Towards a Converged Relational-Graph Optimization Framework

YUNKAI LOU, Alibaba Group, China
LONGBIN LAI, Alibaba Group, China
BINGQING LYU, Alibaba Group, China
YUFAN YANG, Alibaba Group, China
XIAOLI ZHOU, Alibaba Group, China
WENYUAN YU, Alibaba Group, China
YING ZHANG, Zhejiang Gongshang University, China
JINGREN ZHOU, Alibaba Group, China

The recent ISO SQL:2023 standard adopts SQL/PGQ (Property Graph Queries), facilitating graph-like querying within relational databases. This advancement, however, underscores a significant gap in how to effectively optimize SQL/PGQ queries within relational database systems. To address this gap, we extend the foundational SPJ (Select-Project-Join) queries to SPJM queries, which include an additional matching operator for representing graph pattern matching in SQL/PGQ. Although SPJM queries can be converted to SPJ queries and optimized using existing relational query optimizers, our analysis shows that such a graph-agnostic method fails to benefit from graph-specific optimization techniques found in the literature. To address this issue, we develop a converged relational-graph optimization framework called RelGo for optimizing SPJM queries, leveraging joint efforts from both relational and graph query optimizations. Using DuckDB as the underlying relational execution engine, our experiments show that RelGo can generate efficient execution plans for SPJM queries. On well-established benchmarks, these plans exhibit an average speedup of 21.90× compared to those produced by the graph-agnostic optimizer.

CCS Concepts: • **Do Not Use This Code → Generate the Correct Terms for Your Paper**; *Generate the Correct Terms for Your Paper*; Generate the Correct Terms for Your Paper; Generate the Correct Terms for Your Paper.

Additional Key Words and Phrases: Do, Not, Us, This, Code, Put, the, Correct, Terms, for, Your, Paper

ACM Reference Format:
Yunkai Lou, Longbin Lai, Bingqing Lyu, Yufan Yang, Xiaoli Zhou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2024. Towards a Converged Relational-Graph Optimization Framework. *Proc. ACM Manag. Data* 2, 6 (SIGMOD), Article 252 (December 2024), 27 pages. https://doi.org/10.1145/3698828

Authors' Contact Information: Yunkai Lou, Alibaba Group, Hangzhou, China, louyunkai.lyk@alibaba-inc.com; Longbin Lai, Alibaba Group, Hangzhou, China, longbin.lailb@alibaba-inc.com; Bingqing Lyu, Alibaba Group, Hangzhou, China, bingqing.lbq@alibaba-inc.com; Yufan Yang, Alibaba Group, Hangzhou, China, xiaofan.yyf@alibaba-inc.com; Xiaoli Zhou, Alibaba Group, Hangzhou, China, yihe.zxl@alibaba-inc.com; Wenyuan Yu, Alibaba Group, Hangzhou, China, wenyuan.ywy@alibaba-inc.com; Ying Zhang, Zhejiang Gongshang University, Hangzhou, China, ying.zhang@zjgsu.edu.cn; Jingren Zhou, Alibaba Group, Hangzhou, China, jingren.zhou@alibaba-inc.com.

ACM 2836-6573/2024/12-ART252
https://doi.org/10.1145/3698828

1 Introduction

In the realms of data management and analytics, relational databases have long been the bedrock of structured data storage and retrieval, empowering a plethora of applications. The ubiquity of these databases has been supported by the advent of Structured Query Language (SQL) [9], a standardized language that has been adopted widely by various relational database management systems for managing data through schema-based operations.

Despite its considerable success and broad adoption, SQL has its limitations, particularly when it comes to representing and querying intricately linked data. Consider, for instance, the relational tables of `Person` and `Knows`, the latter symbolizing a many-to-many relationship between instances of the former. Constructing a SQL query to retrieve a group of four persons who are all mutually acquainted is not a straightforward endeavor, potentially leading to a cumbersome and complex SQL expression.

In comparison, such a scenario could be succinctly addressed using graph query languages such as Cypher [3], where queries are expressed as graph pattern matching. This discrepancy between the relational and graph querying paradigms has given rise to the innovative SQL/Property Graph Queries (SQL/PGQ), an extension formally adopted in the ISO SQL:2023 standard [40]. SQL/PGQ is designed to amalgamate the extensive capabilities of SQL with the inherent benefits of graph pattern matching. With SQL/PGQ, it is now possible to define and query graphs within SQL expressions, transforming otherwise complex relational queries – characterized by multiple joins – into simpler and more intuitive graph queries.

```
SELECT p2_name, place.name
FROM GRAPH_TABLE (G
  MATCH
    (p1:Person)-[:Likes]->(m:Message),
    (p2:Person)-[:Likes]->(m),
    (p1)-[:Knows]->(p2)
  COLUMNS (
    p1.name AS p1_name,
    p1.place_id AS p1_place_id,
    p2.name AS p2_name
  )
) g
JOIN Place p ON g.p1_place_id = p.id
WHERE g.p1_name = 'Tom';
```

Fig. 1. An example of SQL/PGQ query.

Example 1. *Consider the four relational tables in the database:* `Person(id, name, place_id)`, `Message(id, content, date)`, `Like(p_id, m_id, date)`, *and* `Place(id, name)`. *Using SQL/PGQ, a property graph G is articulated as a* `GRAPH_TABLE`, *established on the basis of the first three tables. In this mapping, rows from* Person *and* Message *are interpreted as vertices with labels "Person" and "Message" respectively, while rows from Like represent edges with the label "Likes". This mapping process will be elaborated as* RGMapping *in Sec. 2.1. An SQL/PGQ query to discover the friends of a person named "Tom" and the place they live in, where "Tom" and friends share an affinity for the same message, can be formulated as shown in Fig. 1. In graph G, a* Graph Pattern Matching *is employed to decode the intricate relationships between persons and messages. Upon executing the pattern matching, a* `COLUMNS` *clause projects the results into a tabular format, enumerating essential attributes. Then the* Relational `JOIN` *is performed on resultant table* `g` *and* `Place` *table to obtain the place's name.*

Table 1. Frequently used notations.

Notation	Definition
R	a relation or relational table
τ and $\tau.attr$	a tuple in a relation, and the value of an attribute of τ
$G(V, E)$	a property graph with V and E
$\mathcal{P}(V, E)$	a pattern graph with V and E
$\mathrm{id}(\epsilon)$, $\ell(\epsilon)$, $\epsilon.\mathrm{attr}$	the identifier, label, and the value of given attribute of a graph element ϵ
$\mathcal{N}(u)$ and $\mathcal{N}^E(u)$	neighbors and adjacent edges of u
GR	a graph relation
$\mathcal{M}(GR, \mathcal{P})$, $\mathcal{M}(\mathcal{P})$	matching $\mathcal{P}$ on a graph relation GR or a graph G
π_A, σ_Ψ, $\bowtie$	projection, selection, and join operators over relations
$\widehat{\pi}_{A*}$, $\widehat{\bowtie}$	projection and join operators over graph relations
$\lambda^s_\ell(e)$, $\lambda^t_\ell(e)$	the total functions for mapping tuples in an edge relation to source and target vertex relations

The SQL/PGQ standardization, while a significant leap forward in the realm of relational databases, primarily addresses language constructs. A discernible gap exists in the theoretical landscape, particularly in analyzing, transforming, and optimizing SQL/PGQ queries with hybrid relational and graph semantics.

Relational query optimization has historically leaned on the SPJ (selection-projection-join) skeleton [11, 43], which provides a systematic approach for analyzing query complexity [10, 21] , devising heuristic optimization rules [12, 16], and computing optimal join order [13, 18]. Recently, graph techniques have been introduced to optimize relational queries [18, 23, 32, 33]. In particular, GRainDB [23] introduced a predefined join operator that materializes the adjacency list (rows) of vertices, enabling more efficient join execution. While these techniques can be empowered by graph techniques, they target purely relational query rather than the relational-graph hybrid query of SQL/PGQ.

In parallel to relational query optimization, significant strides have been made in optimizing graph pattern matching. A common practice is to leverage join-based techniques to optimize the query [5, 27, 28, 51]. Scalable join algorithms, such as binary-join [27], worst-case optimal (abbr. wco) join [5], and their hybrid variants [29, 36, 51], have been proposed for solving the problem over large-scale graphs. However, despite the effectiveness of these techniques for pattern matching on graphs, they cannot be directly applied to relational databases due to the inherent differences in data models.

In this paper, we propose the first converged optimization framework, RelGo, that optimizes relational-graph hybrid queries in a relational database, in response to the advent of SQL/PGQ. A straightforward implementation [1, 47, 48] can involve directly transforming the graph component in SQL/PGQ queries into relational operations, allowing the entire query to be optimized and executed in any existing relational engine. While we contribute to building the theory to make such a transformation workable, this *graph-agnostic* optimization approach suffers from several issues, including graph-unaware join orders, suboptimal join plans, and increased search space, as will be discussed in Sec. 3.1.2.

