = {} & Coauthor, Coauthor \Rightarrow Coauthor \,\| \\ & a.Coauthor.eid = a.Coauthor.eid \text{ as } a.eid, \\ & a.Coauthor.times = a.Coauthor.times \text{ as } a.times, \\ & i.Coauthor.eid = i.Coauthor.eid \text{ as } i.eid, \\ & i.Coauthor.times = i.Coauthor.times \text{ as } i.times. \end{aligned}$$

Then, the graph set $\mathcal{G}_{align}$ is obtained as shown in Fig. 15, and its schema is shown in Fig. 16. It is noted that each vertex labeled *"Researcher"* in $\mathcal{G}_{align}$ has two attributes, i.e. its *rid* in ACM and its *aid* in IEEE. Besides, each edge in $\mathcal{G}_{align}$ has four attributes, namely the id of this coauthorship in ACM and IEEE, and the numbers of coauthored papers in ACM and IEEE. By summing up the number of coauthored paper of two researchers in ACM and IEEE, the total number of papers that two researchers have coauthored can be obtained. □

After the align operator is applied and $\mathcal{G}_{align}$ is obtained, the number of coauthored ACM and IEEE papers of two researchers are stored in one graph. Next, we first find all the frequently-coauthored cliques in $\mathcal{G}_{align}$, and then obtain the *mc*s. The selection operator is utilized again to find all the frequently-coauthored cliques in $\mathcal{G}_{align}$ as shown in Fig. 17, and the schema of $\mathcal{G}_{clique}$ is the same as $\mathcal{G}_{align}$ (i.e., $\mathcal{S}_{clique} = \mathcal{S}_{align}$). In detail, we utilize the contraint

$$\begin{aligned} d_c = {} & Clique() \\ & \wedge \forall e(Edge(e) \rightarrow e.a.times + e.i.times \geq 3) \\ & \wedge \exists x(Vertex(x) \wedge x.name = \text{"Jiawei Han"}) \end{aligned}$$

in the selection operator, where the first line of the constraint require the obtained subgraphs to be cliques, *"*$\forall e(Edge(e) \rightarrow e.a.times + e.i.times \geq 3)$*"* requires that each two researchers in the obtained clique should have coauthored at least three papers in 2018 (i.e., frequently-coauthored), and *"*$\exists x(Vertex(x) \wedge x.name =$ *"Jiawei Han"*$)$*"* requires the obtained cliques to contain *"Jiawei Han"*. Then, the problem of finding *mc*s containg *"Jiawei Han"* becomes extracting the maixmal ones from the obtained cliques.

4.1.6 Difference.

Before the method to extract the *mc*s is proposed, the difference operator is first defined, because it is necessary for judging whether a clique is maximal.

Definition 4.11 (difference, −). Given two graph sets $\mathcal{G}_1 = (\widehat{G}_1, \mathcal{S}_1)$ and $\mathcal{G}_2 = (\widehat{G}_2, \mathcal{S}_2)$, $\mathcal{G}_1 - \mathcal{G}_2 = \mathcal{G}_0 = (\widehat{G}_0, \mathcal{S}_0)$, where $\widehat{G}_0 = \widehat{G}_1 - \widehat{G}_2$ and $\mathcal{S}_0 = \mathcal{S}_1 = \mathcal{S}_2$.

Two graphs G_1 and G_2 are the same iff their schemas and instances are both the same. In the process of difference $\mathcal{G}_1 - \mathcal{G}_2$, for each graph G_i in $\mathcal{G}_1$, if there exists a graph G_j in $\mathcal{G}_2$ satisfying that G_i is the same as G_j, then G_i is removed from $\mathcal{G}_1$.

Please note that the intersection of two graph sets $\mathcal{G}_1$ and $\mathcal{G}_2$ can be expressed as $\mathcal{G}_1 \cap \mathcal{G}_2 = \mathcal{G}_1 - (\mathcal{G}_1 - \mathcal{G}_2)$. The intersection operator is an additional operator, and is proposed later in Section 4.2.2.

Example 4.12. As shown in Fig. 18, the researchers who have coauthored ACM papers with *"Jiawei Han"* in 2018 and 2019 are stored in $\mathcal{G}_{co\text{-}2018}$ and $\mathcal{G}_{co\text{-}2019}$, respectively. Then, we are going to

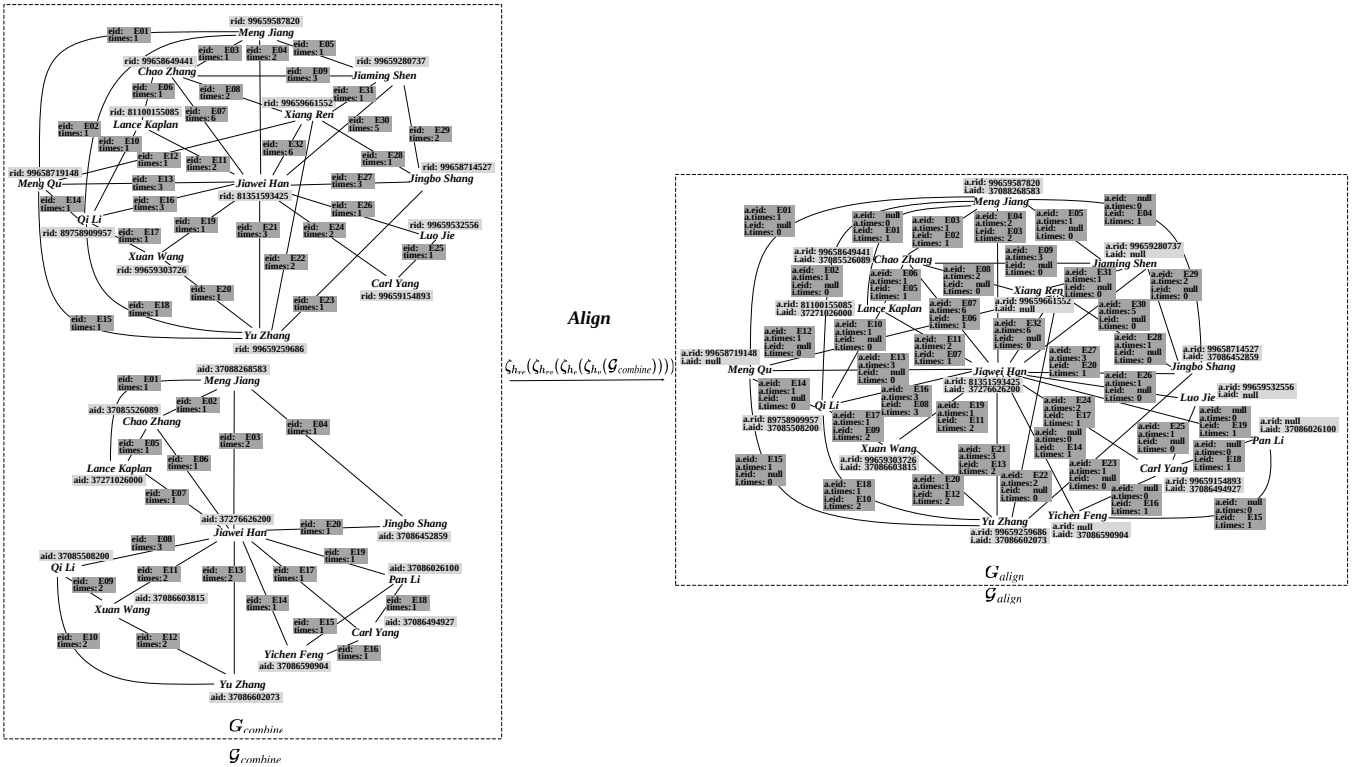


Figure 15: Aligning the vertices and edges in $\mathcal{G}_{combine}$.

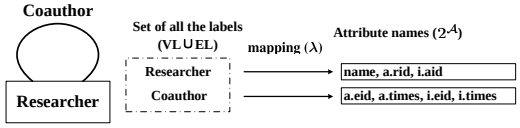


Figure 16: The schema $\mathcal{S}_{align}$ of the graph set $\mathcal{G}_{align}$ obtained by the align operator.

find the researchers who have coauthored ACM papers with *"Jiawei Han"* in 2019, but not in 2018, and these researchers are probably the new coauthors of *"Jiawei Han"*. Specifically, the difference operator is applied, and the results are stored in $\mathcal{G}_{co\text{-}new}$. In detail, the three graphs containing *"Lingrui Gan"*, *"Zongyi Wang"*, and *"Ailing Piao"* respectively are in $\mathcal{G}_{co\text{-}new}$, because these researchers are new coauthors for *"Jiawei Han"*. □

4.1.7 Map. After the difference operator is presented, we start to find *mc*s with it and the map operator. Specifically, the map operator decomposes a graph set $\mathcal{G}$ into several graph sets, each of which contains one of the graphs in $\mathcal{G}$, and then deals with these graph sets simultaneously. Therefore, it can be used to check whether the cliques obtained are maximal individually. The definition of the map operator is as follows.

Definition 4.13 (Map, μ). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$, by applying the map operator on $\mathcal{G}$, we obtain $\mathcal{G}_0 = \mu(\mathcal{G}||\mathcal{G}' : \mathcal{E}(\mathcal{G}'))$. Specifically, the map operator first generates k graph sets, each of which consists of one of the graphs in $\widehat{G} = \{G_1, \cdots, G_k\}$. Then, for each of these generated graph set (denoted as $\mathcal{G}'$), the operators specified in graph algebra expression $\mathcal{E}$ are applied on $\mathcal{G}'$. The graphs in the graph sets obtained by processing each $\mathcal{G}'$ with $\mathcal{E}$ are finally put into the same graph set $\mathcal{G}_0$.

Example 4.14. As shown in Fig. 17, $\mathcal{G}_{clique}$ is obtained, and it consists of all the frequently-coauthored cliques containing *"Jiawei Han"*. Then, with the map operator $\mu(\mathcal{G}_{clique}||\mathcal{G}' : \mathcal{E}(\mathcal{G}'))$, each clique in $\mathcal{G}_{clique}$ can be checked whether it is maximal. Specifically, the expression $\mathcal{E}(\mathcal{G}') = \gamma((\mathcal{G}' \bowtie \mathcal{G}_{clique}) \cap \mathcal{G}_{clique}) \cap \mathcal{G}'$ is used. Denote the graph in $\mathcal{G}'$ by G', $\mathcal{E}(\mathcal{G}')$ returns the graph set $\mathcal{G}'$ itself if G' is maximal in $\mathcal{G}_{clique}$. Otherwise, $\mathcal{E}(\mathcal{G}')$ returns an empty graph set. Please note that $\bowtie$ and $\cap$ are two additional operators that can be expressed with the fundamental operators proposed in this section, and the details of this expression are presented in the next subsection when the additional operator *Maximal* is proposed.

As shown in Fig. 19, let $\mathcal{G}_{clique} = (\widehat{G}_{clique}, \mathcal{S}_{clique})$ as shown in Fig. 17, where $\mathcal{S}_{clique} = \mathcal{S}_{align}$. When $\mathcal{G}' = (\widehat{G}' = \{G_{clique1}\}, \mathcal{S}_{clique})$ or $\mathcal{G}' = (\widehat{G}' = \{G_{clique5}\}, \mathcal{S}_{clique})$, the expression $\mathcal{E}(\mathcal{G}')$ returns $\mathcal{G}'$ itself, because $G_{clique1}$ and $G_{clique5}$ are the maximal graphs in G_{clique}. However, when $\mathcal{G}'$ contains $G_{clique2}$, $G_{clique3}$, or $G_{clique4}$ respectively, because they are all the subgraphs of $G_{clique5}$, the expression $\mathcal{E}(\mathcal{G}')$ returns empty sets.

Next, the obtained graph sets are united, and the final graph set $\mathcal{G}_{mc}$ is generated. It is noted that the graphs in $\mathcal{G}_{mc}$ are all the *mc*s containing *"Jiawei Han"*, and each two researchers in an *mc* are frequently-coauthored (i.e. frequently-coauthored groups). □

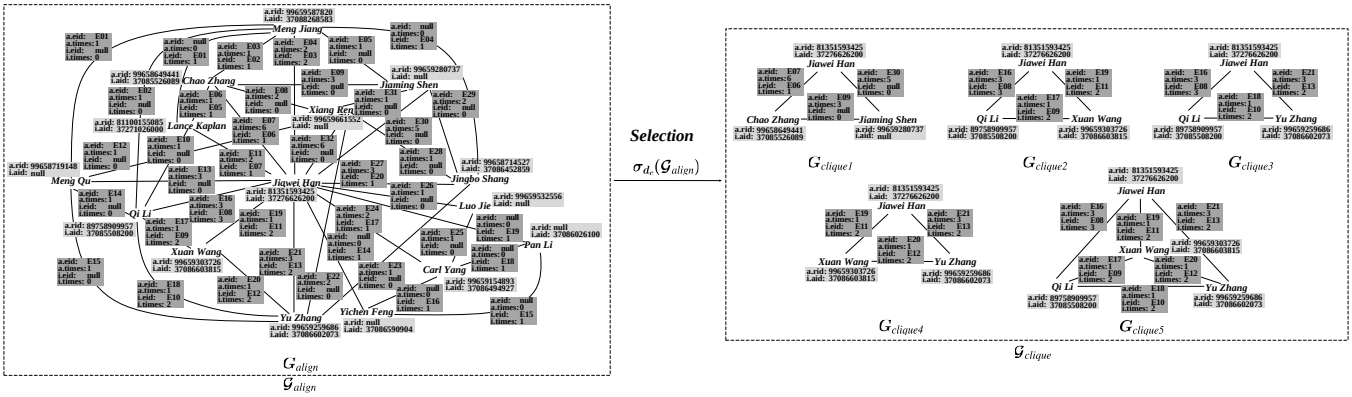


Figure 17: Finding the cliques containing *"Jiawei Han"* in $\mathcal{G}_{align}$. In detail, d_c = *Clique*() $\wedge \forall e$(*Edge(e)* $\rightarrow$ *e.a.times + e.i.times* $\geq 3) \wedge \exists x$(*Vertex*$(x) \wedge$ *x.name = "Jiawei Han"*).

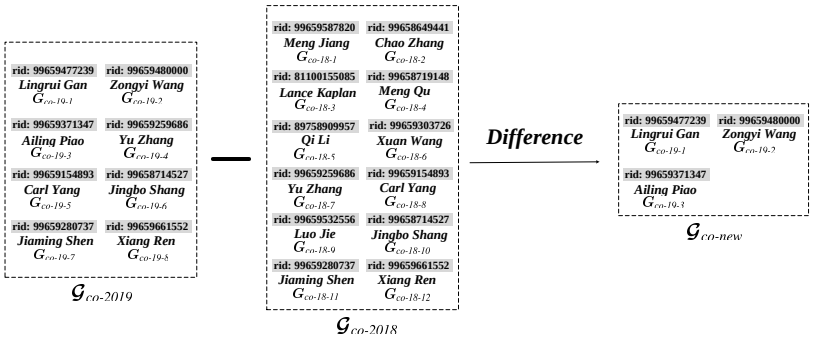


Figure 18: Finding the researchers that have coauthored with *"Jiawei Han"* in 2019 but not in 2018 with the diffenrece operator.

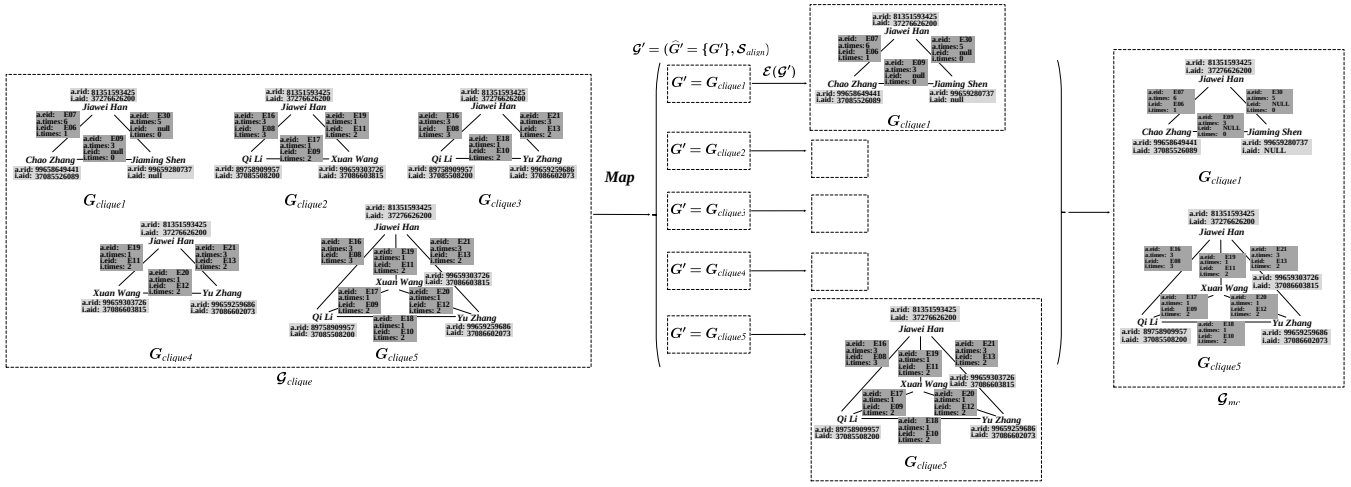


Figure 19: Obtaining the maximal graph in $\mathcal{G}_{clique}$ with the map operator. Specifically, $\mathcal{E}(\mathcal{G}') = \gamma((\mathcal{G}' \bowtie \mathcal{G}_{clique}) \cap \mathcal{G}_{clique})$ is used to judge whether a graph set $\mathcal{G}'$ contains a graph that is maximal in $\mathcal{G}_{clique}$.

4.2 Additional Operators

Besides the fundamental operators, we also propose some additional operators. The additional operators can be expressed with the fundamental operators, but sometimes it is more convenient to utilize the additional operators directly. In this section, four additional operators, i.e., join, intersection, maximal, and union, are presented to perform queries more conveniently. Note that there are still many other possible additional operators, which can also be expressed

by our powerful fundamental operators. The additional operators proposed in this section are just some typical examples.

4.2.1 Join. Given the graph set $\mathcal{G}_a = (\widehat{G}_a, \mathcal{S}_{acm})$, which consists of the 1-hop-neighbor graph G_a of *"Jiawei Han"* in the ACM library in 2018 (i.e., $\widehat{G}_a = \{G_a\}$, as shown in Fig. 11), suppose we want to update G_a with some coauthorships of *"Jiawei Han"* in 2019, and the new coauthorships are stored in a graph in graph set $\mathcal{G}_n = (\widehat{G}_n, \mathcal{S}_{acm})$. Note that $\mathcal{G}_a$ and $\mathcal{G}_n$ have the same schema. The key idea is to form a new graph with the graph in $\mathcal{G}_a$ and the graph in $\mathcal{G}_n$, and align the vertices that represent the same researchers. Then, using the fundamental operators, the expression is:

$$\zeta_{h_1}(\cdots(\zeta_{h_m}(\mathcal{G}_a \times \mathcal{G}_n))\cdots)$$

where $m = |\mathcal{S}.\mathcal{T}|$, and without loss of generality, $\mathcal{S}.\mathcal{T}_1$ to $\mathcal{S}.\mathcal{T}_q$ are the labels of edges (abbr. $\mathcal{T}_1$ to $\mathcal{T}_q$), while $\mathcal{S}.\mathcal{T}_{q+1}$ to $\mathcal{S}.\mathcal{T}_m$ are the labels of vertices (abbr. $\mathcal{T}_{q+1}$ to $\mathcal{T}_m$). Specifically, for $p \in [1, m]$, the constraint $h_p =$ "${}^a\mathcal{T}_p, {}^n\mathcal{T}_p \Rightarrow \mathcal{T}_p || \lambda_{p,1} = \lambda_{p,1}$ as $\lambda_{p,1}, \cdots, \lambda_{p,k} = \lambda_{p,k}$ as $\lambda_{p,k}$", where k is the number of attributes of label $\mathcal{T}_p$, and $\lambda_{p,i}$ is the i-th attribute name of label $\mathcal{T}_p$ in $\mathcal{S}$. Note that for each $\lambda_{p,i} = \lambda_{p,i}$, the first $\lambda_{p,i}$ means the i-th attribute name of ${}^a\mathcal{T}_p$, while the second one means that of ${}^n\mathcal{T}_p$. If $p \in [q+1, m]$, h_p means to align two vertices, which are from $\mathcal{G}_a$ and $\mathcal{G}_n$ respectively, if they have the same labels before cartesian product and have the same values in all the attributes. If $p \in [1, q]$, h_p has the similar meaning, except that the merged two edges need to have the same source and target vertices as well. Then, the join operator is proposed to represent this expression.

Definition 4.15 (join, $\bowtie$). Given two graph sets $\mathcal{G}_1 = (\widehat{G}_1, \mathcal{S})$ and $\mathcal{G}_2 = (\widehat{G}_2, \mathcal{S})$, $\mathcal{G}_1 \bowtie \mathcal{G}_2 = \zeta_{h_1}(\cdots(\zeta_{h_m}(\mathcal{G}_1 \times \mathcal{G}_2))\cdots)$.

Specifically, the join operator can be used to add some vertices and edges into each graph in a graph set. Please note that the join operator does not change the schema, and the schema of $\mathcal{G}_1 \bowtie \mathcal{G}_2$ is the same as that of $\mathcal{G}_1$.

Example 4.16. Given $\mathcal{G}_a$ and the updates $\mathcal{G}_{2019}$ as shown in Fig. 20, they have the same schema (as shown in Fig. 9) because they are both the graph sets of the ACM library. In order to apply the updates, the join operator is utilized, and the result is stored in $\mathcal{G}_{update}$. The schema of $\mathcal{G}_{update}$ is the same as $\mathcal{G}_a$. Note that the vertices that represent *"Jiawei Han"* in $\mathcal{G}_a$ and $\mathcal{G}_{2019}$ are merged. Moreover, the edges between *"Jiawei Han"* and *"Yu Zhang"* are not merged, because they have different values of attributes *"eid"* and *"times"*. □

4.2.2 Intersection. In Section 4.1.6, the difference operator is proposed, and it can be used to express many queries such as finding the researchers who coauthored papers with *"Jiawei Han"* in 2019 but not in 2018. Then, it is also an interesting problem which researchers have coauthored papers with *"Jiawei Han"* in both 2018 and 2019. Therefore, the intersection operator is proposed.

Definition 4.17 (Intersection, $\cap$). Given two graph sets $\mathcal{G}_1 = (\widehat{G}_1, \mathcal{S})$ and $\mathcal{G}_2 = (\widehat{G}_2, \mathcal{S})$, $\mathcal{G}_1 \cap \mathcal{G}_2 = (\widehat{G}_0, \mathcal{S})$, where $\widehat{G}_0 = \{G | G \in \widehat{G}_1 \wedge G \in \widehat{G}_2\}$.

The intersection operator can be expressed with the difference operator. Specifically, given two graph sets $\mathcal{G}_1$ and $\mathcal{G}_2$, the intersection components can be obtained by removing from $\mathcal{G}_1$ the graphs that are not in $\mathcal{G}_2$. In detail, we have:

$$\mathcal{G}_1 \cap \mathcal{G}_2 = \mathcal{G}_1 - (\mathcal{G}_1 - \mathcal{G}_2).$$

Example 4.18. We continue with Example 4.12, and now the researchers who have coauthored ACM papers with *"Jiawei Han"* in both 2018 and 2019 are wanted. Then, the intersection operator is utilized, and the results are shown in Fig. 21. □

4.2.3 Maximal. As stated in Section 4.1.7, it is useful to find out the maximal graphs in a graph set. Then, the maximal operator is defined as follows.

Definition 4.19 (Maximal, $\mathcal{M}$). Given $\mathcal{G} = (\widehat{G}, \mathcal{S})$, $\mathcal{M}(\mathcal{G}) = (\widehat{G}_0, \mathcal{S})$, where a graph $G \in \widehat{G}$ is in $\widehat{G}_0$ iff G is not a subgraph of any other graph in $\widehat{G}$ (i.e., G is maximal in $\widehat{G}$) .

Moreover, the maximal operator can be expressed with

$$\mathcal{M}(\mathcal{G}) = \mu(\mathcal{G} || \mathcal{G}' : \gamma((\mathcal{G}' \bowtie \mathcal{G}) \cap \mathcal{G}) \cap \mathcal{G}').$$

The key idea is that $\gamma((\mathcal{G}' \bowtie \mathcal{G}) \cap \mathcal{G}) \cap \mathcal{G}'$ returns $\mathcal{G}'$ if the graph in $\mathcal{G}'$ is maximal in $\mathcal{G}$. Otherwise, an empty set is returned. The process has been detailed in Example 4.14.

Example 4.20. As stated in Example 4.14, $\mathcal{M}(\mathcal{G}_{clique}) = \mu(\mathcal{G}_{clique} || \mathcal{G}' : \gamma((\mathcal{G}' \bowtie \mathcal{G}_{clique}) \cap \mathcal{G}_{clique}) \cap \mathcal{G}')$ is utilized to obtain the *mc*s in $\mathcal{G}_{clique}$ (shown in Fig. 17). Specifically, the map operator μ is applied to deal with each graph in $\mathcal{G}_{clique}$ respectively, and $\mathcal{G}'$ represents any graph set that contains one graph in $\mathcal{G}_{clique}$. Then, $\mathcal{G}' = (\widehat{G}' = \{G_{clique2}\}, \mathcal{S}_{clique})$ and $\mathcal{G}' = (\widehat{G}' = \{G_{clique5}\}, \mathcal{S}_{clique})$ are used as two instances to explain how to find out the *mc*s in $\mathcal{G}_{clique}$ with the expression $\mathcal{E}(\mathcal{G}')$.

As shown in Fig. 22a, when $\mathcal{G}' = (\widehat{G}' = \{G_{clique5}\}, \mathcal{S}_{clique})$, $\mathcal{G}' \bowtie \mathcal{G}_{clique}$ unites $G_{clique5}$ with each graph in $\mathcal{G}_{clique}$, and obtains $\mathcal{G}_{inter1}$. Then, the intersection operator is applied. For each graph $G \in \mathcal{G}_{inter1}$, G is reserved after the intersection between $\mathcal{G}_{inter1}$ and $\mathcal{G}_{clique}$ iff $G \in \mathcal{G}_{clique}$. It means that each reserved graph G should satisfy that $G_{clique5}$ is a subgraph of G, because $G \in \mathcal{G}_{clique}$ before and after the join operator. In detail, only $G_{clique5}$ is reserved, because $G_{clique5}$ is a subgraph of itself, while $G_{clique5}$ is not a subgraph of $G_{clique1}, \cdots, G_{clique4}$. Moreover, it is noted that if $G_{clique5}$ is maximal, there should be no graph $G \neq G_{clique5}$ satisfying that $G_{clique5}$ is a subgraph of G. Therefore, if $G_{clique5}$ is maximal, $\mathcal{G}_{inter2} = \mathcal{G}_{inter1} \cap \mathcal{G}_{clique}$ should contain $G_{clique5}$ only, and $\mathcal{G}_{inter3} = \gamma(\mathcal{G}_{inter2})$ should contain $G_{clique5}$ only. Then, G_{inter3} and $\mathcal{G}'$ both contain $G_{clique5}$, and $G_{inter3} \cap \mathcal{G}'$ returns a graph set containing $G_{clique5}$ only.

However, when $\mathcal{G}' = (\widehat{G}' = \{G_{clique2}\}, \mathcal{S}_{clique})$, because $G_{clique2}$ is a subgraph of $G_{clique5}$, both $G_{clique2}$ and $G_{clique5}$ are reserved in $\mathcal{G}_{inter2}$. Then, $\mathcal{G}_{inter3}$ only contains $G_{clique5}$, which is not in $\mathcal{G}'$. Therefore, the expression $\mathcal{E}(\mathcal{G}')$ returns an empty graph set. The process is shown in Fig. 22b.