To address these challenges, RelGo is proposed to leverage the strengths of both relational and graph query optimization techniques. Building upon the foundation of SPJ queries, we introduce the SPJM query skeleton, which extends SPJ with a matching operator to represent graph queries. We adapt state-of-the-art graph optimization techniques, such as the decomposition method [51] and the cost-based optimizer [29], to the relational context, effectively producing worst-case optimal graph subplans for the matching operator. To facilitate efficient execution of the matching operator, we introduce graph index inspired by GRainDB's predefined join [23], based on which graph-based physical operations are implemented. The relational part of the query, together with the optimized graph subplans encapsulated within a special operator called SCAN_GRAPH_TABLE, is then optimized using standard relational optimizers. Finally, we incorporate heuristic rules, such as FilterIntoMatchRule, to handle cases unique to SPJM queries that involve the interplay between relational and graph components.

We have made the following contributions in this paper:

(1) We map relational data models to property graph models as specified by SQL/PGQ using RGMapping. Based on RGMapping, we introduce a new query skeleton called SPJM, which is designed to better analyze relational-graph hybrid queries. (Sec. 2)
(2) We construct the theory for transforming any SPJM query into an SPJ query. Such a graph-agnostic approach enables existing relational databases to handle SPJM queries without low-level modifications. We also formally prove that the search space of the graph-agnostic approach can be exponentially larger than our solution. (Sec. 3)
(3) We introduce RelGo, a converged optimization framework that leverages the strengths of both relational and graph query optimization techniques to optimize SPJM queries. RelGo adapts state-of-the-art graph optimization techniques to the relational context, and implements graph-based physical operations based on graph index for efficient query execution. (Sec. 4)
(4) We develop RelGo by integrating it with the industrial relational optimization framework, Calcite [17], and employing DuckDB [2] for execution runtime. We conducted extensive experiments to evaluate its performance. The results on the LDBC Social Network Benchmark [30] indicate that RelGo significantly surpasses the performance of the graph-agnostic baseline, with an average speedup of 21.9×, and 5.4× even after graph index is enabled for the baseline. (Sec. 5)

This paper is organized in the order of the contributions. We survey related work in Sec. 6 and conclude the paper in Sec. 7.

2 Preliminaries

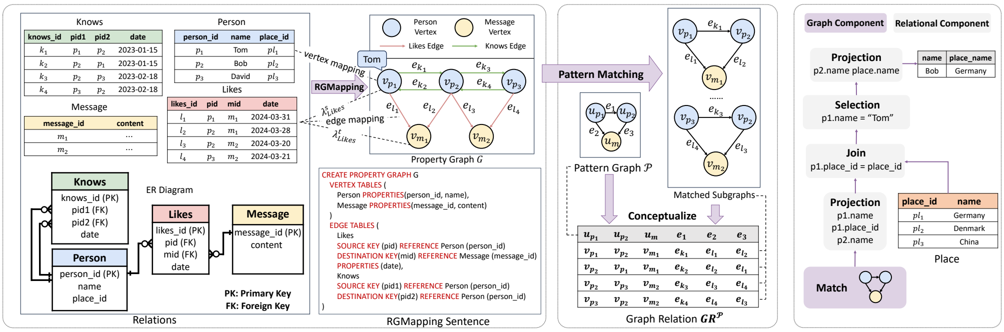
In this section, we propose the utilized data model and define the SPJM query processed in this paper. Frequently used notations in this paper are summarized in Table 1.

2.1 Data Model

A schema, denoted as $S = (a_1, a_2, \ldots, a_n)$, is a collection of attributes. Each attribute a_i is associated with a specific data domain D_i, which defines the set of permissible values that a_i can take. A relation R is defined as a set of tuples. We consider R to be a relation over schema S, if and only if, every tuple $\tau = (d_1, d_2, \ldots, d_n)$ in R adheres to the schema's constraints, such that the value d_i for each position in the tuple corresponds to the data domain D_i of the attribute a_i in S. In other words, each value d_i in a tuple τ is drawn from the appropriate data domain D_i for its corresponding attribute a_i. Moreover, for any tuple τ in the relation R, the notation $\tau.a_i = d_i$ signifies that the attribute a_i in tuple τ has value d_i. A table is a representation of a relation with rows corresponds to tuples in the relation, and columns represent attributes in the schema. In this paper, we use the terms of relation and table interchangeably.

We define a *Property Graph* as $G = (V_G, E_G)$, where V stands for the set of vertices. Let $E \subseteq V \times V$ denote the set of edges in the graph. An edge $e \in E$ is represented as an ordered pair $e = (v_s, v_t)$, where $v_s \in V$ is the source vertex and $v_t \in V$ is the target vertex, indicating that the edge e connects from v_s to v_t. For any graph element ϵ that is either a vertex or an edge, we denote $\mathrm{id}(\epsilon)$ and $\ell(\epsilon)$ as the globally unique ID and the label of ϵ, respectively. Given an attribute a_i, $\epsilon.a_i$ denotes the value of the attribute a of ϵ.

Given a vertex v, we denote its adjacent edges as $\mathcal{N}_G^E(v) = \{e = (v, v_t) | e \in E\}$ and its adjacent vertices (i.e., neighbors) as $\mathcal{N}_G(v) = \{v_t | (v, v_t) \in E\}$. It is important to note that the adjacent edges



Knows

knows_id	pid1	pid2	date
k_1	p_1	p_2	2023-01-15
k_2	p_2	p_1	2023-01-15
k_3	p_2	p_3	2023-02-18
k_4	p_3	p_2	2023-02-18

Person

person_id	name	place_id
p_1	Tom	pl_1
p_2	Bob	pl_2
p_3	David	pl_3

Message

message_id	content
m_1	...
m_2	...

Likes

likes_id	pid	mid	date
l_1	p_1	m_1	2024-03-31
l_2	p_2	m_1	2024-03-28
l_3	p_2	m_2	2024-03-20
l_4	p_3	m_2	2024-03-21

```
CREATE PROPERTY GRAPH G
  VERTEX TABLES (
    Person PROPERTIES(person_id, name),
    Message PROPERTIES(message_id, content)
  )
  EDGE TABLES (
    Likes
    SOURCE KEY (pid) REFERENCE Person (person_id)
    DESTINATION KEY(mid) REFERENCE Message (message_id)
    PROPERTIES (date),
    Knows
    SOURCE KEY (pid1) REFERENCE Person (person_id)
    DESTINATION KEY(pid2) REFERENCE Person (person_id)
)
```

u_{p_1}	u_{p_2}	u_m	e_1	e_2	e_3
v_{p_1}	v_{p_2}	v_{m_1}	e_{k_1}	e_{l_1}	e_{l_2}
v_{p_2}	v_{p_1}	v_{m_1}	e_{k_2}	e_{l_2}	e_{l_1}
v_{p_2}	v_{p_3}	v_{m_2}	e_{k_3}	e_{l_3}	e_{l_4}
v_{p_3}	v_{p_2}	v_{m_2}	e_{k_4}	e_{l_4}	e_{l_3}

Projection
p2.name place.name

name	place_name
Bob	Germany

Selection
p1.name = “Tom”

Join
p1.place_id = place_id

Projection
p1.name
p1.place_id
p2.name

place_id	name
pl_1	Germany
pl_2	Denmark
pl_3	China

(a) The Process of RGMapping
(b) Apply Pattern Matching on G and Conceptualize the Matching Results
(c) The SPJM Query

Fig. 2. An example of RGMapping.

and vertices can be defined for both directions of an edge $e = (v_s, v_t)$, i.e., when $v = v_s$ or $v = v_t$. However, for simplicity, we only define one direction in this notation. In the actual semantics of the paper, both directions may be considered. The degree of v is defined as $d_G(v) = |\mathcal{N}_G(v)|$, and the average degree of all vertices in the graph is $\overline{d}_G = \frac{1}{|V_G|} \sum_{v \in V_G} d_G(v)$. In the rest of the paper, when the context is clear, we may remove G from the subscript for simplicity, for example $G = (V, E)$.

Considering two graphs G_1 and G_2, we assert that G_2 is a subgraph of G_1, symbolized as $G_2 \subseteq G_1$, if and only if $V_{G_2} \subseteq V_{G_1}$, and $E_{G_2} \subseteq E_{G_1}$. Furthermore, G_2 qualifies as an induced subgraph of G_1 under the condition that G_2 is already a subgraph of G_1, and for every pair of vertices in G_2, any edge e that exists between them in G_1 must also present in G_2.

To illustrate the integration of graph syntax within the realm of relational data, we introduce the concept of a *Relations-to-Graph Mapping* (i.e. RGMapping), to facilitate the transformation of relational data structures into a property graph.