After all the possible $\mathcal{G}'$ are traversed and the obtained graph sets are united, the generated graph set contains all the *mc*s in $\mathcal{G}_{clique}$. □

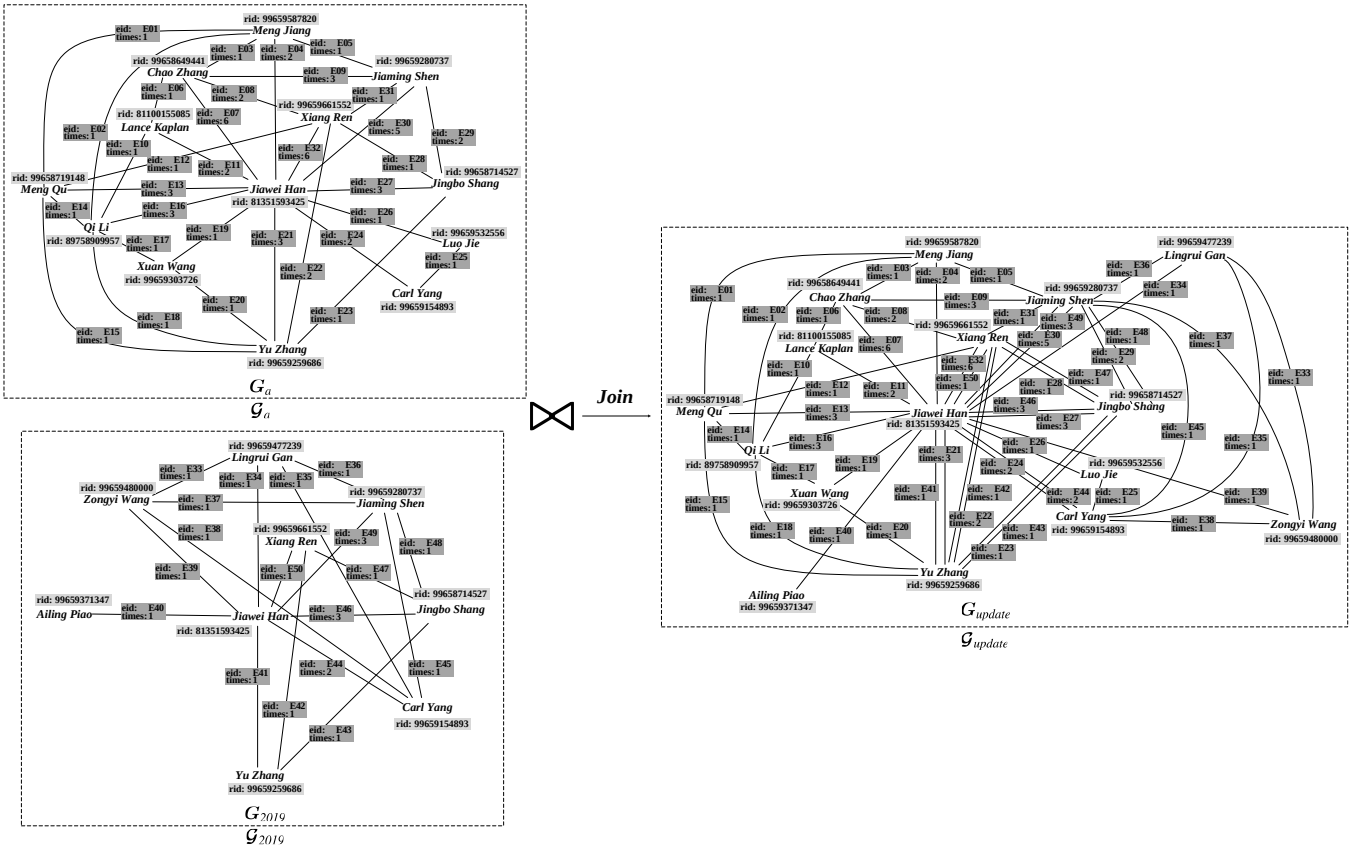


Figure 20: Updating the 1-hop-neighbor graph of *"Jiawei Han"* (i.e., $\mathcal{G}_a$) with the join operator.

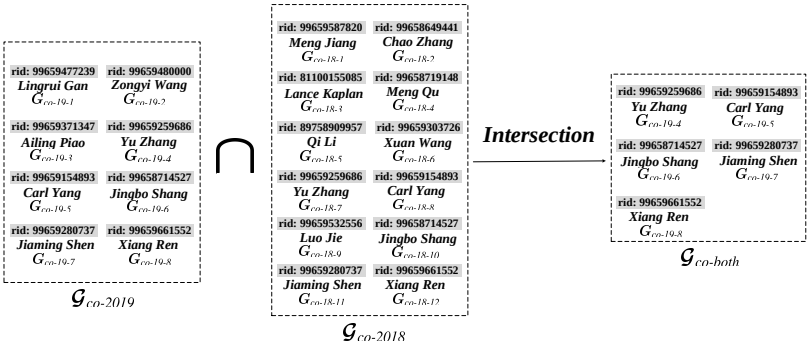
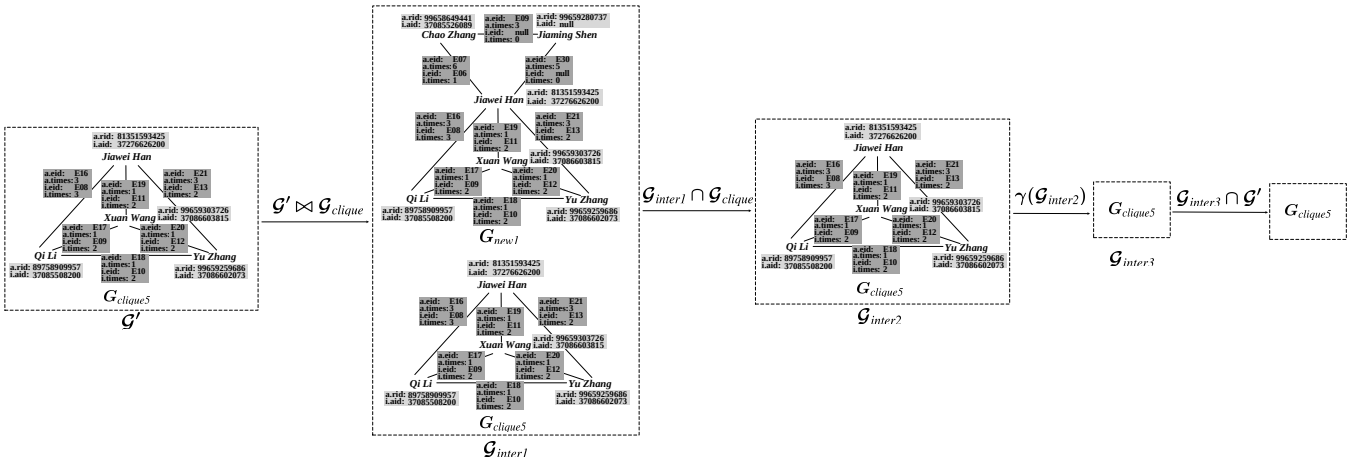


Figure 21: Finding researchers who have coauthored with *"Jiawei Han"* in both 2018 and 2019 with the intersection operator.

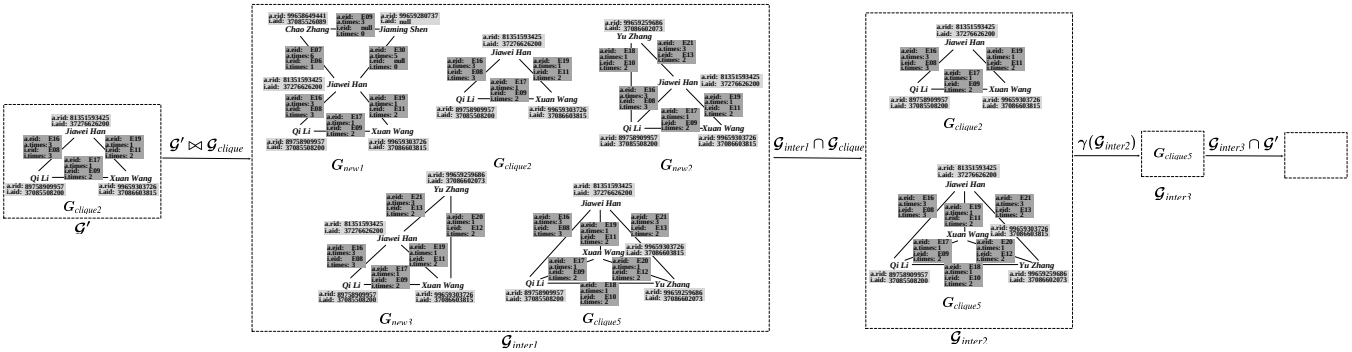
4.2.4 Union. Suppose the researchers who have coauthored ACM papers with *"Jiawei Han"* in 2018 or 2019 are wanted. Since the coauthors of *"Jiawei Han"* in 2018 and 2019 have been obtained and stored in $\mathcal{G}_{co\text{-}2018}$ and $\mathcal{G}_{co\text{-}2019}$ respectively as stated in Example 4.12, it is an intuitive idea to put the graphs in $\mathcal{G}_{co\text{-}2018}$ and $\mathcal{G}_{co\text{-}2019}$ into a new graph set $\mathcal{G}_{co\text{-}2018\text{-}2019}$, as the query result. Therefore, the union operator is proposed to unite such two graph sets.

Definition 4.21 (union, $\cup$). Given two graph sets $\mathcal{G}_1 = (\widehat{G}_1, \mathcal{S})$ and $\mathcal{G}_2 = (\widehat{G}_2, \mathcal{S})$, $\mathcal{G}_1 \cup \mathcal{G}_2 = \mathcal{G} = (\widehat{G} = \{G | G \in \widehat{G}_1 \vee G \in \widehat{G}_2\}, \mathcal{S})$.

The union operator can be expressed with the fundamental operators. The key idea is that given two graph sets $\mathcal{G}_1$ and $\mathcal{G}_2$, we first combine the graphs in these two graph sets with the product cartesian operator, and a new graph set (denoted as $\mathcal{G}_p$) is obtained. Then, new graphs with the vertices and edges from $\mathcal{G}_1$ and new graphs with the vertices and edges from $\mathcal{G}_2$ are extracted from the graphs in $\mathcal{G}_p$ with the selection operator respectively. The results form a graph set $\mathcal{G}_s$. Next, the maximal operator is applied to reserve only the maximal graphs in $\mathcal{G}_s$. Finally, the schema of the obtained graph set is changed to be that of $\mathcal{G}_1$ and $\mathcal{G}_2$ with the align operator.



(a) The process of evaluating whether $\mathcal{G}_{clique5}$ is an *mc*.



(b) The process of evaluating whether $\mathcal{G}_{clique2}$ is an *mc*.

Figure 22: The results of searching for *mc*s with the existing graph query languages (GraphQL, Gremlin, Cypher) and our graph query language *SOGQL*.

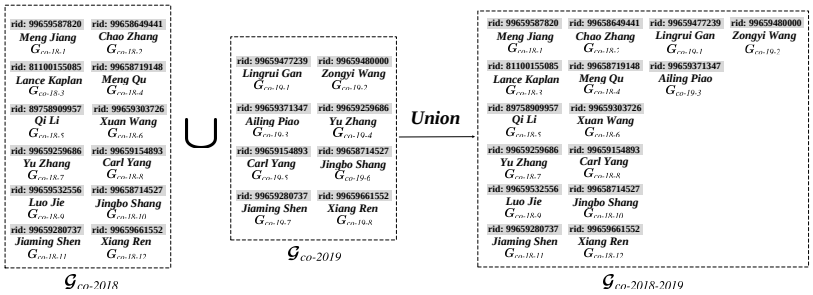


Figure 23: Finding the researchers that have coauthored with *"Jiawei Han"* in 2018 or 2019 with the union operator.

In detail, the expression is as follows:

$$\begin{aligned}&\mathcal{G}_1 \cup \mathcal{G}_2\\ &= \zeta_{h_1^{(1)}}(\cdots\zeta_{h_m^{(1)}}(\zeta_{h_1^{(2)}}(\cdots\zeta_{h_m^{(2)}}(\mathcal{M}(\sigma_{d_l}(\mathcal{G}_1\times\mathcal{G}_2)))\cdots))\cdots)\end{aligned}$$

where d_l = "$\forall x(x.label = {}^1\mathcal{T}_1 \vee \cdots \vee x.label = {}^1\mathcal{T}_m) \vee \forall x(x.label = {}^2\mathcal{T}_1 \vee \cdots \vee x.label = {}^2\mathcal{T}_m)$", and $\mathcal{T}_1, \cdots, \mathcal{T}_m$ are all the labels in $\mathcal{S}$. This selection operator finds the subgraphs of the graphs in $\mathcal{G}_1 \times \mathcal{G}_2$, and for each obtained subgraph, its vertices and edges are all from $\mathcal{G}_1$ or all from $\mathcal{G}_2$. Then, the maximal operator is applied to find the maximal ones in the obtained subgraphs, and the maximal subgraphs are the graphs in $\mathcal{G}_1$ and $\mathcal{G}_2$. At last, align operators are used to change the label names to be those in $\mathcal{S}$, and remove the prefixes "1." and "2.". Specifically, we have

$$\begin{aligned}h_k^{(1)} = {}^1\mathcal{T}_k, {}^1\mathcal{T}_k \Rightarrow \mathcal{T}_k \,||\\ \lambda_{k,1} = \lambda_{k,1}\ as\ \lambda_{k,1}, \cdots, \lambda_{k,n} = \lambda_{k,n}\ as\ \lambda_{k,n}\end{aligned}$$

$$\begin{aligned}h_k^{(2)} = {}^2\mathcal{T}_k, {}^2\mathcal{T}_k \Rightarrow \mathcal{T}_k \,||\\ \lambda_{k,1} = \lambda_{k,1}\ as\ \lambda_{k,1}, \cdots, \lambda_{k,n} = \lambda_{k,n}\ as\ \lambda_{k,n}\end{aligned}$$

where n is the number of attributes of $\mathcal{T}_k$, $\lambda_{k,1}, \cdots, \lambda_{k,n}$ are the attributes of $\mathcal{T}_k$, and $h_k^{(1)}$ and $h_k^{(2)}$ are used to remove the prefixes "1." and "2." respectively.

Example 4.22. As shown in Fig. 23, by applying the union operator on $\mathcal{G}_{co\text{-}2018}$ and $\mathcal{G}_{co\text{-}2019}$, the researchers who have coauthored ACM papers with *"Jiawei Han"* in 2018 or 2019 are stored in the graphs in $\mathcal{G}_{co\text{-}2018\text{-}2019}$. □

4.3 Examples of Graph Algebra

In this subsection, some examples are given to explain how to express queries with the proposed graph algebra. Please note that we focus on proving the sufficient expressive power of our graph algebra here. In the practical implementation, optimizations can be conducted to achieve better efficiency, but it is beyond the scope of this paper.