An RGMapping consists of an vertex mapping and an edge mapping that map tuples in relations to unique vertices or edges. To better describe these vertex and edge mappings, we can leverage the Entity-Relationship (ER) diagram [14, 46]. In relational data modeling, an ER diagram includes entities and relationships. Consequently, vertices can be mapped from relations corresponding to entities, and edges can be mapped from relations corresponding to relationships. Relations mapped to vertices and edges are referred to as vertex relations and edge relations, respectively.

In detail, if a tuple τ in relation R is mapped to a vertex $v \in V$ (or an edge $e = (v_s, v_t) \in E$), it is assigned an ID id(v) (or id(e)), a label $\ell(v)$ (or $\ell(e)$) that corresponds to the name of R, and attributes $v.attr*$ (or $e.attr*$) that reflect the attributes $attr*$ of τ. For an edge relation R_e, there must exist two vertex relations, R_{p_s} and R_{p_t}. Two total functions are defined: $\lambda^s_e : R_e \longrightarrow R_{p_s}$ and $\lambda^t_e : R_e \longrightarrow R_{p_t}$. Consider a tuple $\tau \in R_e$ mapped to an edge e, and tuples $\tau_s \in R_{p_s}$ and $\tau_t \in R_{p_t}$, where $\lambda^s_e(e) = \tau_s$ and $\lambda^t_e(e) = \tau_t$. Through the vertex mapping, τ_s is mapped to the source vertex v_s and τ_t to the target vertex v_t of the edge e. The two total functions are often established through primary-foreign key relationships, as illustrated in an ER diagram.

Example 2. *In Fig. 2(a), we have illustrated some relational tables and their corresponding ER diagram. An* RGMapping *can be defined following the grammar of SQL/PGQ with* `CREATE PROPERTY GRAPH` *statements. The described* RGMapping *involves assigning tuples from vertex relations (i.e. entities), such as R_{Person} and $R_{Message}$, to graph vertices. For instance, the vertex v_{p_1} is associated with the tuple τ_{p_1} in R_{Person}, and thus assigned the label “Person” and the name attribute “Tom”. Similarly, edge relations (i.e. relationships) R_{Likes} and R_{Knows} correspond to graph edges. Regarding R_{Likes} that is mapped to graph edges, two total functions can be identified, namely $\lambda^s_{Likes} : R_{Likes} \longrightarrow R_{Person}$ and $\lambda^t_{Likes} : R_{Likes} \longrightarrow R_{Message}$. Let's consider the edge e_{l_1}. It originates from the tuple τ_{l_1} in the R_{Likes}*

relation. Its source vertex v_{p_1} is linked to the tuple τ_{p_1} in R_{Person} via the function λ^s_{Likes}, following the primary-foreign key relationship "$\tau_{l_1}.pid = \tau_{p_1}.person\_id$". Similarly, its target vertex v_{m_1} is associated with the tuple τ_{m_1} in $R_{Message}$ via the function λ^t_{Likes}, following "$\tau_{l_1}.mid = \tau_{m_1}.message\_id$". As a result of this mapping, the edge e_{l_1} is assigned the label "Likes" and the attribute "date" with the value "2024-03-31".

2.2 Matching Operator

Consider a property graph $G(V_G, E_G)$, alongside a *connected* pattern graph, represented as $\mathcal{P}(V_\mathcal{P}, E_\mathcal{P})$. Here, $\mathcal{P}$ is a property graph that does not possess attributes, and we denote n and m as the number of vertices and edges in the $\mathcal{P}$, respectively. Graph pattern matching seeks to determine all subgraphs in G that are *homomorphic* to $\mathcal{P}$. Formally, given a subgraph $g \subseteq G$, a homomorphism from $\mathcal{P}$ to g is a *surjective*, total mapping $f : V_\mathcal{P} \cup E_\mathcal{P} \longrightarrow V_g \cup E_g$ that satisfies the following conditions: (1) For every vertex $u \in V_\mathcal{P}$, there is a corresponding vertex $v = f(u) \in V_g$ with $\ell(v) = \ell(u)$; (2) For each edge $e = (u_s, u_t) \in E_\mathcal{P}$, there is a corresponding edge $f(e) = (v_s, v_t) \in E_g$, ensuring that the mapping preserves the edge's the label, as well as its source and target vertices, that is $\ell(e) = \ell(f(e))$, and $f(u_s) = v_s$, $f(u_t) = v_t$. It's important to highlight the homomorphism semantics, as one of the widely used semantics for graph pattern matching [6], do not require each pattern vertex and edge being uniquely mapped to distinct vertices and edges in the data graph. This facilitates a seamless integration between graph pattern matching and relational operations, but alternative semantics for graph pattern matching such as isomorphism can also be adopted, as will be further discussed in Sec. 3.1.

The outcomes of graph pattern matching can be succinctly modeled as a relation $GR^\mathcal{P}_G$, or more compactly $GR^\mathcal{P}$ in clear contexts, defined over the schema $S = V_\mathcal{P} \cup E_\mathcal{P}$. Here, the sets V_G and E_G serve as the respective domains for the vertices and edges identified through the matching process. Within this framework, we refer to such a relation as a *Graph Relation*, a construct where all attributes are derived from the domain of a property graph. It is essential to recognize that any property graph G can be conceptualized as a graph relation GR^G, represented by a singular tuple that collectively encompasses all of its vertices and edges. Throughout this paper, we treat the notions of a property graph and a tuple of graph relation as essentially interchangeable terms. In alignment with this perspective, we elaborate on the *Matching* operator as follows.

Definition 1 (Matching Operator, $\mathcal{M}$). *The Matching Operator, denoted as $\mathcal{M}$, is designed to perform graph pattern matching on a given graph relation GR against a specified pattern graph $\mathcal{P}$. For each graph instance g in GR, $\mathcal{M}$ identifies all subgraphs of g that are homomorphic to $\mathcal{P}$, and subsequently, aggregates these mappings to construct a comprehensive graph relation. The operation of the matching Operator can be formally articulated as $\mathcal{M}(GR, \mathcal{P}) = \bigcup_{g \in GR} GR^\mathcal{P}_g$.*

Example 3. *Let G denote the property graph derived from the relations via* RGMapping *in Example 2. Given a pattern graph $\mathcal{P}$ in Fig. 2(b), the results of graph pattern matching are subgraphs of G that are homomorphic to $\mathcal{P}$, represented as a graph relation $GR^\mathcal{P} = \mathcal{M}(GR^G, \mathcal{P})$, each tuple corresponds to one matched subgraph.*

This definition ensures that the matching operator is inherently closed regarding graph relations, which adheres to the language opportunities of "nested matching" (specified as PGQ-079) in SQL/PGQ [40]. In this paper, we only handle cases where G represents the entire property graph, and thereafter simplify the matching operator notation to $\mathcal{M}(\mathcal{P})$ when the context is clear.

2.3 Problem Definition

To study relational query optimization, it is common to focus on SPJ queries, which consists of three most frequently used operations: select, project, and (natural) join. These operations form the backbone of many relational queries. Given a set of relations $R_1, R_2, \ldots, R_m$, an SPJ query is formally represented as:

$$Q = \pi_A(\sigma_\Psi(R_1 \bowtie \cdots \bowtie R_m)).$$

Inspired from the SPJ paradigm, we introduce a novel category of queries, termed SPJM queries, to logically formulate SQL/PGQ [40] queries that blend relational and graph operations. The SPJM framework augments SPJ queries by incorporating a matching operator to enrich the query's expressive power, to seamlessly navigate both relational and graph data domains. Given the set of relations and a property graph G constructed from these relations via an RGMapping, an SPJM query is articulated as:

$$Q = \pi_A(\sigma_\Psi(R_1 \bowtie \cdots \bowtie R_m \bowtie (\widehat{\pi}_{A*}\mathcal{M}_G(\mathcal{P})))) \tag{1}$$

In this formulation, $\widehat{\pi}_{A*}\mathcal{M}_G(\mathcal{P})$ is the *graph component* of the query, while the remaining part of the query is an SPJ expression referred to as the *relational component*. Here, $\mathcal{M}_G(\mathcal{P})$ represents the process of matching the pattern $\mathcal{P}$ on the graph G and returns a graph relation as defined in Def. 1. The operator $\widehat{\pi}_{A*}$ is a graph-calibrated projection operator that extracts the ID, label, and other attributes from the vertices and edges in the matched results. This process helps "flatten" graph elements into relational tuples. For example, given a graph relation GR that contains a vertex of {ID:0, label:Person, name:"Tom"}, the projection $\widehat{\pi}_{\mathrm{id}(v)\rightarrow \mathrm{v\_id}, \ell(v)\rightarrow \mathrm{v\_label}, v.name\rightarrow \mathrm{v\_name}}(GR)$ turns the vertex into a relational tuple of (0, Person, "Tom"). The projection is designed to reflect the `COLUMNS` clause in SQL/PGQ to retrieve specific attributes from vertices and edges as required. For simplicity, we assume that all attributes are extracted unless otherwise specified.

In this paper, we study the problem of optimizing SPJM queries in Eq. 1. Fig. 2(c) illustrates the SPJM query skeleton corresponding to the SQL/PGQ query in Example 1.

3 Optimizing Matching Operator

In this section, we focus on handling the matching operator, which plays a distinct role within the SPJM queries. We discuss two main perspectives of optimizing the matching operator: logical transformation and physical implementation. Logical transformation is responsible for transforming a matching operator into a logically equivalent representation, while physical implementation focuses on how the matching operator can be efficiently executed.

3.1 Logical Transformation

We commence with an intuitive, graph-agnostic transformation before introducing a graph-aware technique grounded on the concept of decomposition tree, which is the key to the optimization of graph pattern matching in the literature [29, 51].

Before proceeding, we introduce the concept of pattern decomposition that decomposes $\mathcal{P}$ into two overlapping patterns, $\mathcal{P}_1$ and $\mathcal{P}_2$, with shared vertices $V_o = V_{\mathcal{P}_1} \cap V_{\mathcal{P}_2}$ and shared edges $E_o = E_{\mathcal{P}_1} \cap E_{\mathcal{P}_2}$. Denote $\mathcal{P} = \mathcal{P}_1 \cup \mathcal{P}_2$. Under the homomorphism semantics, the matching of $\mathcal{P}$ can be represented as:

$$\mathcal{M}(\mathcal{P}) = \mathcal{M}(\mathcal{P}_1) \widehat{\bowtie}_{V_o, E_o} \mathcal{M}(\mathcal{P}_2), \tag{2}$$

where $\widehat{\bowtie}$ is a natural join operator for joining two graph relations based on the common vertices and edges. Note that Eq. 2 is also applicable to alternative semantics, including isomorphism and non-repeated-edge [6]. To support these semantics, a special all-distinct operator can be applied as a filter to remove results that contain duplicate vertices and/or edges. The adoption of the all-distinct operator is compatible with all techniques in this paper.

3.1.1 Graph-agnostic Transformation. If the matching operator can be transformed into purely relational operations, the SPJM query becomes a standard SPJ query, which can then be optimized using existing relational optimizers (Sec. 4.1). This graph-agnostic approach is intuitive and easy to implement on top of existing relational databases, making it a straightforward choice in prototyped systems [1, 47, 48]. However, there is no theoretical guarantee that such a transformation is lossless in the context of RGMapping. In this subsection, we bridge this gap by demonstrating the lossless transformation of the matching operator under RGMapping.

Consider a pattern graph $\mathcal{P}$ and one of its edges $e = (u_s, u_t)$. According to the definition of the matching operator (Sec. 2.2), the graph edges and vertices that can be matched with e must have the labels $\ell(e)$, $\ell(u_s)$, and $\ell(u_t)$. We further denote the relations corresponding to these edges and vertices via RGMapping as $R_{\ell(e)}$, $R_{\ell(u_s)}$, and $R_{\ell(u_t)}$, respectively. Moreover, there must be total functions $\lambda^s_{\ell(e)}$ and $\lambda^t_{\ell(e)}$ for mapping tuples from $R_{\ell(e)}$ to $R_{\ell(u_s)}$ and $R_{\ell(u_t)}$, respectively. We define the following EVJoin relational operation regarding $\lambda^s_{\ell(e)}$ as:

$$\begin{aligned} R_{\ell(e)} \bowtie_{\epsilon v} R_{\ell(u_s)} = \{(\tau_e, \tau_s) \mid \\ \tau_e \in R_{\ell(e)} \land \tau_s \in R_{\ell(u_s)} \land \lambda^s_{\ell(e)}(\tau_e) = \tau_s\}. \end{aligned} \tag{3}$$

The EVJoin regarding $\lambda^t_{\ell(e)}$ is defined analogously. Although called EVJoin, the operation is associative like any relation join, meaning that the order in which the edge and vertex relations are joined does not affect the final result.

We have the following lemma.

Lemma 1. *Under* RGMapping, *the matching operation in an* SPJM *query can be losslessly transformed into a sequence of relational joins involving n vertex relations and m edge relations.*

Proof. Consider a pattern $\mathcal{P}_m$ of m edges, where the i-th vertex is denoted as u_i, and the i-th edge is $e_i = (u_{s_i}, u_{t_i})$.

The proof proceeds by induction, starting with a pattern graph $\mathcal{P}_0$ with a single vertex only. It is clear that $\mathcal{M}(\mathcal{P}_0)$ yields a subset of vertices with label $\ell(u_0)$, which is mapped from the relation $R_{\ell(u_0)}$ via RGMapping. As a result, we have $R_0 = \widehat{\pi}_{A*}(\mathcal{M}(\mathcal{P}_0)) = R_{\ell(u_0)}$.

Next, consider $\mathcal{P}_1$ with one edge, $e_1 = (u_{s_1}, u_{t_1})$. Matching $\mathcal{P}_1$ is equivalent to retrieving the edge relation, together with the corresponding source and target vertices. Therefore, we have:

$$R_1 = \widehat{\pi}_{A*}(M(\mathcal{P}_1)) = R_{\ell(u_{s_1})} \bowtie_{\epsilon v} R_{\ell(e_1)} \bowtie_{\epsilon v} R_{\ell(u_{t_1})}$$

Assume that when $m = k-1$, $\widehat{\pi}_{A*}(\mathcal{M}(\mathcal{P}_{k-1}))$ can be converted to a sequence of relational operators, resulting in R_{k-1}. When $m = k$, we consider $\mathcal{P}_k$ of k edges constructed from $\mathcal{P}_{k-1}$ by adding edge $e_k = (u_{s_k}, u_{t_k})$. For $\mathcal{P}_k$ to be connected, it must share at least one common vertex V_o with $\mathcal{P}_{k-1}$. According to Eq. 2, we have:

$$\mathcal{M}(\mathcal{P}_k) = \mathcal{M}(\mathcal{P}_{e_k}) \widehat{\bowtie}_{V_o} \mathcal{M}(\mathcal{P}_{k-1}),$$

where $\mathcal{P}_{e_k}$ denotes a pattern that contains only the edge e_k, and V_o is the common vertex shared by $\mathcal{P}_{k-1}$ and $\mathcal{P}_{e_k}$. Applying $\widehat{\pi}_{A*}$ to the above equation, we get:

$$\begin{aligned} R_k &= \widehat{\pi}_{A*}(\mathcal{M}(\mathcal{P}_k)) \\ &= \widehat{\pi}_{A_1*}(\mathcal{M}(\mathcal{P}_{e_k})) \bowtie_{V_o.attr} \widehat{\pi}_{A_2*}(\mathcal{M}(\mathcal{P}_{k-1})) \\ &= R_{\ell(u_{s_k})} \bowtie_{\epsilon v} R_{\ell(e_k)} \bowtie_{\epsilon v} R_{\ell(u_{t_k})} \bowtie_{V_o.attr} R_{k-1} \end{aligned}$$

By induction, denoting $R'_i = R_{\ell(u_{s_i})} \bowtie_{\epsilon v} R_{\ell(e_i)} \bowtie_{\epsilon v} R_{\ell(u_{t_i})}$, we have the matching operator losslessly converted to a sequence of relational join operations:

$$\widehat{\pi}_{A*}(\mathcal{M}(\mathcal{P}_k)) = R'_k \bowtie R'_{k-1} \bowtie \cdots \bowtie R'_1 \bowtie R_0. \tag{4}$$

We thus conclude the proof. □

Example 4. *Given pattern graph $\mathcal{P}$ in Fig. 2(b), the matching operation $\mathcal{M}(\mathcal{P})$ can be converted to a sequence of join operations as follows. Without loss of generality, we start from $\mathcal{P}_0$ containing only the vertex u_{p_1}, and we have $R_0 = R^1_{Person}$ (note that the superscript 1 is used to differentiate relations of* *the same name). Next, we sequentially add the edges $e_1 = (u_{p_1}, u_{p_2})$, $e_2 = (u_{p_1}, u_m)$, and $e_3 = (u_{p_2}, u_m)$ to $\mathcal{P}_0$, resulting in the following relations:*

$$R'_1 = R^1_{Person} \bowtie_{person\_id=pid1} R_{Knows} \bowtie_{pid2=person\_id} R^2_{Person},$$
$$R'_2 = R^1_{Person} \bowtie_{person\_id=pid} R^1_{Likes} \bowtie_{mid=message\_id} R_{Message},$$
$$R'_3 = R^2_{Person} \bowtie_{person\_id=pid} R^2_{Likes} \bowtie_{mid=message\_id} R_{Message}.$$

Finally, we have $\widehat{\pi}_{A*}(\mathcal{M}(\mathcal{P})) = R'_3 \bowtie R'_2 \bowtie R'_1 \bowtie R_0$. Note that R^1_{Person} in R'_2, as well as R^2_{Person} and $R_{Message}$ in R'_3, are redundant and can be removed from the final join. By eliminating them, we obtain a sequence of joins with 3 vertex relations and 3 edge relations.