Example 4.23 (Maximal Cliques (mcs)). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$, all the *mc*s in the graphs in $\mathcal{G}$ can be obtained with the following expression:

$$\mathcal{G}_{mc} = \mathcal{M}(\sigma_{d_{clique}}(\mathcal{G})) \tag{1}$$

where d_{clique} = "$Clique()$". In detail, the selection operator is used to find all the cliques in $\mathcal{G}$, and the maximal operator is used to get the maximal ones from the obtained cliques. □

Example 4.24 (Mcs Containing a Given Vertex v). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$ and a vertex v, all the *mc*s containing v can be obtained with the following expression:

$$\mathcal{G}_{mcv} = \mathcal{M}(\sigma_{d_{cliquev}}(\mathcal{G})) \tag{2}$$

where $d_{cliquev}$ = "$Clique() \wedge \exists v'(v'.label = v.label \wedge v'.attr_1 = v.attr_1 \wedge \cdots \wedge v'.attr_k = v.attr_k)$", k is the number of attributes of v and v', and $attr_1, \cdots, attr_k$ are the k attributes of v and v'. □

Example 4.25 (Connected Components). Given a graph set $\mathcal{G} = (\widehat{G}, \mathcal{S})$ of simple graphs, all the connected components of the graphs in $\mathcal{G}$ can be obtained with the following expressions:

$$\mathcal{G}_{sub} = \sigma_{\text{True}}(\mathcal{G}) \tag{3}$$

$$\begin{aligned}\mathcal{G}_{edge} = \sigma_{d_e}(\mathcal{G}), \text{where } d_e = \text{"}&\exists v_1, v_2 (Vertex(v_1, v_2) \\ &\wedge Unique(v_1, v_2) \wedge Complete(v_1, v_2)) \\ &\wedge \neg \exists v_1, v_2, e_1, e_2, x (Unique(v_1, v_2, e_1, e_2, x))\text{"}\end{aligned} \tag{4}$$

$$\begin{aligned}\mathcal{G}_c = \mu(\mathcal{G}_{sub} || \mathcal{G}' : \gamma((\sigma_{d_v}(\mathcal{G}') \bowtie \mathcal{G}_{edge}) \cap \mathcal{G}_{edge}) \cap \mathcal{G}') \\ \text{where } d_v = \text{"}\exists x (Vertex(x)) \wedge \neg \exists x, y (Unique(x, y))\text{"}\end{aligned} \tag{5}$$

$$\mathcal{G}_{cc} = \mu(\mathcal{G}_c || \mathcal{G}' : \mathcal{G}' - \mathcal{G}' \bowtie (\mathcal{G}_c - \mathcal{G}')) \tag{6}$$

The basic idea of searching for connected components in $\mathcal{G}$ with our graph algebra is that for a given connected component G_c in G, all the edges adjacent to vertices in G_c are in G_c. Specifically, given a graph G, if a subgraph of G named G_c satisfies that all the edges adjacent to vertices in G_c are in G_c, G_c is called a maximal subgraph (abbreviated as an MSG), and a connected component in G is an MSG. Please note that an MSG may consist of several connected components and be unconnected. Therefore, we first find all the MSGs of the graphs in $\mathcal{G}$ with Eqs. 3–5, and then remove the MSGs that are not connected.

In detail, we first obtain all the subgraphs of the graphs in $\mathcal{G}$ with Eq. 3. Please note that all the subgraphs can be obtained because they all satisfy the constaint. Then, a graph set of the edges in $\mathcal{G}$ is computed and denoted as $\mathcal{G}_{edge}$ (Eq. 4), and each graph in $\mathcal{G}_{edge}$ consists of an edge in $\mathcal{G}$. Next, each graph G' in $\mathcal{G}_{sub}$ (i.e., each subgraph of graphs in $\mathcal{G}$) is put into $\mathcal{G}'$ and distinguished whether it is an MSG.

Specifically, for each graph set $\mathcal{G}' = (\widehat{G}' = \{G'\}, \mathcal{S})$ that contains a graph G' in $\mathcal{G}_{sub}$, the selection operator is applied to extract the vertices in G' first (i.e., $\sigma_{d_v}(\mathcal{G}')$). Then, the edges adjacent to vertices in G' are obtained with the join and interection operators. Specifically, the join operator combines each verex v in G' with an edge e in $\mathcal{G}_{edge}$, and if e is adjacent to v, the result of this combination is still e because v is merged with an adjacent vertex of e, and e is reserved in the intersection operator. Otherwise, if e is not adjacent to v, the result of the combination is an edge and a vertex, which is no longer an edge in $\mathcal{G}_{edge}$, and is filtered out in the intersection operator. Therefore, with $(\sigma_{d_v}(\mathcal{G}') \bowtie \mathcal{G}_{edge}) \cap \mathcal{G}_{edge}$, all the edges adjacent to a vertex in G' is obtained, and the reduce operator is used to put these edges and their adjacent vertices into a graph. If G' is an MSG, the obtained graph should be the same as G', and the intersection operator applied on $\gamma((\sigma_{d_v}(\mathcal{G}') \bowtie \mathcal{G}_{edge}) \cap \mathcal{G}_{edge})$ and $\mathcal{G}'$ should return a graph set consisting of G', and then G' is put into $\mathcal{G}_c$. Otherwise, if G' is not an MSG, the obtained graph is different from G', and the intersection operator returns an empty graph set. Hence, $\mathcal{G}_c$ contains all the MSGs in $\mathcal{G}$.

Moreover, an MSG can be disconnected if it consists of at least two smaller MSGs. Therefore, the connected components should be the minimal graphs in $\mathcal{G}_c$, i.e., an MSG $G \in \mathcal{G}_c$ is a connected component iff there is not any other MSG $G' \in \mathcal{G}_c$ satisfying that G' is a subgraph of G. Eq. 6 is utilized to removed the MSGs that are not minimal. In detail, given $\mathcal{G}'$ that contains a graph G' in $\mathcal{G}_c$, $G' \notin (\mathcal{G}' \bowtie (\mathcal{G}_c - \mathcal{G}'))$ iff G' is minimal. Then, $G' \in \mathcal{G}' - \mathcal{G}' \bowtie (\mathcal{G}_c - \mathcal{G}')$ iff G' is minimal. Therefore, G' is put into $\mathcal{G}_{cc}$ iff G' is minimal, and Eq. 6 can return a graph set of all the connected components in $\mathcal{G}$. □

5 GRAPH QUERY LANGUAGE *SOGQL*

The grammar of declarative graph query language *SOGQL* is designed based on the graph data model. It consists of a collection of important statements for graph data, and the detailed grammar is shown in Appendix A. In this section, some cases of querying with *SOGQL* are presented.

Example 5.1. Given a graph set *GS* that contains the graph shown in Fig. 1, suppose we want to search for all the *mc*s in *GS*. Formally, we have

GS = {({v_1[Researcher], v_2[Researcher], $\cdots$, v_{12}[Researcher]},
{$e_1 = (v_1, v_7)$[Coauthor], $e_2 = (v_2, v_3)$[Coauthor],
$e_3 = (v_2, v_7)$[Coauthor], $e_4 = (v_3, v_2)$[Coauthor],
$e_5 = (v_3, v_7)$[Coauthor], $\cdots$,
$e_{29} = (v_{12}, v_{10})$[Coauthor], $e_{30} = (v_{12}, v_7)$[Coauthor]},
({v_1<acmid="99659587820", ieeeid="37088268583",
name="Meng Jiang">, $\cdots$,
v_7<acmid="81351593425", ieeeid="37276626200",
name="Jiawei Han">, $\cdots$,
v_{12}<acmid="99659259686", ieeeid="37086602073",
name="Yu Zhang">},
{e_1<eid="E01", times="4">, e_2<eid="E15", times="3">,
e_3<eid="E02", times="7">, e_4<eid="E15", times="3">,
e_5<eid="E13", times="5">, $\cdots$,
e_{29}<eid="E10", times="3">, e_{30}<eid="E07", times="5">})) }.

In the statement given below, *GS* is used instead of listing the complete form of the graph set for clarity. The corresponding statement for the *mc* query is as follows:

```
select maximal * from GS where Clique();
```

Note that the second selection operator is utilized to find all the cliques in GS, and the maximal operator is applied to obtain the *mc*s . The query results are shown in Fig. 2b.

Example 5.2. Suppose we want to find all the connected components in graph of the ACM library. Then, the following query statements can be utilized to achieve this goal:

```
select as GSsub * from GS where True;
select as GSedge * from GS where Exist v1, v2(Vertex(v1, v2) and
Unique(v1, v2) and Complete(v1, v2)) and not Exist v1, v2, e1, e2, x(Unique(v1,
v2, e1, e2, x));
select as GSc * from GSsub map [
   (select reduce * from
      ((select * from
         (select * from GS' where Exist x(Vertex(x)) and not Exist x,
y(Unique(x, y))) join GSedge
      ) intersect GSedge)
   ) intersect GS'
] by GS';
select as GScc * from GSc map [
   GS' - select * from GS' join (GSc - GS')
] by GS';
```

These four statements correspond to the four equations in Example 4.25, and the connected components of the graphs in GS are stored in graph set GScc. Please note that these four expressions can also be merged to be one *SOGQL* expression. Therefore, connected components can be obtained with one *SOGQL* query. □

6 EXPRESSIVE POWER

In this section, we prove that our proposed graph algebra and graph calculus have the same expressive power. That is to say, for each expression of graph algebra, there exists an expression of graph calculus with the same meaning and vice versa. Since the graph calculus is based on MSOL, it suggests that *SOGQL* has the expressive power of MSOL. In the proofs, if not specified, the graph set, on which the queries are performed, is denoted as $\mathcal{G}_0$, i.e., the expressions of graph algebra and graph calculus are used to construct a new graph set based on the graphs in $\mathcal{G}_0$.

6.1 Graph Algebra ⊂ Graph Calculus

First, we prove that each graph algebra expression has a corresponding graph calculus expression equal to it, and the following theorem is presented.

THEOREM 6.1. *If $\mathcal{E}_a$ is a graph algebra expression, there exists a graph calculus expression, denoted as $\mathcal{E}_c$, equal to $\mathcal{E}_a$.*

PROOF. Induction on the number of operators in $\mathcal{E}_a$ is used. Note that operators in the map operation (μ) are also counted.

Suppose Theorem 6.1 is correct when the number of the operators in $\mathcal{E}_a$ is no more than N. When there are $N+1$ operators in $\mathcal{E}_a$, let $\mathcal{E}_a = \mathcal{E}_1 \beta \mathcal{E}_2$, where $\beta \in \{\times, -\}$; or $\mathcal{E}_a = \beta(\mathcal{E}_1)$, where $\beta \in \{\rho_s, \sigma_d, \gamma, \zeta_h\}$; or $\mathcal{E}_a = \beta(\mathcal{E}_1 || \mathcal{G}' : \mathcal{E}_2)$, where $\beta \in \{\mu\}$. As there are no more than N operators in $\mathcal{E}_1$ and $\mathcal{E}_2$, there exist two graph calculus expressions equal to $\mathcal{E}_1$ and $\mathcal{E}_2$ in the above expressions, respectively. Specifically, the graph calculus expression equal to $\mathcal{E}_1$ is denoted by $\{G|\Phi_1(G)\}$, and that equal to $\mathcal{E}_2$ is denoted by $\{G|\Phi_2(G)\}$, where G represents a graph and is a set of elements. To prove the correctness of Theorem 6.1, let β be the seven different operators separately. Note that when $\beta = \mu$, expression $\mathcal{E}_2$ queries on graph set $\mathcal{G}'$ rather than $\mathcal{G}_0$, and $\mathcal{G}'$ consists of a graph in the graph set obtained by $\mathcal{E}_1$.

Condition 1: Suppose $\mathcal{E}_a = \mathcal{E}_1 \times \mathcal{E}_2$, and then we have $\{G|\exists G_1(\exists G_2(\Phi_1(G_1) \wedge \Phi_2(G_2) \wedge \forall t(t \in G \rightarrow \exists u(((u \in G_1 \wedge t[label] = {}^{F(1)}u[label]) \vee (u \in G_2 \wedge t[label] = {}^{F(2)}u[label])) \wedge t[2] = u[2] \wedge \cdots \wedge t[k] = u[k])) \wedge \forall u(u \in G_1 \rightarrow \exists t(t \in G \wedge {}^{F(1)}u[label] = t[label] \wedge u[2] = t[2] \wedge \cdots \wedge u[k] = t[k])) \wedge \forall u(u \in G_2 \rightarrow \exists t(t \in G \wedge {}^{F(2)}u[label] = t[label] \wedge u[2] = t[2] \wedge \cdots \wedge u[k] = t[k])))))\}$ equal to $\mathcal{E}_a$, where k represents the number of components in t. It is assumed that the graph set that contains G_i is $\mathcal{G}_{F(i)}$.

Condition 2: Suppose $\mathcal{E}_a = \rho_s(\mathcal{E}_1)$, and denote by s' the schema of the graph set obtained by $\mathcal{E}_1$, s is a new schema contained by s'. Let $\mathcal{T}$ be the list of all the labels in s, and $\mathcal{A}_{\mathcal{T}_i,j}$ be the j-th attribute of label $\mathcal{T}_i$. Then, the attribute names of label $\mathcal{T}_i$ in s are denoted by $\mathcal{A}_{\mathcal{T}_i,1}, \cdots, \mathcal{A}_{\mathcal{T}_i,m_{\mathcal{T}_i}}$, where $m_{\mathcal{T}_i}$ is the number of $\mathcal{T}_i$'s attributes in s. Without loss of generality, suppose that for each label $\mathcal{T}_i$, the reserved attributes are its first $m_{\mathcal{T}_i}$ attributes in s'. Denote by $\widehat{m}_t$ the number of components in element t. Then, the expression $\{G|\exists G_1(\Phi_1(G_1) \wedge \forall t(t \in G \rightarrow (t[label] \in \mathcal{T} \wedge \exists t_1(t_1 \in G_1 \wedge t_1[1] = t[1] \wedge \cdots \wedge t_1[\widehat{m}_t] = t[\widehat{m}_t]))) \wedge \forall t_1((t_1 \in G_1 \wedge t_1[label] \in \mathcal{T}) \rightarrow \exists t(t \in G \wedge t[1] = t_1[1] \wedge \cdots \wedge t[\widehat{m}_t] = t_1[\widehat{m}_t]))))\}$ equals $\mathcal{E}_a$.

Condition 3: Suppose $\mathcal{E}_a = \sigma_d(\mathcal{E}_1)$, and d is the selection constraint. Then, $\{G|\exists G_1(\Phi_1(G_1) \wedge Subgraph(G, G_1) \wedge d')\}$ equals $\mathcal{E}_a$. Specifically, d is an FOL expression, and d' is an MSOL expression obtained from d by stating that every bound variable belongs to G. For example, if $d = \exists x(Vertex(x))$, it is obtained that $d' = \exists x(x \in G \wedge Vertex(x))$.

Condition 4: Suppose $\mathcal{E}_a = \gamma(\mathcal{E}_1)$, $\{G|\forall G_1(\Phi_1(G_1) \rightarrow \forall t_1(t_1 \in G_1 \rightarrow t_1 \in G)) \wedge \forall t(t \in G \rightarrow \exists G_1(\Phi_1(G_1) \wedge t \in G_1))\}$ equals $\mathcal{E}_a$.