3.1.2 Graph-aware Transformation. We introduce a graph-aware transformation that incorporates key ideas from the literature on graph optimization. Following Eq. 2, we can recursively decompose $\mathcal{P}$, forming a tree structure called the *decomposition tree*. The tree has a root node that represents $\mathcal{P}$, and each non-leaf *intermediate* node is a sub-pattern (a subgraph of the pattern) $\mathcal{P}' \subset \mathcal{P}$, which has a left and right child node, denoted as $\mathcal{P}'_l$ and $\mathcal{P}'_r$, respectively. The leaf nodes of the tree are called *Minimum Matching Components* (MMC), correspond to indivisible patterns directly solvable with specific physical operations as will be introduced in Sec. 3.2. The decomposition tree naturally forms a logical plan for solving $\mathcal{M}(\mathcal{P})$, as demonstrated in Fig. 3. For any non-leaf node $\mathcal{P}'$, there exists a relationship $\mathcal{M}(\mathcal{P}') = \mathcal{M}(\mathcal{P}'_l)\widehat{\bowtie}\mathcal{M}(\mathcal{P}'_r)$ according to Eq. 2. The plan allows for the recursive computation of the entire pattern.

Following state-of-the-art graph optimizers [29, 51], to guarantee a *worst-case optimal* execution plan [39], all intermediate sub-patterns in the decomposition tree must be induced subgraphs of $\mathcal{P}$. Furthermore, MMC is restricted to be a single-vertex pattern and a *complete star*. A star-shaped pattern is denoted as $\mathcal{P}(u; V_s)$, where u is the root vertex and V_s is the set of leaf vertices[1]. In the decomposition tree, given $\mathcal{P}' = \mathcal{P}'' \cup \mathcal{P}(u; V_s)$, $\mathcal{P}(u; V_s)$ is a complete star if and only if it is a right child and $V_s \subseteq V_{\mathcal{P}''}$, meaning that the leaf vertices of the complete star must all be common vertices for the decomposition. A single-edge pattern is a special case of a complete star. The complete star logically represents the physical operations of `EXPAND_INTERSECT`, which will be discussed in Sec. 3.2. As shown in Fig. 3, a single-edge pattern, such as $\mathcal{P}_3$, is further decomposed into a single-vertex pattern and the pattern itself, allowing the optimizer to select from which vertex the edge can be expanded. The intermediate sub-patterns pruned from the decomposition tree are also presented in Fig. 3. Some previous studies, such as EmptyHeaded [4] and CLFTJ [24], have also explored decomposition trees. However, our method significantly differs from theirs. Specifically, in these previous methods, the tree nodes represent sets of relations, and the edges in the decomposition trees connect nodes with common join keys. In contrast, the nodes in our decomposition trees represent sub-patterns (relations that can form a graph after RGMapping) of queries. Each edge in our tree connects two nodes such that the child sub-pattern can be computed from the parent sub-pattern in some execution plan.

Remark 1. *The graph-aware transformation is fundamentally different from its graph-agnostic counterpart. While the graph-agnostic approach consistently converts pattern matching operations into relational joins between vertex and edge relations, the graph-aware transformation does not, due to the constraints imposed by pattern decomposition. While the graph-agnostic approach is straightforward, it has the following drawbacks:*

***Graph-unaware Join Order**: It may lead the relational optimizer to reorder the join of vertex and edge relations, potentially missing chances to use graph indexes for efficiently computing adjacent edges and vertices, as discussed in Sec. 3.2.1.*

[1] Edge directions between u and V_s are not important, and we assume they all point from u to V_s.

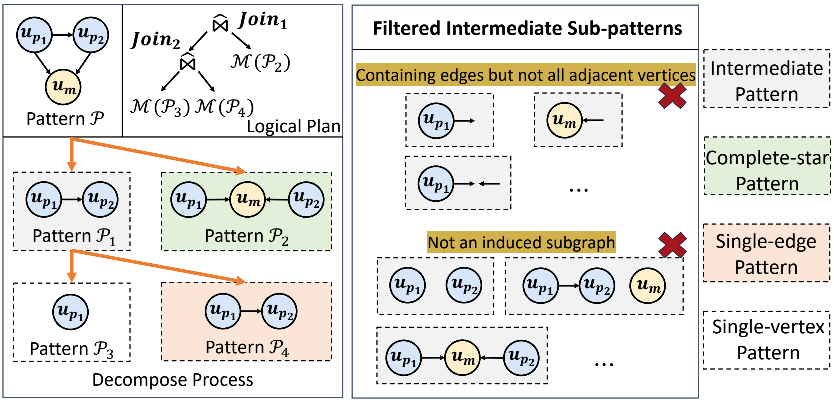
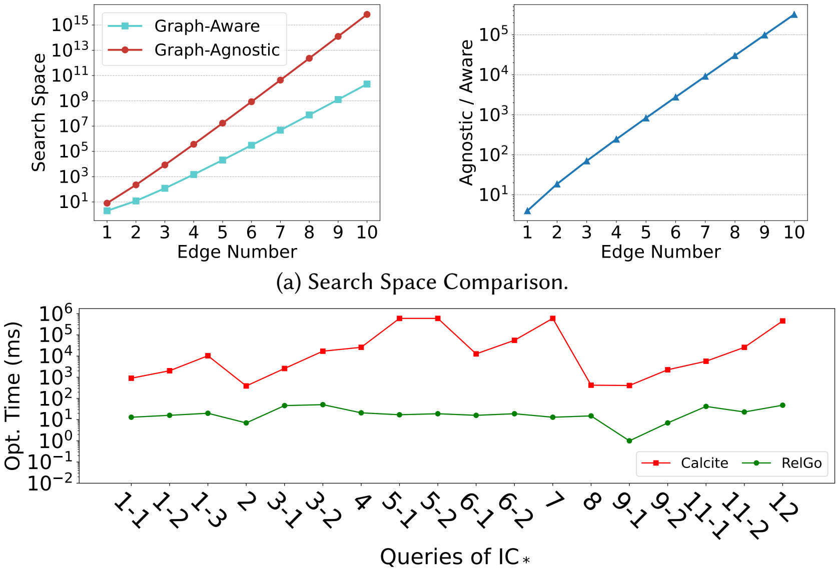


Fig. 3. Example of decomposition trees and the corresponding logical plans. Note that sub-pattern $\mathcal{P}_2$ can be a leaf node, but it cannot be an intermediate node.



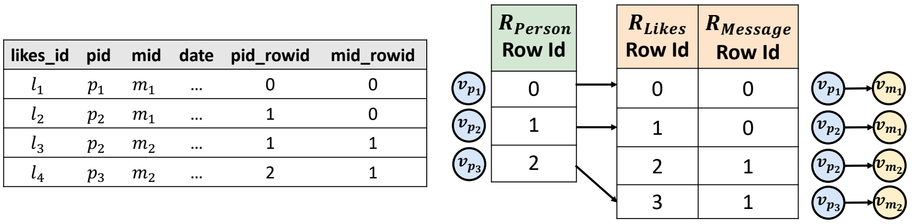
(a) Search Space Comparison.

(b) Optimization Time Cost on LDBC Queries.

Fig. 4. Compare the search space and optimization time.

***Suboptimal Join Plans**: It generates plans that consistently reflect edge-based join plans that have been shown to be suboptimal in terms of worst-case performance [27].*
***Increased Search Space**: Compared to the graph-aware transformation, it can lead to an exponentially larger search space when computing optimal plans, which will be discussed in the following.*

3.1.3 The Search Space: Graph-agnostic vs Graph-aware. After applying graph-agnostic transformations to the matching operator, the optimizer searches for the optimal join order. In contrast, applying graph-aware transformations leads to a search for the optimal decomposition tree. The search space for the graph-agnostic approach is clearly larger than that of the graph-aware approach, given the constraints imposed on the decomposition tree in the latter approach. However, the precise difference in search space complexity between the two approaches has not been rigorously



likes_id	pid	mid	date	pid_rowid	mid_rowid
l_1	p_1	m_1	...	0	0
l_2	p_2	m_1	...	1	0
l_3	p_2	m_2	...	1	1
l_4	p_3	m_2	...	2	1

(a) The EV-Index on R_{Likes}

	R_{Person} Row Id
v_{p_1}	0
v_{p_2}	1
v_{p_3}	2

R_{Likes} Row Id	$R_{Message}$ Row Id	
0	0	$v_{p_1} \rightarrow v_{m_1}$
1	0	$v_{p_2} \rightarrow v_{m_1}$
2	1	$v_{p_2} \rightarrow v_{m_2}$
3	1	$v_{p_3} \rightarrow v_{m_2}$

(b) The VE-Index on R_{Person} in the CSR Format

Fig. 5. The graph index constructed among relations R_{Person}, R_{Likes} and R_{Message} in Fig. 2(a).

analyzed. In this subsection, we analyze the gap between the two search spaces and conclude that the graph-aware approach can be exponentially more efficient in this regard.

Theorem 1. *The search space in graph-aware transformation can be exponentially smaller than that of the graph-agnostic transformation, for optimizing the matching operator in an* SPJM *query.*

3.1.4 Comparison of Search Space and Optimization Time. To further illustrate Theorem 1, we used a special case of a path graph to compare the search spaces directly. We conducted a micro-benchmark experiment using a path graph with m edges, programming an enumerator to explore the search space of both graph-agnostic and graph-aware approaches while varying m. The results, shown in Fig. 4a, confirm the significant difference in search space size between the two approaches.

Additionally, we compared the optimizer's query optimization time. In our comparison, Apache Calcite, a generic relational optimization framework, served as the optimizer for the graph-agnostic method. In contrast, our RelGo, implemented based on Calcite, acts as the optimizer for the graph-aware method. Both RelGo and Calcite are implemented in Java, utilizing the VolcanoPlanner of Calcite with default rules. Notably, we did not consider aggressive pruning rules as used in commercialized database like DuckDB [2] for either Calcite or RelGo, providing a fair comparison and a clear demonstration of the reduced search space. The optimization time was evaluated using the queries in our experiment (details in Sec. 5). Optimizations that do not complete within 10 minutes are recorded as taking 10 minutes. Since Calcite often exceeds the 10-minute limit on JOB queries[35], we only report the results on LDBC queries. The results in Fig. 4b indicate that RelGo can complete optimizing almost all queries within 10-100 milliseconds. Besides, the results demonstrate RelGo's significant superiority over Calcite in query optimization speed. For instance, on IC_{5-1}, the optimization time using RelGo is more than 10^4 times faster compared to Calcite.

3.2 Physical Implementation

In the graph view, given a vertex v, it is efficient to obtain its adjacent edges and vertices (i.e., neighbors). However, in the relational view, such adjacency relationships between vertices and edges are not directly stored in relations but must be computed via the EVJoin operations (Eq. 3). While there are multiple ways to construct the graph view in the literature [20, 45], we refer to the method introduced in GRainDB [23], which is free from materializing the graph. This approach avoids the extra storage cost associated with graph materialization and ensures compatibility with the relational context, Specifically, GRainDB introduces an indexing technique called pre-defined join to improve the performance of join operations. As the pre-defined join essentially materializes the adjacency relationships, we treat it as a *graph index* in this work.

3.2.1 Graph Index. As shown in Fig. 5, given the three relations R_{Person}, R_{Likes}, and R_{Message}, the complete information of "Person likes messages" can be obtained by conducting the join:

$$R_{\text{Person}} \bowtie_{\text{person\_id = pid}} R_{\text{Likes}} \bowtie_{\text{mid = message\_id}} R_{\text{Message}}.$$

GRainDB introduces two kinds of indexes to the relational tables to efficiently process the join: the EV-index and the VE-index. The EV-index, shown in Fig. 5(a), is constructed by appending extra columns to the table R_{Likes}. The column "pid_rowid" stores the row ID of the corresponding tuple in the table R_{Person}, denoted as $\mathsf{rid}(\tau_p)$, where $\tau_p \in R_{\text{Person}}$. Similarly, the column "mid_rowid" stores the row ID of the corresponding tuple in the table R_{Message}, denoted as $\mathsf{rid}(\tau_m)$, where $\tau_m \in R_{\text{Message}}$. These row ids help quickly route a tuple $\tau_l \in R_{\text{Likes}}$ to the joinable tuples τ_p and τ_m without additional operations like hash-table lookup or sorting.

The VE-index in Fig. 5(b) is created on R_{Person} for efficiently computing its "liked messages". For each tuple $\tau_p \in R_{\text{Person}}$, the VE-index records the row ids of $\tau_l \in R_{\text{Likes}}$ and the corresponding $\tau_m \in R_{\text{Message}}$ that are joinable with τ_p. In the graph view, treating "Person-[Likes]->Messages" as an edge of a property graph, the VE-index maintains the adjacent edges and vertices of each person.

We can adopt GRainDB's approach to construct the graph indexes during the RGMapping process. Given an edge relation R_e and its associated vertex relations R_{v_s} and R_{v_t}, the EV-index can be constructed on R_e for each tuple $\tau_e \in R_e$ by including $\mathsf{rid}(\lambda_e^s(\tau_e))$ and $\mathsf{rid}(\lambda_e^t(\tau_e))$, which are the row ids of the corresponding tuples in R_{v_s} and R_{v_t}, respectively. Meanwhile, the VE-index can be constructed on R_{v_s} for each tuple $\tau_{v_s} \in R_{v_s}$ by including the row ids of all tuples $\tau_e \in R_e$ such that $\lambda_e^s(\tau_e) = \tau_{v_s}$, along with the row ids of the corresponding tuples $\tau_{v_t} \in R_{v_t}$ such that $\lambda_e^t(\tau_e) = \tau_{v_t}$. The construction of VE-index on R_{v_t} is analogous.

3.2.2 The Graph-Aware Execution Plan. We delve into the physical implementation of the execution plan provided by the graph-aware method for solving $\mathcal{M}(\mathcal{P})$. The entry point of the plan is always matching a single-vertex pattern $\mathcal{P}_u$, which is one of the leaf nodes in the decomposition tree.

The implementation of $\mathcal{M}(\mathcal{P}_u)$ is straightforward: scanning the corresponding vertex relation $R_{\ell(u)}$ and encoding each tuple as a graph vertex object that contains its ID, label (mandatory) and necessary attributes. The row ID of the tuple in the relation can be directly used as the ID. To ensure globally uniqueness, the name of the relation can be incorporated as a prefix of the ID. Advanced encoding techniques are necessary for production use, but they are beyond the scope of this paper.

The plan is then constructed in a bottom-up manner. As shown in Fig. 3, there are three fundamental cases to consider when implementing the plan.

Case I: Solving $\mathcal{M}(\mathcal{P}') = \mathcal{M}(\mathcal{P}'_l) \widehat{\bowtie}_{V_o,E_o} \mathcal{M}(\mathcal{P}'_r)$, where $\mathcal{P}'_l$ and $\mathcal{P}'_r$ are both intermediate patterns in the decomposition tree. The implementation of such a join is similar to a conventional relational join. The join is constrained to a natural join, where the join condition is simply the equality of the common vertices V_o and edges E_o between $\mathcal{P}'_l$ and $\mathcal{P}'_r$. During the implementation of the join, the identifiers of the vertices and edges can serve as the keys for comparison. Note that the input and output of the join are both graph relations, which will not be projected into relational tuples until the last stage that obtains the results $\mathcal{M}(\mathcal{P})$.

Case II: Solving $\mathcal{M}(\mathcal{P}') = \mathcal{M}(\mathcal{P}'_l) \widehat{\bowtie}_{u_s} \mathcal{M}(\mathcal{P}_e)$, where $\mathcal{P}_e$ is a single-edge pattern, and u_s is the source vertex in $\mathcal{P}'_l$ from which the edge $e = (u_s, u_t)$ is expanded. Note that it's not possible for both u_s and u_t to be in $\mathcal{P}'_l$, as it would violate the fact that $\mathcal{P}'_l$ is either a single vertex or an induced sub-pattern.

When there is no graph index, $\mathcal{M}(\mathcal{P}_e)$ is computed via $R_{\ell(u_s)} \bowtie_{\epsilon\nu} R_{\ell(e)} \bowtie_{\epsilon\nu} R_{\ell(u_t)}$. This case is then reduced to Case I.

When graph indexes exist, the implementation is handled by the physical operators of `EXPAND_EDGE` and `GET_VERTEX`. For each tuple $\tau \in \mathcal{M}(\mathcal{P}'_l)$, $\tau.u_s$ must record a graph vertex v_s that matches u_s in the pattern $\mathcal{P}'_l$. The `EXPAND_EDGE` operator looks up the VE-index of v_s, which allows it to efficiently computes v_s's adjacent edges (more precisely, it's the corresponding edge tuples). Furthermore, the `GET_VERTEX` operator is used to obtain the matched vertex v_t that is connected to v_s via the previous matched edges, which can be achieved by looking up the EV-index of the matched edges.