Condition 5: Suppose $\mathcal{E}_a = \zeta_h(\mathcal{E}_1)$, where $h = l_1, l_2 \Rightarrow l_0 || attr_{i_1} = attr_{j_1}$ *as* $a_1 \wedge \cdots \wedge attr_{i_k} = attr_{j_k}$ as a_k. Without loss of generality, let $attr_{i_1}, \cdots, attr_{i_k}$ be the first k attributes of l_1, and let $attr_{j_1}, \cdots, attr_{j_k}$ be the first k attributes of l_2. Then, $\{G|\exists G_1(\Phi_1(G_1) \wedge \forall t((t \in G \wedge t[label] = l_0) \rightarrow (P \vee (\neg P \wedge Q))) \wedge \forall t((t \in G \wedge t[label] \neq l_0 \rightarrow (t \in G_1)) \wedge \forall u((u \in G_1 \wedge u[label] \neq l_1 \wedge u[label] \neq l_2) \rightarrow u \in G) \wedge \forall u((u \in G_1 \wedge u[label] = l_1) \rightarrow (\exists t(t \in G \wedge t[label] = l_0 \wedge t[2] = u[2] \wedge \cdots \wedge t[m] = u[m]))) \wedge \forall u((u \in G_1 \wedge u[label] = l_2) \rightarrow (\exists t(t \in G \wedge t[label] = l_0 \wedge ((Vertex(t) \wedge t[2] = u[2] \wedge \cdots \wedge t[k+1] = u[k+1] \wedge t[m+1] = u[k+2] \wedge \cdots \wedge t[m+n-1-k] = u[n]) \vee (Edge(t) \wedge t[2] = u[2] \wedge \cdots \wedge t[k+3] = u[k+3] \wedge t[m+1] = u[k+4] \wedge \cdots \wedge t[m+n-3-k] = u[n])))))))\}$ is equal to $\mathcal{E}_a$, where m and n represent the number of components in the elements labeled l_1 and l_2 respectively.

Specifically, $P = (\exists u(\exists v(u \in G_1 \wedge v \in G_1 \wedge u[label] = l_1 \wedge v[label] = l_2 \wedge ((Vertex(t,u,v) \wedge t[2] = u[2] \wedge \cdots \wedge t[m] = u[m] \wedge t[m+1] = v[k+2] \wedge \cdots \wedge t[m+n-1-k] = v[n] \wedge u[2] = v[2] \wedge \cdots \wedge u[k+1] = v[k+1] \wedge \neg\exists p(Vertex(p) \wedge p \in G_1 \wedge Unique(u,v,p) \wedge p[1] = u[1] \wedge \cdots \wedge p[k+1] = u[k+1])) \vee (Edge(t,u,v) \wedge t[src] = u[src] \wedge t[tgt] = u[tgt] \wedge t[src] = v[src] \wedge t[tgt] = v[tgt] \wedge t[4] = u[4] \wedge \cdots \wedge t[m] = u[m] \wedge t[m+1] = v[k+4] \wedge \cdots \wedge t[m+n-3-k] = v[n] \wedge u[4] = v[4] \wedge \cdots \wedge u[k+3] = v[k+3] \wedge \neg\exists p(Edge(p) \wedge p \in G_1 \wedge Unique(u,v,p) \wedge p[1] = u[1] \wedge u[src] = p[src] \wedge u[tgt] =$

$p[tgt] \wedge p[4] = u[4] \wedge \cdots \wedge p[k+3] = u[k+3])))))$, which means that t is obtained by aligning two elements. Besides, $Q = (\exists u(u \in G_1 \wedge (((Vertex(t) \wedge u[label] = l_1 \wedge t[2] = u[2] \wedge \cdots \wedge t[m] = u[m] \wedge t[m+1] = NULL \wedge \cdots \wedge t[m+n-1-k] = NULL) \vee (Edge(t) \wedge u[label] = l_1 \wedge t[2] = u[2] \wedge \cdots \wedge t[m] = u[m] \wedge t[m+1] = NULL \wedge \cdots \wedge t[m+n-3-k] = NULL)) \vee ((Vertex(t) \wedge u[label] = l_2 \wedge t[2] = u[2] \wedge \cdots \wedge t[k+1] = u[k+1] \wedge t[k+2] = NULL \wedge \cdots \wedge t[m] = NULL \wedge t[m+1] = u[k+2] \wedge \cdots \wedge t[m+n-1-k] = u[n]) \vee (Edge(t) \wedge u[label] = l_2 \wedge t[2] = u[2] \wedge \cdots \wedge t[k+3] = u[k+3] \wedge t[k+4] = NULL \wedge \cdots \wedge t[m] = NULL \wedge t[m+1] = u[k+4] \wedge \cdots \wedge t[m+n-3-k] = u[n])))))$, which means that t is obtained by adding more attributes to an element labeled l_1 or l_2 in G_1.

Condition 6: Suppose $\mathcal{E}_a = \mathcal{E}_1 - \mathcal{E}_2$, and then $\{G|\Phi_1(G) \wedge \neg\Phi_2(G)\}$ is equal to $\mathcal{E}_a$.

Condition 7: Suppose $\mathcal{E}_a = \mu(\mathcal{E}_1||\mathcal{G}' : \mathcal{E}_2)$, and then the number of operators in $\mathcal{E}_2$ is no more than N. Denote the graph calculus expression equal to $\mathcal{E}_2$ by $\{G|\Phi_2(G)\}$. Then, $\{G|\exists G'(\Phi_1(G') \wedge \Phi_2(G))\}$ equals $\mathcal{E}_a$, where G' can appear in Φ_2.

Therefore, when graph algebra expressions with N operators have graph calculus expressions equal to them, for the graph algebra expressions with $N+1$ operators, they also have corresponding graph calculus expressions equal to them.

Then, the graph algebra expressions with no operator are also considered. When there is no operator in $\mathcal{E}_a$, if $\mathcal{E}_a$ is not in the $\mathcal{E}_2$ part of $\mu(\mathcal{E}_1||\mathcal{G}' : \mathcal{E}_2)$, $\mathcal{E}_a$ should be of the form $(\{G_1, G_2, \cdots, G_k\}, \mathcal{S})$ that represents a constant graph set, where each G_i is a graph consisting of elements. Then, $\mathcal{E}_c = \{G|\Phi(G)\}$ is equal to $\mathcal{E}_a$, where $\Phi(G) = GraphEqual(G, G_1) \vee \cdots \vee GraphEqual(G, G_k)$. Otherwise, suppose $\mathcal{E}_a$ is in the $\mathcal{E}_2$ part of μ. If $\mathcal{E}_a$ is in the form of $(\{G_1, G_2, \cdots, G_k\}, \mathcal{S})$, $\mathcal{E}_c = \{G|GraphEqual(G, G_1) \vee \cdots \vee GraphEqual(G, G_k)\}$ equals $\mathcal{E}_a$. Otherwise, if $\mathcal{E}_a = \mathcal{G}'$, as stated in Condition 7, $\{G|\exists G'(\Phi_1(G') \wedge \Phi_2(G))\}$ is equal to $\mu(\mathcal{E}_1||\mathcal{G}' : \mathcal{E}_2)$, and then $\{G|GraphEqual(G, G')\}$ is equal to $\mathcal{E}_a$.

In conclusion, every graph algebra expression has a graph calculus expression equal to it, i.e., Theorem 6.1 is correct. □

6.2 Graph Calculus ⊂ Graph Algebra

Then, it is to be proved that for each graph calculus expression, there is a graph algebra expression equal to it. First, some concepts used in the proof are proposed. Please note that the subformula of a graph calculus expression $\mathcal{E}_c$ means the subformula of the formula in $\mathcal{E}_c$.

Definition 6.2 (Context). Given a graph calculus expression $\{G|\Phi(G)\}$ and a subformula of it (denoted as $\widehat{\Phi}(G)$), the context of the subformula, which is denoted as $Context(\widehat{\Phi}(G))$, consists of the free variables (e.g., G) and the related bound variables satisfying that $\widehat{\Phi}(G)$ is within their scopes.

Definition 6.3 (Unconstrained Context). Given a graph calculus expression $\{G|\Phi(G)\}$ and a subformula of it (denoted as $\widehat{\Phi}(G)$), the unconstrained context of $\widehat{\Phi}(G)$ is the variables in $Context(\widehat{\Phi}(G))$ that do not appear in $\widehat{\Phi}(G)$. The unconstrained context of $\widehat{\Phi}(G)$ is denoted by $UC(\widehat{\Phi}(G))$.

Example 6.4. Given a graph calculus expression, $\mathcal{E}_c = \{G|\exists G'(G' \in \mathcal{G}_0 \wedge \exists t(t \in G' \wedge t \in G))\}$, where $\mathcal{G}_0$ is a graph set that contains graphs $\{G_1, G_2, \cdots, G_k\}$. Then, $\widehat{\Phi}(G) = t \in G$ is a subformula of $\mathcal{E}_c$, and its context is $Context(\widehat{\Phi}(G)) = \{G, G', t\}$. G is the free variable representing the graph to obtain, G' and t are the related bound variables of $\widehat{\Phi}(G)$, and $\widehat{\Phi}(G)$ is within their scopes. Among the context of $\widehat{\Phi}(G)$, only G' does not appear in $\widehat{\Phi}(G)$. Therefore, $UC(\widehat{\Phi}(G)) = \{G'\}$. Moreover, let $\widehat{\Phi}'(G)$ be $G' \in \mathcal{G}_0 \wedge \exists t(t \in G' \wedge t \in G)$, and then $Context(\widehat{\Phi}'(G)) = \{G, G'\}$. t is not in $Context(\widehat{\Phi}'(G))$ because $\widehat{\Phi}'(G)$ is beyond the scope of t. □

Definition 6.5 (Domain). The domain of a variable v is the value that v can have, and it is denoted as $D(v)$. Given a subformula $\widehat{\Phi}(G)$ and its context $Context(\widehat{\Phi}(G)) = \{var_1, var_2, \cdots, var_k\}$, the domain of $Context(\widehat{\Phi}(G))$ is defined as $D(var_1) \times D(var_2) \times \cdots \times D(var_k)$.

In an expression of graph calculus, there should be at least one constant graph set, which represents the graph set that queries are applied on. Given an expression of graph calculus denoted as $\mathcal{E}_c = \{G|\Phi(G)\}$, suppose there are k constant graph sets in $\mathcal{E}_c$ (i.e., $\mathcal{G}_1, \cdots, \mathcal{G}_k$), and then the domains of the variables in the unconstrained context of a subformula $\widehat{\Phi}(G)$ of $\mathcal{E}_c$ are as follows. Given a variable (denoted as t) in the unconstrained context of $\widehat{\Phi}(G)$, firstly, if t is a vertex variable, then $D(t) = \sigma_{d_v}(\mathcal{G}_\cup)$ where

$$\mathcal{G}_\cup = \zeta_{h_1}(\cdots(\zeta_{h_m}(\gamma(\mathcal{G}_1) \times \cdots \times \gamma(\mathcal{G}_k)))\cdots).$$

Specifically, $h_1, \cdots, h_m$ are m alignment conditions utilized to align vertices and edges, rename the label names, and rename the attribute names according to the query.

$$d_v = \exists v(Vertex(v)) \wedge \neg\exists u, v(Unique(u, v))$$

is used to find subgraphs of the graphs in $\mathcal{G}_\cup$, and each of the subgraphs contains only a vertex. Secondly, if the variable is an edge variable (denoted as t), then $D(t) = \sigma_{d_e}(\mathcal{G}_\cup)$, where

$$d_e = \exists e(Edge(e)) \wedge \neg\exists e, u, v, x(Unique(e, u, v, x))$$

is used to find subgraphs with only an edge and its adjacent vertices. Note that u and v represent the adjacent vertices of e. Thirdly, if the variable represents a graph (e.g., G and G' in $\{G|\exists G'(GraphEqual(G, G'))\}$), $D(G) = \sigma_{\text{True}}(\mathcal{G}_\cup)$.

Example 6.6. Let us continue with Example 6.4. As there is only one constant graph set in the formula, it is satisfied that $\mathcal{G}_\cup = \gamma(\mathcal{G}_0)$. When $\widehat{\Phi}(G) = t \in G$ is dealt with, because $G' \in UC(\widehat{\Phi}(G))$, $D(G') = \sigma_{\text{True}}(\mathcal{G}_\cup)$, and the domain of $UC(\widehat{\Phi}(G))$ is $D(G')$. Moreover, when we focus on $\widehat{\Phi}'(G) = t \in G'$, $G \in UC(\widehat{\Phi}(G))$ and G represents the graph to obtain, $D(G) = \sigma_{\text{True}}(\mathcal{G}_\cup)$. □

THEOREM 6.7. *If $\mathcal{E}_c$ is a graph calculus expression, there exists a graph algebra expression, denoted as $\mathcal{E}_a$, equal to it.*

PROOF. $\wedge, \vee, \neg, \exists$ and $\forall$ are the operators that can appear in an expression of graph calculus. Because $\Phi_1 \wedge \Phi_2$ can be replaced with $\neg(\neg\Phi_1 \vee \neg\Phi_2)$, and $\forall u(\Phi_1(u))$ can be replaced with $\neg\exists u(\neg\Phi_1(u))$, only the operators of $\vee$, $\neg$ and $\exists$ are discussed here. Please note that the process of renaming the label names and attribute names with the align operator is sometimes omitted in this proof for simplicity.

Let $\widehat{\Phi}(G)$ be a subformula of $\mathcal{E}_c$, and $\widehat{\Phi}(G)$ is said to be equal to a graph algebra expression $\mathcal{E}'_a$, if the domain of $Context(\widehat{\Phi}(G))$ is the same as the graphs in the graph set obtained by $\mathcal{E}'_a$. Specifically,

when $\mathcal{E}_a = \mathcal{E}'_a$, $\mathcal{E}_c = \{G|\widehat{\Phi}(G)\}$, and $\mathcal{E}_a$ equals $\widehat{\Phi}(G)$, $\mathcal{E}_a$ is the graph algebra expression equal to $\mathcal{E}_c$. Therefore, it is to be proved that each subformula $\widehat{\Phi}(G)$ has a graph algebra expression equal to it, and induction on the number of operators in a subformula of $\mathcal{E}_c$ is used. When there are other free variables in $\mathcal{E}_c$ besides G, we can use $\rho_{\mathcal{S}_G}(\mathcal{E}_a)$ to obtain the target graphs, where $\mathcal{S}_G$ is the schema of the graph to obtain. For simplicity, in the following proofs, it is assumed that the only free variable in $\mathcal{E}_c$ is the graph to obtain (i.e., G).

If $\widehat{\Phi}(G)$ has no operator, it should be an atom formula, i.e., $\widehat{\Phi}(G)$ is in the form of $T(p)$, $p[i]\theta q[j]$, $p[i]\theta c$, $c\theta p[i]$, $p \in G$, $p_c \in G$, or $p \in G_c$, where p, q are element variables, and p_c is a constant element that represents a stable vertex or a stable edge. The corresponding graph algebra expressions of $\widehat{\Phi}(G)$ with no operator are discussed as follows:

$T(p)$: T can be "*Vertex*" or "*Edge*", meaning that p represents a vertex or an edge, respectively. $\sigma_{d'}(\mathcal{G}_\cup) \times D(UC(\widehat{\Phi}(G))$ is equal to it, where d' is the selection constraint generated according to T. In detail, when T = "Vertex", $d' = d_v$, and when T = "Edge", $d' = d_e$.

$p[i]\theta q[j]$: $\sigma_d(\mathcal{G}_\cup) \times D(UC(\widehat{\Phi}(G))$ equals it. In detail, if p is a vertex variable and q is an edge variable, then, let $d = \exists p, q(Vertex(p) \wedge Edge(q) \wedge p[i] = q[j]) \wedge \exists v_1, v_2, v_3(Vertex(v_1, v_2, v_3) \wedge Unique(v_1, v_2, v_3) \wedge \neg\exists v_4(Vertex(v_4) \wedge Unique(v_1, v_2, v_3, v_4))) \wedge \neg\exists e_1, e_2(Edge(e_1, e_2) \wedge Unique(e_1, e_2))$. For other conditions such as both p and q being vertex variables, the corresponding d can be generated similarly.

$p[i]\theta c, c\theta p[i]$: $\sigma_d(\mathcal{G}_\cup) \times D(UC(\widehat{\Phi}(G))$ is equal to $\widehat{\Phi}(G)$, where $d = \exists p(Vertex(p) \wedge p[i] = c) \wedge \neg\exists p, x(Unique(p, x))$ if p is a vertex variable, and $d = \exists p(Edge(p) \wedge p[i] = c) \wedge \neg\exists p, u, v, x(Unique(p, u, v, x))$ if p is an edge variable.

$p \in G$, $p_c \in G$, or $p \in G_c$: In this subformula, p is an element variable, p_c is a constant element , G is a graph variable, and G_c is a constant graph.

If $\widehat{\Phi}(G) = p_c \in G$, $\sigma_{\exists t(Equal(p_c,t))}(\mathcal{G}_\cup) \times D(UC(\widehat{\Phi}(G)))$ is equal to $\widehat{\Phi}(G)$; else if $\widehat{\Phi}(G) = p \in G$, $\mu(\sigma_{\text{True}}(\mathcal{G}_\cup)||\mathcal{G}' : \mathcal{G}' \times \sigma_{d_v}(\mathcal{G}')) \times D(UC(\widehat{\Phi}(G)))$ equals $\widehat{\Phi}(G)$ when p represents a vertex, and $\mu(\sigma_{\text{True}}(\mathcal{G}_\cup)||\mathcal{G}' : \mathcal{G}' \times \sigma_{d_e}(\mathcal{G}')) \times D(UC(\widehat{\Phi}(G)))$ equals $\widehat{\Phi}(G)$ when p is an edge. Otherwise, if $\widehat{\Phi}(G) = p \in G_c$, $\sigma_{d_v}(\mathcal{G}_c) \times D(UC(\widehat{\Phi}(G)))$ equals $\widehat{\Phi}(G)$ when p represents a vertex, and $\sigma_{d_e}(\mathcal{G}_c) \times D(UC(\widehat{\Phi}(G)))$ equals $\widehat{\Phi}(G)$ when p is an edge. Note that $\mathcal{G}_c$ is a graph set that only contains graph G_c.

Suppose a subformula $\widehat{\Phi}(G)$ of $\mathcal{E}_c$ with no more than N operators has a graph algebra expression equal to it, and now there is a subformula of $\mathcal{E}_c$ denoted as $\Phi(G)$ with $N + 1$ operators. There are totally three possible methods to partition $\Phi(G)$, i.e., $\Phi(G) = \Phi_1(G) \vee \Phi_2(G)$, $\Phi(G) = \neg\Phi_1(G)$ and $\Phi(G) = \exists p(\Phi_1(G))$. Because $\Phi_1(G)$ and $\Phi_2(G)$ both have fewer than $N + 1$ operators, they both have graph algebra expressions equal to them according to our presumption.

Condition 1: Suppose $\Phi(G) = \Phi_1(G) \vee \Phi_2(G)$. Let $\mathcal{E}_1$ and $\mathcal{E}_2$ be the expressions of graph algebra equal to $\{G|\Phi_1(G)\}$ and $\{G|\Phi_2(G)\}$, respectively. Then, $\mathcal{E}_1 \cup \mathcal{E}_2$ is equal to $\Phi(G)$.

Condition 2: Suppose $\Phi(G) = \neg\Phi_1(G)$. Let $\mathcal{E}_1$ be the expression of graph algebra equal to $\{G|\Phi_1(G)\}$. Then, we have $D(Context(\Phi(G))) - \mathcal{E}_1$ equal to $\Phi(G)$.

Table 3: Datasets and query graphs. |V|, |E|, and |L| are the numbers of vertices, edges, and vertex labels, respectively.

Dataset	$\|V\|$	$\|E\|$	$\|L\|$	Query Graph
Yeast	3,112	12,519	71	$Q_4, Q_{8s}, Q_{8d}, Q_{16s}, Q_{16d}$, $Q_{24s}, Q_{24d}, Q_{32s}, Q_{32d}$
HPRD	9,460	34,998	307	
Human	4,674	86,282	44	$Q_4, Q_{8s}, Q_{8d}, Q_{12s}, Q_{12d}$, $Q_{16s}, Q_{16d}, Q_{20s}, Q_{20d}$

Condition 3: Suppose $\Phi(G) = \exists p(\Phi_1(G))$. Suppose $\mathcal{E}_1$ is the expression of graph algebra equal to $\{G|\Phi_1(G)\}$. Let $\mathcal{S}$ be the schema of $\Phi_1(G)$ and remove the parts of p (including the label and attribute names) from $\mathcal{S}$ to obtain a new schema $\mathcal{S}'_h$. Then, $\rho_{\mathcal{S}'}(\mathcal{E}_1)$ is equal to $\Phi(G)$.

Therefore, it is proved that each subformula $\widehat{\Phi}(G)$ has a graph algebra expression equal to it, which indicates that for each graph calculus expression, there is a graph algebra expression equal to it. In summary, Theorem 6.7 is proved. □

In conclusion, according to Theorem 6.1 and Theorem 6.7, it is proved that the proposed graph algebra and graph calculus have the same expressive power. Therefore, *SOGQL* has the power of MSOL.

7 EFFICIENCY EVALUATION

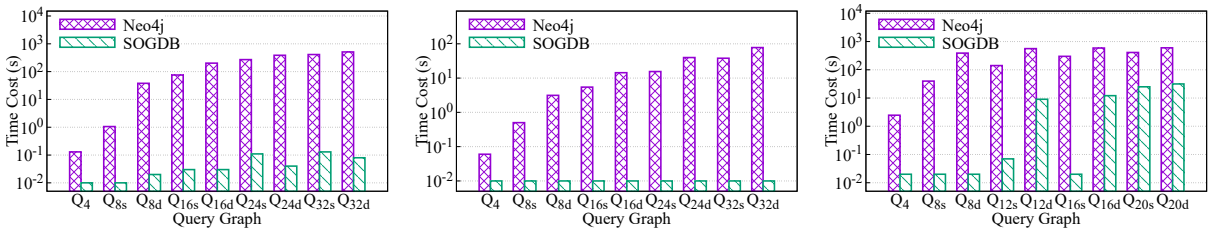
We have build a prototype graph database system (named *SOGDB*) based on *SOGQL*, and experiments are conducted on *SOGDB* to show its efficiency. The details of the experiments are presented in this section.

7.1 Experimental Settings

As one of the most widely-used graph databases, Neo4j is used as the baseline and is compared with *SOGDB*. Note that Neo4j follows the grammar of Cypher, whose expressive power is still unclear due to the lack of the formal definition [13]. However, since Cypher cannot express the maximal clique query, it should not have the expressive power of MSOL.

As subgraph matching is a fundamental query in graph databases [26], we evaluate the efficiency of *SOGDB* with subgraph matching queries. The used datasets and query graphs are presented in Table 3. In detail, three real-world datasets, i.e., Yeast, HPRD, and Human, are used in the experiments, and these datasets are widely-used in previous study [9, 26, 45, 46]. Following [45], for each dataset, nine sets of query graphs are generated with random walk, and each set contains 100 query graphs. Note that each graph in Q_{ns} or Q_{nd} contains n vertices ($n \in \{8, 12, 16, 20, 24, 32\}$), and graphs in Q_{ns} are sparse (with an average degree of $d_{\text{avg}} < 3$), while those in Q_{nd} are relatively dense ($d_{\text{avg}} \geq 3$). For Q_4, the average degree is not constrained. The maximum query graph used for Human only contains 20 vertices, which is fewer than that for Yeast and HPRD, because Human is much denser and subgraph querying on Human is more time-consuming.

In *SOGDB*, the input statement is first analyzed by the parser of *SOGQL*, and then subgraph matching is performed with DAF [26]. The experiments are carried out on a server with Intel Xeon E5-2630 2.20GHz CPU, 315GB RAM, and Ubuntu 16.04 operating system. For Neo4j, the 3.5.22 version is used, and the default settings are used except that the *maximum heap size* is set to 300GB, and the *maximum stack size* is set to 512MB.



(a) Yeast **(b) HPRD** **(c) Human**

Figure 24: The results of the efficiency experiments.

7.2 Experimental Results

In the experiments, all the real-world datasets and their corresponding query graphs presented in Table 3 are utilized. Given a dataset and one of its query graph sets (e.g., $Q = Q_4$), which contains 100 query graphs, 100 queries are performed and the average time cost is reported. Note that the time cost of a query includes the time cost of compiling the query statement and that of matching. Following [45], a query is stopped after 10^5 matching results are obtained. Moreover, if a query cannot finish in ten minutes, the query is stopped and the time cost is recorded as ten minutes. The experimental results are shown in Fig. 24.

The results show that *SOGDB* is usually about 100 times faster than Neo4j, which indicates that *SOGDB* is much more efficient than Neo4j no matter which datasets and query graphs are used. Specifically, *SOGDB* is about 19,482 times faster than Neo4j when Q_{8d} is queried on Human dataset. The experimental results suggest that our system *SOGDB* based on *SOGQL* has good performance, and can be applied to deal with practical problems.

8 RELATED WORK

A well-designed graph query language is crucial for employing graph databases rationally and efficiently, and numerous graph query languages have been proposed in the previous literature. The well-known graph query languages include G-SPARQL [42], Gremlin [38], and Cypher [21, 23]. In detail, G-SPARQL is a SPARQL-like query language for attributed graphs. Gremlin is a graph traversal machine and language that supports both graph traversal and graph pattern matching. Cypher is the language for the property graph model of Neo4j, and it applies ASCII-Art syntax. Moreover, Hölsch et al. [30] propose to optimize queries in Neo4j by extending relational algebra with two operators, i.e., GetNodes and Expand. Besides, Byun et al. [12] extend Gremlin and propose a query language for temporal graphs.

To the best of our knowledge, G [17] is the first graph query language, and it is for labeled directed graphs. As its semantics is based on simple paths, the cost of processing its query statements is unacceptable. Barceló [7] compares G with graph query languages based on arbitrary paths (e.g., RPQs and C2RPQs), and proves the superiority of using arbitrary paths on expressive power, the complexity of evaluation and the containment problem. GraphLog [15] is obtained by improving G. It proposes to use negation in queries and specifies a distinguished edge to define the relation each query result represents. GraphLog is shown to have the same expressive power as stratified linear Datalog.

Object-oriented data models are commonly used to model graphs, and several query languages are proposed for them. GraphDB [24] defines three kinds of classes (i.e., simple classes, link classes, and path classes) to store objects. It supports some practical graph operations such as shortest path query and subgraph query. Gyssens et al. [25] propose a data transformation language on the graph-oriented object database model (abbreviated as GOOD). The language supports the addition and deletion of nodes and edges. Based on the same data model, Paredaens et al. [37] propose a new declarative query language G-Log which performs queries by specifying schemas and rules.

Semistructured data are common in the real world. Several query languages are also proposed for semistructured data. Lorel [1] is one of these languages, and is designed based on OEM (Object Exchange Model) data model. It proposes to employ coercion extensively and supports navigational querying useful when the exact structure is unknown. UnQL [11] is another language for querying semistructured data, and is based on structural recursion. Besides expressing all relational algebra queries, its internal algebra (named UnCAL) can obtain non-flat results and perform polymorphic queries. BiQL [20] is also proposed for semistructured data, and it can call multiple external tools to perform data mining. Moreover, Hidders [29] proposes GDM based on the GOOD approach, and GDM can simulate both object-oriented schema and semistructured data.

Moreover, some graph query languages [19, 40] are proposed based on temporal graph data models. There are also some graph query languages proposed to process data in a specific field, for example, SoQL [39] for social networks, PQL [33] for biological networks, and DNAQL [10] for DNA computing. Furthermore, some studies [6, 48] survey and compare existing graph query languages.

Another main difference of the graph query language is the basic unit. For instance, Gram [2] presents an algebra for hyperwalks, which defines several commonly-used operations such as selection and projection. Levene et al. [34] propose a data model based on hypernodes and a declarative language for it. Moreover, G-CORE [3] and GraphQL [27] use graphs as their basic units. Specifically, G-CORE adds paths into its data model and is a composable query language. GraphQL contains the relational algebra, but it still does not have the power of MSOL.

In this paper, we propose a new graph query language named *SOGQL*. *SOGQL* is based on our graph algebra that can express various complex queries. Therefore, our graph query language can support more useful queries (e.g., clique searching) in one statement than the existing algorithms based on graph patten matching and graph navigation [4]. Besides, the graph algebra is proved to have

the same expressive power as the graph calculus proposed based on MSOL. It confirms that *SOGQL* is more expressive than the existing graph query languages.