By combining the results of EXPAND_EDGE and GET_VERTEX, the tuple of $(\tau, \mathcal{N}^E(v_s), \mathcal{N}(v_s))$ is rendered. For example, in Fig. 5(b), if we apply EXPAND_EDGE and GET_VERTEX to a tuple τ from v_{p_2}, the result $(\tau, [e_{l_2}, e_{l_3}], [v_{m_1}, v_{m_2}])$ is returned. Furthermore, to obtain $\mathcal{M}(\mathcal{P}')$, we flatten the adjacent edges and vertices and pair them up. In the case of $(\tau, [e_{l_2}, e_{l_3}], [v_{m_1}, v_{m_2}])$, two tuples (τ, e_{l_2}, v_{m_1}) and (τ, e_{l_3}, v_{m_2}) are generated.

In practice, a vertex may be adjacent to multiple types of edges. For example, in Fig. 2, a Person vertex can be connected to both Likes and Knows edges. To handle such cases, we can record edge's ID instead of just the row ID of the tuple. Given that the edge's ID is a combination of its label and the tuple's row ID, the adjacent edges of a specific label can be easily obtained from the VE-Index.

Case III: Solving $\mathcal{M}(\mathcal{P}') = \mathcal{M}(\mathcal{P}'_l)\widehat{\bowtie}_{V_s,E_s}\mathcal{M}(\mathcal{P}(u; V_s))$, where pattern $\mathcal{P}(u; V_s)$ is a complete k-star with $V_s = \{u_1, \ldots, u_k\}$.

When there is no graph index, solving Case III involves continuously joining $|V_s|$ single-edge patterns. When graph indexes are available, the EXPAND_INTERSECT operator can be used to efficiently compute the join. Unlike HUGE [51], which has a graph storage that naturally supports EXPAND_INTERSECT, we have implemented this operator directly on a relational database. Given a tuple $\tau \in \mathcal{M}(\mathcal{P}'_l)$, let $\{v_1, \ldots, v_k\}$ be the vertices in τ that match the leaf vertices $\{u_1, \ldots, u_k\}$ in the complete star $\mathcal{P}(u; V_s)$. Vertices matching the root vertex u of the star must be common neighbors of all the leaf vertices.

Consequently, for the tuple τ, the physical EXPAND_INTERSECT operator performs the following steps:

(1) For each leaf vertex $u_i \in V_s$ $(1 \leq i \leq k)$, apply the EXPAND_EDGE and GET_VERTEX operators to obtain the adjacent edges and neighbors of the corresponding vertices v_i respectively.
(2) Compute the intersections of all adjacent edges and neighbors returned by the EXPAND_EDGE and GET_VERTEX operators.
(3) Return a new tuple as follows; for the sake of simplicity, the details of the edges are omitted: $(\tau, \bigcap_{1 \leq i \leq k} \mathcal{N}(v_i))$.

Note that the above step (1) and (2) can be computed in a pipeline manner, following a certain order of among the leaf vertices. Similar to Case II, we flatten the common edges and vertices and pair them up to obtain the final result.

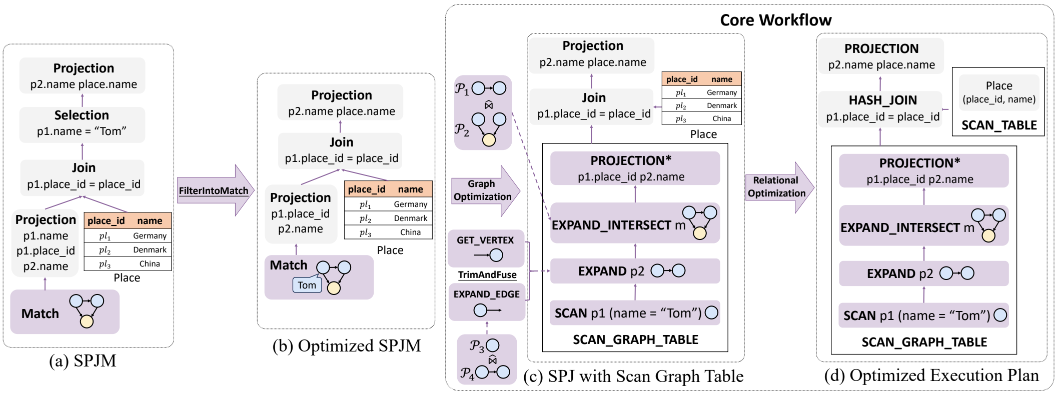
Example 5. *Given $\mathcal{P}$ in Fig. 3, a decomposition tree and its corresponding logical plan are presented. We illustrate the physical implementation of $\mathcal{M}(\mathcal{P}_1)\widehat{\bowtie}\mathcal{M}(\mathcal{P}_2)$ using* EXPAND_INTERSECT*when a graph index is available. Consider the tuple $(v_{p_1}, e_{k_1}, v_{p_2})$ from $\mathcal{M}(\mathcal{P}_1)$ as an example. First, the* EXPAND_EDGE *and* GET_VERTEX *operators are applied to obtain the adjacent edges and neighbors of v_{p_1} and v_{p_2}, resulting in*

$$(v_{p_1}, e_{k_1}, v_{p_2}, [e_{l_1}], [v_{m_1}]) \text{ and } (v_{p_1}, e_{k_1}, v_{p_2}, [e_{l_2}, e_{l_3}], [v_{m_1}, v_{m_2}]).$$

Next, the intersection process is conducted. Since $\mathcal{N}(v_{p_1}) \cap \mathcal{N}(v_{p_2}) = [v_{m_1}]$, the edges in both sets that have v_{m_1} as the target vertex are retained, resulting in $(v_{p_1}, e_{k_1}, v_{p_2}, [(e_{l_1}, e_{l_2}, v_{m_1})])$. Finally, the tuple is flattened to $(v_{p_1}, e_{k_1}, v_{p_2}, e_{l_1}, e_{l_2}, v_{m_1})$.

4 The Converged Optimization Framework

This section presents RelGo, a converged relational/graph optimization framework designed to optimize the query processing of SPJM queries. We begin by introducing a naive solution built upon the graph-agnostic method for solving the matching operator. We then delve into the converged workflow of RelGo, which leverages the graph-aware method for solving the matching operator



(a) SPJM (b) Optimized SPJM (c) SPJ with Scan Graph Table (d) Optimized Execution Plan

Fig. 6. The converged optimization workflow

and introduces a complete workflow that aims to integrate techniques from both relational and graph optimization modules.

4.1 Graph-Agnostic Approach

The graph-agnostic approach is straightforward: it applies the graph-agnostic transformation for the matching operator in an SPJM query into a series of relational operations (Lemma 1), effectively converting the SPJM query into an SPJ query. The resulting SPJ query can then be optimized by any existing relational optimizer, producing an execution plan. As an improvement, if a graph index (Sec. 3.2.1) is available, certain hash-join operators in the execution plan can be replaced by the predefined-join operator, as discussed in GRainDB [23]. The main advantage of this solution is its easy integration with any existing relational database. However, it suffers from two significant drawbacks discussed in Remark 1.

4.2 The Converged Approach

As illustrated in Fig. 6, the core workflow of the RelGo framework consists of two components: *graph optimization* and *relational optimization*. The graph optimization is responsible for handling the graph component in an SPJM query, leveraging graph optimization techniques to determine the optimal decomposition tree of the matching operator. On the other hand, the relational optimization takes over to optimize the relational component in the query. The order in which these two components are applied is not strictly defined. However, for the purpose of our discussion, we will first focus on the graph optimization and then proceed to the relational optimization. In addition to the core workflow, we further explore heuristic rules that highlight the non-trivial interplay between the relational and graph components in an SPJM query.

4.2.1 The Graph Optimization. We adopt the graph optimization techniques developed in GLogS [29]. However, it is crucial to note that GLogS was originally designed for native graph data, whereas our framework deals with relational data, which necessitates a careful adaptation of GLogS's techniques to the relational setting.

GLogue Construction. GLogS is built upon a data structure called GLogue, which is essentially a graph $\mathbb{G}_{\mathcal{P}}(V, E)$. In this graph, each vertex represents a pattern $\mathcal{P}'$ consisting of up to k vertices (typically, $k = 3$) that has non-empty matched instances in the original graph. There is an edge from $\mathcal{P}''$ to $\mathcal{P}'$, if there is a decomposition tree where $\mathcal{P}''$ is a child node of $\mathcal{P}'$.

Each vertex $\mathcal{P}'$ in GLogue maintains $|\mathcal{M}(\mathcal{P}')|$, denoting the cardinality of the pattern. To reduce computation costs, GLogS employs a sparsification technique to construct a subgraph G'. The pattern cardinality can then be estimated using $|\mathcal{M}_{G'}(\mathcal{P}')|$ based on subgraph G'. In our work, we adapt this sparsification technique to construct GLogue. We sample a subset of vertex and edge relations in the RGMapping process. Once the subset of relations is obtained, they can serve as the input tables to the techniques presented in [45] for constructing the sparsified graph G'.

Cost Calculation. The optimization process is essentially searching for the execution plan that incurs the minimal cost. Let the cost of an execution plan Φ for computing $\mathcal{M}(\mathcal{P})$ be $\text{Cost}_\Phi(\mathcal{P})$.