9 CONCLUSIONS

As existing graph query languages are usually based on FOL or its extended versions, and have limited expressive power, we propose a new graph query language named *SOGQL* based on MSOL to better support various queries on graph data. First, a graph calculus based on MSOL is proposed. Then, we present a new graph data model with a graph algebra for graph sets, and the new graph algebra is proved to have the same expressive power as our graph calculus. Next, the detailed grammar of our graph query language *SOGQL* is carefully designed and introduced with examples. Some examples of *SOGQL* are provided to confirm that this language can conveniently express complex and useful queries such as cliques searching. Besides, a prototype graph database system named *SOGDB* is implemented and compared with the widely-used graph database system Neo4j. The experimental results show the superiority of *SOGDB* in efficiency. Moreover, there are some interesting future works such as designing efficient query optimization methods for *SOGQL* and further improve our prototype graph database system.

A THE DETAILED GRAMMAR OF *SOGQL*

In this section, we present the grammar of the proposed graph query language *SOGQL*. In detail, *SOGQL* consists of two parts, i.e., DML (Data Manipulation Language) and DDL (Data Definition Language).

⟨*statement*⟩ ::= ⟨*dml-statement*⟩ '`;`' | ⟨*ddl-statement*⟩ '`;`'

⟨*dml-statement*⟩ ::= ⟨*drop-stat*⟩ | ⟨*insert-stat*⟩ | ⟨*remove-stat*⟩ | ⟨*update-stat*⟩ | ⟨*query-stat*⟩

⟨*drop-stat*⟩ ::= ⟨*drop-gset-stat*⟩ | ⟨*drop-g-stat*⟩ | ⟨*drop-schema-stat*⟩

⟨*insert-stat*⟩ ::= ⟨*insert-gset-stat*⟩ | ⟨*insert-g-stat*⟩

⟨*remove-stat*⟩ ::= ⟨*remove-gset-stat*⟩ | ⟨*remove-g-stat*⟩

⟨*drop-gset-stat*⟩ ::= '`drop`' '`graphset`' ⟨*gsid*⟩

⟨*drop-g-stat*⟩ ::= '`drop`' '`graph`' ⟨*gid*⟩

⟨*drop-schema-stat*⟩ ::= '`drop`' '`schema`' ⟨*schemaid*⟩

⟨*insert-gset-stat*⟩ ::= '`insert`' '`into`' '`graphset`' ⟨*gsid*⟩ '`{`' ⟨*graph-name-list*⟩ '`}`'

⟨*insert-g-stat*⟩ ::= '`insert`' '`into`' '`graph`' ⟨*gid*⟩ ⟨*graph*⟩

⟨*graph-name-item*⟩ ::= ⟨*graph*⟩ '`as`' ⟨*gid*⟩ | ⟨*gid*⟩

⟨*graph-name-list*⟩ ::= ⟨*graph-name-item*⟩ | ⟨*graph-name-item*⟩ '`,`' ⟨*graph-name-list*⟩

⟨*remove-gset-stat*⟩ ::= '`remove`' '`from`' '`graphset`' ⟨*gsid*⟩ '`{`' ⟨*graph-list*⟩ '`}`'

⟨*remove-g-stat*⟩ ::= '`remove`' '`from`' '`graph`' ⟨*gid*⟩ '`(`' ⟨*vertex-info*⟩ '`,`' ⟨*edge-info*⟩ '`)`'

⟨*update-stat*⟩ ::= '`update`' ⟨*gsid*⟩ '`at`' ⟨*gid*⟩ '`by`' '`set`' ⟨*attribute-name*⟩ '`=`' ⟨*value*⟩

⟨*name-graph*⟩ ::= '`as`' ⟨*gsid*⟩ |

⟨*graph*⟩ ::= '`(`' ⟨*vertex-info*⟩ '`,`' ⟨*edge-info*⟩ '`,`' ⟨*attribute-info*⟩ '`)`' | '`(`' ⟨*vertex-info*⟩ '`,`' ⟨*edge-info*⟩ '`)`'

⟨*graph-item*⟩ ::= ⟨*graph*⟩ | ⟨*gid*⟩

⟨*graph-list*⟩ ::= ⟨*graph-item*⟩ | ⟨*graph-item*⟩ '`,`' ⟨*graph-list*⟩

⟨*graphset*⟩ ::= '`{`' ⟨*graph-list*⟩ '`}`'

⟨*vertex-info*⟩ ::= '`{`' '`}`' | '`{`' ⟨*vid-list*⟩ '`}`'

⟨*edge-info*⟩ ::= '`{`' '`}`' | '`{`' ⟨*edge-list*⟩ '`}`'

⟨*edge-list*⟩ ::= ⟨*edge*⟩ '`[`' ⟨*lid*⟩ '`]`' | ⟨*eid*⟩ '`=`' ⟨*edge*⟩ '`[`' ⟨*lid*⟩ '`]`' | ⟨*edge*⟩ '`[`' ⟨*lid*⟩ '`]`' '`,`' ⟨*edge-list*⟩ | ⟨*eid*⟩ '`=`' ⟨*edge*⟩ '`[`' ⟨*lid*⟩ '`]`' '`,`' ⟨*edge-list*⟩

⟨*edge*⟩ ::= ⟨*vid-tuple*⟩

⟨*attribute-info*⟩ ::= '`(`' ⟨*node-attribute-info*⟩ '`,`' ⟨*edge-attribute-info*⟩ '`)`' | '`(`' ⟨*node-attribute-info*⟩ '`)`'

⟨*node-attribute-info*⟩ ::= '`{`' '`}`' | '`{`' ⟨*node-attribute-list*⟩ '`}`'

⟨*node-attribute-list*⟩ ::= ⟨*node-attribute-item*⟩ | ⟨*node-attribute-item*⟩ '`,`' ⟨*node-attribute-list*⟩

⟨*node-attribute-item*⟩ ::= ⟨*vid*⟩ '`<`' '`>`' | ⟨*vid*⟩ '`<`' ⟨*attr-eq-item-list*⟩ '`>`'

⟨*edge-attribute-info*⟩ ::= '`{`' '`}`' | '`{`' ⟨*edge-attribute-list*⟩ '`}`'

⟨*edge-attribute-list*⟩ ::= ⟨*edge-attribute-item*⟩ | ⟨*edge-attribute-item*⟩ '`,`' ⟨*edge-attribute-list*⟩

⟨*edge-attribute-item*⟩ ::= ⟨*eid*⟩ '`<`' '`>`' | ⟨*edge*⟩ '`<`' '`>`' | ⟨*eid*⟩ '`<`' ⟨*attr-eq-item-list*⟩ '`>`' | ⟨*edge*⟩ '`<`' ⟨*attr-eq-item-list*⟩ '`>`'

⟨*query-stat*⟩ ::= ⟨*query-stat*⟩ ⟨*multiple-operator*⟩ ⟨*query-stat*⟩ | '`(`' ⟨*query-stat*⟩ '`)`' | '`select`' ⟨*name-graph*⟩ ⟨*reduce-word*⟩ ⟨*maximal-word*⟩ ⟨*project-condition*⟩ ⟨*from-condition*⟩ ⟨*align-condition-list*⟩ ⟨*where-condition*⟩ ⟨*map-condition*⟩ | gsid

⟨*multiple-operator*⟩ ::= '`-`' | '`intersect`' | '`union`'

⟨*project-condition*⟩ ::= ⟨*schema-name-list*⟩ '`for`' ⟨*lid-list*⟩ | '`*`'

⟨*from-condition*⟩ ::= '`from`' ⟨*operator-g-set*⟩ | '`from`' ⟨*operation-statement*⟩

⟨*align-condition-list*⟩ ::= ⟨*align-condition*⟩ ⟨*align-condition-list*⟩ |

⟨*align-condition*⟩ ::= '`align`' ⟨*lid*⟩ '`,`' ⟨*lid*⟩ '`,`' ⟨*lid*⟩ '`on`' ⟨*attr-eq-item-list-uncertain*⟩

⟨*where-condition*⟩ ::= '`where`' ⟨*first-order-stat*⟩ |

⟨*map-condition*⟩ ::= '`map`' '`[`' ⟨*operator-g-set*⟩ '`]`' '`by`' ⟨*gsid*⟩ |

⟨*reduce-word*⟩ ::= '`reduce`' |

⟨*maximal-word*⟩ ::= ‘`maximal`’ |

⟨*operation-statement*⟩ ::= ⟨*product-statement*⟩ | ⟨*join-statement*⟩

⟨*operator-g-set*⟩ ::= ⟨*gsid*⟩ | ⟨*graphset*⟩ ⟨*name-graph*⟩ | ‘`(`’ ⟨*graphset*⟩ ⟨*namea-graph*⟩ ‘`)`’ | ⟨*query-stat*⟩

⟨*schema-name-item*⟩ ::= ‘`{`’ ‘`}`’ | ‘`{`’ ⟨*attribute-name-list*⟩ ‘`}`’ | ‘`*`’

⟨*schema-name-list*⟩ ::= ⟨*schema-name-item*⟩ | ⟨*schema-name-item*⟩ ‘`,`’ ⟨*schema-name-list*⟩

⟨*product-statement*⟩ ::= ⟨*operator-g-set*⟩ ‘`,`’ ⟨*operator-g-set*⟩

⟨*join-statement*⟩ ::= ⟨*operator-g-set*⟩ ‘`join`’ ⟨*operator-g-set*⟩

⟨*value*⟩ ::= ⟨*number*⟩ | ⟨*text*⟩

⟨*ent-id*⟩ ::= ⟨*vid*⟩ | ⟨*eid*⟩

⟨*vid-list*⟩ ::= ⟨*vid*⟩ ‘`[`’ ⟨*lid*⟩ ‘`]`’ | ⟨*vid*⟩ ‘`[`’ ⟨*lid*⟩ ‘`]`’ ‘`,`’ ⟨*vid-list*⟩

⟨*lid-list*⟩ ::= ⟨*lid*⟩ | ⟨*lid*⟩ ‘`,`’ ⟨*lid-list*⟩

⟨*vid-tuple*⟩ ::= ‘`(`’ ⟨*vid*⟩ ‘`,`’ ⟨*vid*⟩ ‘`)`’

⟨*vid-tuple-list*⟩ ::= ⟨*vid-tuple*⟩ | ⟨*vid-tuple*⟩ ‘`,`’ ⟨*vid-tuple-list*⟩

⟨*attr-eq-item*⟩ ::= ⟨*attribute-name*⟩ ‘=’ ⟨*value*⟩

⟨*attr-eq-item-list*⟩ ::= ⟨*attr-eq-item*⟩ | ⟨*attr-eq-item*⟩ ‘`,`’ ⟨*attr-eq-item-list*⟩

⟨*attribute-name-list*⟩ ::= ⟨*attribute-name*⟩ | ⟨*attribute-name*⟩ ‘`,`’ ⟨*attribute-name-list*⟩

⟨*attr-eq-item-uncertain*⟩ ::= ⟨*attribute-name*⟩ ‘=’ ⟨*attribute-name*⟩

⟨*attr-eq-item-list-uncertain*⟩ ::= ⟨*attr-eq-item-uncertain*⟩ | ⟨*attr-eq-item-uncertain*⟩ ‘`,`’ ⟨*attr-eq-item-list-uncertain*⟩

⟨*compare-operator*⟩ ::= ‘`>`’ | ‘`<`’ | ‘`>=`’ | ‘`<=`’ | ‘`=`’ | ‘`!=`’

⟨*calculate-operator*⟩ ::= ‘`+`’ | ‘`-`’ | ‘`*`’ | ‘`/`’

⟨*logic-flag-bi*⟩ ::= ‘`and`’ | ‘`or`’ | ‘`->`’

⟨*logic-flag-not*⟩ ::= ‘`not`’

⟨*first-order-stat*⟩ ::= ⟨*first-order-unit*⟩
| ⟨*first-order-stat*⟩ ⟨*logic-flag-bi*⟩ ⟨*first-order-stat*⟩
| ‘`(`’ ⟨*first-order-stat*⟩ ‘`)`’ ⟨*logic-flag-bi*⟩ ‘`(`’ ⟨*first-order-stat*⟩ ‘`)`’
| ⟨*first-order-stat*⟩ ⟨*logic-flag-bi*⟩ ‘`(`’ ⟨*first-order-stat*⟩ ‘`)`’
| ‘`(`’ ⟨*first-order-stat*⟩ ‘`)`’ ⟨*logic-flag-bi*⟩ ⟨*first-order-stat*⟩
| ⟨*logic-flag-not*⟩ ⟨*first-order-stat*⟩
| ⟨*logic-flag-not*⟩ ‘`(`’ ⟨*first-order-stat*⟩ ‘`)`’
| ‘`True`’ | ‘`False`’

⟨*first-order-unit*⟩ ::= ⟨*exist-stat*⟩ | ⟨*foreach-stat*⟩ | ⟨*compare-val-stat*⟩ | ⟨*func-stat*⟩

⟨*exist-stat*⟩ ::= ‘`Exist`’ ⟨*var-list*⟩ ‘`(`’ ⟨*first-order-stat*⟩ ‘`)`’

⟨*foreach-stat*⟩ ::= ‘`ForEach`’ ⟨*var-list*⟩ ‘`(`’ ⟨*first-order-stat*⟩ ‘`)`’

⟨*compare-stat*⟩ ::= ⟨*compare-unit*⟩
| ⟨*compare-stat*⟩ ⟨*calculate-operator*⟩ ⟨*compare-stat*⟩
| ‘`(`’ ⟨*compare-stat*⟩ ‘`)`’ ⟨*calculate-operator*⟩ ‘`(`’ ⟨*compare-stat*⟩ ‘`)`’
| ⟨*compare-stat*⟩ ⟨*calculate-operator*⟩ ‘`(`’ ⟨*compare-stat*⟩ ‘`)`’
| ‘`(`’ ⟨*compare-stat*⟩ ‘`)`’ ⟨*calculate-operator*⟩ ⟨*compare-stat*⟩
| ‘`True`’ | ‘`False`’

⟨*compare-unit*⟩ ::= ⟨*value*⟩ | ⟨*attribute-name*⟩

⟨*compare-val-stat*⟩ ::= ⟨*compare-stat*⟩ ⟨*compare-operator*⟩ ⟨*compare-stat*⟩

⟨*func-stat*⟩ ::= ⟨*func-name*⟩ ‘`(`’ ⟨*object-list*⟩ ‘`)`’
| ⟨*func-name*⟩ ‘`(`’ ‘`)`’

⟨*func-name*⟩ ::= ‘`Equal`’ | ‘`Vertex`’ | ‘`Edge`’ | ‘`Complete`’ | ‘`Unique`’ | | ‘`Clique`’ | ‘`HasEdge`’ | ‘`SizeGEQ`’ | ‘`GraphEqual`’ | ‘`GraphInSet`’ | ‘`Subgraph`’ | ‘`InducedSub`’ | ‘`Connect`’

⟨*object*⟩ ::= ⟨*variable-name*⟩ | ⟨*variable-name*⟩ ‘`[`’ ⟨*index-int*⟩ ‘`]`’

⟨*object-list*⟩ ::= ⟨*object*⟩ | ⟨*object*⟩ ‘`,`’ ⟨*object-list*⟩

⟨*var-list*⟩ ::= ⟨*variable-name*⟩ | ⟨*variable-name*⟩ ‘`,`’ ⟨*var-list*⟩

⟨*ddl-statement*⟩ ::= ⟨*create-schema-stat*⟩ | ⟨*create-gset-stat*⟩ | ⟨*create-g-stat*⟩

⟨*create-schema-stat*⟩ ::= ‘`create`’ ‘`schema`’ ⟨*schemaid*⟩ ‘`{`’ ⟨*schema-type-list*⟩ ‘`}`’

⟨*schema-item*⟩ ::= ‘`{`’ ⟨*schema-type-list*⟩ ‘`}`’ | ⟨*schemaid*⟩

⟨*create-gset-stat*⟩ ::= ‘`create`’ ‘`graphset`’ ⟨*gsid*⟩ ⟨*schema-item*⟩

⟨*create-g-stat*⟩ ::= ‘`create`’ ‘`graph`’ ⟨*gid*⟩ ⟨*schema-item*⟩

⟨*schema-type-list*⟩ ::= ⟨*lid*⟩ ‘`<`’ ⟨*schema-attribute-item*⟩ ‘`>`’ | ⟨*lid*⟩ ‘`<`’ ⟨*schema-attribute-item*⟩ ‘`>`’ ‘`,`’ ⟨*schema-type-list*⟩

⟨*schema-attribute-item*⟩ ::= ⟨*attr-item*⟩ | ⟨*attr-item*⟩ ‘`,`’ ⟨*schema-attribute-item*⟩

⟨*attr-item*⟩ ::= ⟨*attribute-name*⟩ ‘`:`’ ⟨*data-type*⟩

⟨*data-type*⟩ ::= ‘`INT`’ | ‘`Float`’ | ‘`Double`’ | ‘`Char`’ | ‘`String`’

⟨*attribute-name*⟩ ::= ⟨*name-string*⟩

⟨*gsid*⟩ ::= ⟨*name-string*⟩

⟨*gid*⟩ ::= ⟨*name-string*⟩

⟨*vid*⟩ ::= ⟨*name-string*⟩

⟨*eid*⟩ ::= ⟨*name-string*⟩

⟨*lid*⟩ ::= ⟨*name-string*⟩

⟨*schemaid*⟩ ::= ⟨*name-string*⟩

⟨*variable-name*⟩ ::= ⟨*name-string*⟩

⟨*index-int*⟩ ::= ⟨*integer*⟩

⟨*letter*⟩ ::= ‘A’ | ‘B’ | ‘C’ | ‘D’ | ‘E’ | ‘F’ | ‘G’ | ‘H’ | ‘I’ | ‘J’ | ‘K’ | ‘L’ | ‘M’ | ‘N’ | ‘O’ | ‘P’ | ‘Q’ | ‘R’ | ‘S’ | ‘T’ | ‘U’ | ‘V’ | ‘W’ | ‘X’ | ‘Y’ | ‘Z’ | ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i’ | ‘j’ | ‘k’ | ‘l’ | ‘m’ | ‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t’ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ | ‘z’

⟨*digit*⟩ ::= ‘0’ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’

⟨*symbol*⟩ ::= ‘|’ | ‘!’ | ‘#’ | ‘$’ | ‘%’ | ‘&’ | ‘(’ | ‘)’ | ‘*’ | ‘+’ | ‘,’ | ‘-’ | ‘.’ | ‘/’ | ‘:’ | ‘;’ | ‘>’ | ‘=’ | ‘<’ | ‘?’ | ‘@’ | ‘[’ | ‘\’ | ‘]’ | ‘^’ | ‘_’ | ‘{’ | ‘}’ | ‘~’

⟨*char*⟩ ::= ⟨*letter*⟩ | ⟨*digit*⟩ | ⟨*symbol*⟩

⟨*text*⟩ ::= ‘’’ (⟨*char*⟩ | ‘ ’)+ ‘’’

⟨*name-string*⟩ ::= ((⟨*char*⟩)+ ‘.’)*(⟨*letter*⟩ | ⟨*digit*⟩)+(‘’’)*

⟨*integer*⟩ ::= [0-9] | [1-9]+[0-9]*

⟨*number*⟩ ::= ‘–’ ⟨*integer*⟩ ‘.’ ⟨*integer*⟩ | ‘–’ ⟨*integer*⟩ | ⟨*integer*⟩ ‘.’ ⟨*integer*⟩ | ⟨*integer*⟩

REFERENCES

[1] Serge Abiteboul, Dallan Quass, Jason McHugh, Jennifer Widom, and Janet L Wiener. 1997. The Lorel query language for semistructured data. *International journal on digital libraries* 1, 1 (1997), 68–88.

[2] Bernd Amann and Michel Scholl. 1993. Gram: a graph data model and query languages. In *Proceedings of the ACM conference on Hypertext*. 201–211.

[3] Renzo Angles, Marcelo Arenas, Pablo Barceló, Peter Boncz, George Fletcher, Claudio Gutierrez, Tobias Lindaaker, Marcus Paradies, Stefan Plantikow, Juan Sequeda, et al. 2018. G-CORE: A core for future graph query languages. In *Proceedings of the 2018 International Conference on Management of Data*. 1421–1432.

[4] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of modern query languages for graph databases. *ACM Computing Surveys (CSUR)* 50, 5 (2017), 1–40.

[5] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W. Hare, Jan Hidders, Victor E. Lee, Bei Li, Leonid Libkin, Wim Martens, Filip Murlak, Josh Perryman, Ognjen Savković, Michael Schmidt, Juan Sequeda, Slawek Staworko, and Dominik Tomaszuk. 2021. PG-Keys: Keys for Property Graphs. In *Proceedings of the 2021 International Conference on Management of Data* (Virtual Event, China) *(SIGMOD '21)*. Association for Computing Machinery, New York, NY, USA, 2423–2436.

[6] Renzo Angles and Claudio Gutierrez. 2018. An introduction to graph data management. In *Graph Data Management*. Springer, 1–32.

[7] Pablo Barceló Baeza. 2013. Querying graph databases. In *Proceedings of the 32nd ACM SIGMOD-SIGACT-SIGAI symposium on Principles of database systems*. 175–188.

[8] Jon Barwise. 1977. An introduction to first-order logic. In *Studies in Logic and the Foundations of Mathematics*. Vol. 90. Elsevier, 5–46.

[9] Fei Bi, Lijun Chang, Xuemin Lin, Lu Qin, and Wenjie Zhang. 2016. Efficient subgraph matching by postponing cartesian products. In *Proceedings of the 2016 International Conference on Management of Data*. 1199–1214.

[10] Robert Brijder, Joris JM Gillis, and Jan Van den Bussche. 2013. The DNA query language DNAQL. In *Proceedings of the 16th International Conference on Database Theory*. 1–9.

[11] Peter Buneman, Mary Fernandez, and Dan Suciu. 2000. UnQL: a query language and algebra for semistructured data based on structural recursion. *The VLDB Journal* 9, 1 (2000), 76–110.

[12] Jaewook Byun, Sungpil Woo, and Daeyoung Kim. 2019. Chronograph: Enabling temporal graph traversals for efficient information diffusion analysis over time. *IEEE Transactions on Knowledge and Data Engineering* 32, 3 (2019), 424–437.

[13] Jaime Castro and Adrián Soto. 2017. A Comparison between Cypher and Conjunctive Queries.. In *AMW*.

[14] Edgar F Codd. 2009. Derivability, redundancy and consistency of relations stored in large data banks. *ACM SIGMOD Record* 38, 1 (2009), 17–36.

[15] Mariano P Consens and Alberto O Mendelzon. 1990. GraphLog: a visual formalism for real life recursion. In *Proceedings of the ninth ACM SIGACT-SIGMOD-SIGART symposium on Principles of database systems*. 404–416.

[16] Bruno Courcelle and Joost Engelfriet. 2012. *Graph structure and monadic second-order logic: a language-theoretic approach*. Vol. 138. Cambridge University Press.

[17] Isabel F Cruz, Alberto O Mendelzon, and Peter T Wood. 1987. A graphical query language supporting recursion. *ACM SIGMOD Record* 16, 3 (1987), 323–330.

[18] Joana MF da Trindade, Konstantinos Karanasos, Carlo Curino, Samuel Madden, and Julian Shun. 2020. Kaskade: Graph views for efficient graph analytics. In *2020 IEEE 36th International Conference on Data Engineering (ICDE)*. IEEE, 193–204.

[19] Ariel Debrouvier, Eliseo Parodi, Matías Perazzo, Valeria Soliani, and Alejandro Vaisman. 2021. A model and query language for temporal graph databases. *The VLDB Journal* 30, 5 (2021), 825–858.

[20] Anton Dries, Siegfried Nijssen, and Luc De Raedt. 2009. A query language for analyzing networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*. 485–494.

[21] 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*. 1433–1445.

[22] Sergio Greco and Cristian Molinaro. 2015. Datalog and logic databases. *Synthesis Lectures on Data Management* 7, 2 (2015), 1–169.

[23] Alastair Green, Paolo Guagliardo, Leonid Libkin, Tobias Lindaaker, Victor Marsault, Stefan Plantikow, Martin Schuster, Petra Selmer, and Hannes Voigt. 2019. Updating Graph Databases with Cypher. *Proc. VLDB Endow.* 12, 12 (Aug. 2019), 2242–2254.

[24] Ralf Hartmut Güting. 1994. GraphDB: Modeling and querying graphs in databases. In *VLDB*, Vol. 94. 12–15.

[25] Marc Gyssens, Jan Paredaens, Jan Van den Bussche, and Dirk Van Gucht. 1994. A graph-oriented object database model. *IEEE Transactions on knowledge and Data Engineering* 6, 4 (1994), 572–586.

[26] Myoungji Han, Hyunjoon Kim, Geonmo Gu, Kunsoo Park, and Wook-Shin Han. 2019. Efficient Subgraph Matching: Harmonizing Dynamic Programming, Adaptive Matching Order, and Failing Set Together. In *Proceedings of the 2019 International Conference on Management of Data* (Amsterdam, Netherlands) *(SIGMOD '19)*. Association for Computing Machinery, New York, NY, USA, 1429–1446.

[27] Huahai He and Ambuj K Singh. 2008. Graphs-at-a-time: query language and access methods for graph databases. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*. 405–418.

[28] Jesper G Henriksen, Jakob Jensen, Michael Jørgensen, Nils Klarlund, Robert Paige, Theis Rauhe, and Anders Sandholm. 1995. Mona: Monadic second-order logic in practice. In *International Workshop on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 89–110.

[29] Jan Hidders. 2003. Typing graph-manipulation operations. In *International Conference on Database Theory*. Springer, 394–409.

[30] Jürgen Hölsch and Michael Grossniklaus. 2016. An algebra and equivalences to transform graph patterns in neo4j. In *EDBT/ICDT 2016 Workshops: EDBT Workshop on Querying Graph Structured Data (GraphQ)*.

[31] Michael Huth and Mark Ryan. 2004. *Logic in Computer Science: Modelling and reasoning about systems*. Cambridge university press.

[32] Kevin C. Klement. 2004. Propositional Logic. In *Internet Encyclopedia of Philosophy*.

[33] Ulf Leser. 2005. A query language for biological networks. *Bioinformatics* 21, suppl_2 (2005), ii33–ii39.

[34] Mark Levene and Alexandra Poulovassilis. 1990. The hypernode model and its associated query language. In *Proceedings of the 5th Jerusalem Conference on Information Technology, 1990.'Next Decade in Information Technology'*. IEEE, 520–530.

[35] Can Lu, Jeffrey Xu Yu, Hao Wei, and Yikai Zhang. 2017. Finding the maximum clique in massive graphs. *Proceedings of the VLDB Endowment* 10, 11 (2017), 1538–1549.

[36] Symeon Papadopoulos, Yiannis Kompatsiaris, Athena Vakali, and Ploutarchos Spyridonos. 2012. Community detection in social media. *Data Mining and Knowledge Discovery* 24, 3 (2012), 515–554.

[37] Jan Paredaens, Peter Peelman, and Letizia Tanca. 1995. G-Log: A graph-based query language. *IEEE Transactions on Knowledge and Data Engineering* 7, 3 (1995), 436–453.

[38] Marko A Rodriguez. 2015. The gremlin graph traversal machine and language (invited talk). In *Proceedings of the 15th Symposium on Database Programming Languages*. 1–10.

[39] Royi Ronen and Oded Shmueli. 2009. SoQL: A language for querying and creating data in social networks. In *2009 IEEE 25th International Conference on Data Engineering*. IEEE, 1595–1602.

[40] Christopher Rost, Kevin Gomez, Matthias Täschner, Philip Fritzsche, Lucas Schons, Lukas Christ, Timo Adameit, Martin Junghanns, and Erhard Rahm. 2022. Distributed temporal graph analytics with GRADOOP. *The VLDB journal* 31, 2 (2022), 375–401.

[41] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M Tamer Özsu. 2019. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *The VLDB Journal* (2019), 1–24.

[42] Sherif Sakr, Sameh Elnikety, and Yuxiong He. 2012. G-SPARQL: a hybrid engine for querying large attributed graphs. In *Proceedings of the 21st ACM international conference on Information and knowledge management*. 335–344.

[43] Matthew C Schmidt, Nagiza F Samatova, Kevin Thomas, and Byung-Hoon Park. 2009. A scalable, parallel algorithm for maximal clique enumeration. *J. Parallel and Distrib. Comput.* 69, 4 (2009), 417–428.

[44] Abraham Silberschatz, Henry F. Korth, and S. Sudershan. 2019. *Database System Concepts* (7th ed.). McGraw-Hill, Inc.

[45] Shixuan Sun and Qiong Luo. 2020. In-memory subgraph matching: An in-depth study. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1083–1098.

[46] Shixuan Sun and Qiong Luo. 2022. Subgraph Matching With Effective Matching Order and Indexing. *IEEE Transactions on Knowledge and Data Engineering* 34, 1 (2022), 491–505. https://doi.org/10.1109/TKDE.2020.2980257

[47] Ruijie Wang, Meng Wang, Jun Liu, Siyu Yao, and Qinghua Zheng. 2018. Graph embedding based query construction over knowledge graphs. In *ICBK 2018*. IEEE, 1–8.

[48] Peter T Wood. 2012. Query languages for graph databases. *ACM Sigmod Record* 41, 1 (2012), 50–60.