Consider $\mathcal{M}(\mathcal{P}') = \mathcal{M}(\mathcal{P}'_l)\widehat{\bowtie}\mathcal{M}(\mathcal{P}'_r)$ as an intermediate computation in an execution plan. We have:

$$\text{Cost}_\Phi(\mathcal{P}') = \text{Cost}_{\Phi_l}(\mathcal{P}'_l) + \text{Cost}_{\Phi_r}(\mathcal{P}'_r) + \text{Cost}(\widehat{\bowtie}),$$

where Φ_l and Φ_r are the execution plans for computing $\mathcal{M}(\mathcal{P}'_l)$ and $\mathcal{M}(\mathcal{P}'_r)$, respectively, and $\text{Cost}(\widehat{\bowtie})$ is the cost of the join operation.

When a graph index is available, there are three physical implementations of $\widehat{\bowtie}$, depending on the type of $\mathcal{P}'_r$, and the calculation of $\text{Cost}(\widehat{\bowtie})$ differs accordingly:

- If $\mathcal{P}'_r$ is a single-edge pattern, $\widehat{\bowtie}$ is implemented using the EXPAND_EDGE operator followed by GET_VERTEX. The cost is calculated based on the cardinality of $\mathcal{M}(\mathcal{P}'_l)$ (can be looked up in the GLogue) and the average degree of the graph, namely $|\mathcal{M}(\mathcal{P}'_l)| \times \overline{d}$.
- If $\mathcal{P}'_r$ is a complete star pattern, $\widehat{\bowtie}$ is implemented using the EXPAND_INTERSECT operator. The cost is calculated based on the cardinality of $\mathcal{M}(\mathcal{P}'_l)$ and the average intersection size of the neighbors of the vertices being intersected, which is maintained on the corresponding edge from P' to $\mathcal{P}'_l$ in GLogue.
- If $\mathcal{P}'_r$ is any arbitrary pattern, $\widehat{\bowtie}$ is implemented as a HASH_JOIN. The cost is calculated as the product of the cardinalities of the two relations being joined, i.e., $\text{Cost}(\widehat{\bowtie}) = |\mathcal{M}(\mathcal{P}'_l)| \times |\mathcal{M}(\mathcal{P}'_r)|$.

In the absence of a graph index, HASH_JOIN is used for the entire plan of the matching operator for simplicity, and its cost is computed as the product of the cardinalities of the two relations being joined. Although other physical join implementations, such as nested loop join, may be more effective if the join condition is not selective, considering these alternatives is planned for future work.

Plan Computation. Searching for the optimal execution plan in RelGo remains the same as in GLogS. The optimal plan is obtained by searching for the shortest path in the GLogue from the single-vertex pattern to the queried pattern. Fig. 6(c) demonstrates a physical plan for matching the given triangle pattern when a graph index is present. The plan reflects the example in Example 5, with one exception: the pair of EXPAND_EDGE and GET_VERTEX operators is fused into a single EXPAND operator, which will be discussed as a heuristic rule called TrimAndFuseRule.

4.2.2 The Relational Optimization. Once the graph optimizer has computed the optimal execution plan for $\mathcal{M}(\mathcal{P})$, the next step is to integrate this plan with the remaining relational operators in the SPJM query. The relational optimization is responsible for optimizing these remaining operators, which are all relational operators. Relational optimization has evolved into a well-established field, producing numerous significant results [11, 18]. Since existing relational optimization techniques can be seamlessly integrated into RelGo, we will focus on how graph optimization techniques can be applied to enhance relational queries.

Specifically, to prevent the relational optimizer delve into the internal details of the graph pattern matching process, we introduce a new physical operator called SCAN_GRAPH_TABLE, as shown in Fig. 6(c), which encapsulates the $\widehat{\pi}_{A*}$ operator and the optimal execution plan for $\mathcal{M}(\mathcal{P})$. The SCAN_GRAPH_TABLE operator acts as a bridge between the graph and relational components of the query. From the perspective of the relational optimizer, SCAN_GRAPH_TABLE behaves like a standard SCAN operator, providing a relational interface to the matched results.

4.2.3 Heuristic Optimization Rules. In real-life use cases, heuristic rules may involve non-trivial interactions between the relational and graph components of an SPJM query. We explore two representative rules, FilterIntoMatchRule and TrimAndFuseRule, which can be applied at different stages of the optimization process to improve query performance.

FilterIntoMatchRule. To elaborate the rule, we extend the definition of a pattern $(\mathcal{P}, \Psi)$, introducing constraints within Ψ. For example, constraints can specify predicate d such as $\text{id}(v_1) = p_1$ for a vertex v_1, or $e_1.date >$ "2024-03-31" for an edge e_1. With the constraints defined, any matching result of $\mathcal{P}$ must have the corresponding vertices and edges adhering to the predicates.

While writing queries, users may not specify constraints on the pattern but rather use the selection operator after matching results have been projected into the relational relation, described as:

$$\sigma_{d'_{v_a}}(\widehat{\pi}_{v.a\to \text{v\_a},\dots}\mathcal{M}(\mathcal{P}))$$

The predicate d'_{v_a} defines a predicate in terms of an attribute of the pattern vertex that is projected by $\widehat{\pi}$ from the matched results. The motivation example in Example 1 illustrates such a case, where the selection predicate `g.p1_name = "Tom"` is applied to the pattern vertex v_{p_1}. There is wasteful computation if the selection is applied after the costly pattern matching. A more efficient approach is to push the selection predicate down into the matching operator. The FilterIntoMatchRule is formally defined as:

$$\sigma_{\Psi}(\widehat{\pi}_{v.a\to \text{v\_a},\dots}\mathcal{M}(\mathcal{P})) \equiv \sigma_{\Psi'}(\widehat{\pi}_{v.a\to \text{v\_a},\dots}\mathcal{M}((\mathcal{P}, \{d_v\}))),$$

where $\Psi' = \Psi \setminus \{d'_{v_a}\}$, and $\{d_v\}$ is the corresponding constraints that are appended to the pattern $\mathcal{P}$.

It is recommended to apply the FilterIntoMatchRule before graph optimization, as this allows the optimizer to leverage the pushed-down constraints to recalculate the cost, potentially generating more efficient execution plans. Fig. 6(b) showcases the effects of applying the FilterIntoMatchRule, where the selection predicate `g.p1_name = "Tom"` is pushed down into the matching operator.

TrimAndFuseRule. The TrimAndFuseRule is utilized to streamline a query plan by merging the `EXPAND_EDGE` and `GET_VERTEX` operators which are commonly coupled in the implementation of matching operations, into a single `EXPAND` operator that retrieves the neighboring vertices directly. However, such a fusion is permissible solely when the output edges by `EXPAND_EDGE` are deemed unnecessary, so this rule further incorporates a preceded field trim step. Specifically, the field trimmer would examine whether any subsequent relational processes rely on these edges, such as utilizing them for property projections or for filtering based on their attributes. If no such operations are found, the edges can be trimmed. Furthermore, the field trimmer would also consider a special case that the edges might be projected in the `SCAN_GRAPH_TABLE` operator as part of the matching results, but are subsequently unused in relational operations. In such cases, the edges can be trimmed as well. After the field trim step, if the output edges are trimmed, the `EXPAND_EDGE` operator can be fused with the `GET_VERTEX` operator to form a single `EXPAND` operator, which can directly retrieve the neighboring vertices efficiently by looking up the VE-index of the source vertex when the graph index is available.

Note that FilterIntoMatchRule is actually a global optimization rule because there are cases where pushing the predicate into the matching operator does not always yield better plans[35]. However, since it is mostly effective, we greedily apply FilterIntoMatchRule in the current version. A more comprehensive evaluation of this rule will be conducted in future work. On the other hand, TrimAndFuseRule is a local optimization rule specifically designed for graph optimization. The effectiveness of these two rules is validated in Sec. 5.2. Our RelGo framework is designed to be generic, allowing different optimization rules to be easily integrated.

4.3 System Implementation

We engineered the frontend of RelGo in Java and built it upon Apache Calcite [17] to utilize its robust relational query optimization infrastructure. Firstly, we enhanced Calcite's SQL parser to recognize SQL/PGQ extensions, specifically to parse the `GRAPH_TABLE` clause. We created a new `ScanGraphTableRelNode` that inherits from Calcite's core `RelNode` class, translating the `GRAPH_TABLE` clause into this newly defined operator within the logical plan. Following the formation of the logical plan, the frontend invokes the converged optimizer to generate the optimal physical plan. For the relational-graph interplay optimizations, we incorporate heuristic rules such as FilterIntoMatchRule and TrimAndFuseRule into Calcite's rule-based HepPlanner, by specifying the activation conditions and consequent transformations of each rule. For more nuanced optimization, we rely on the VolcanoPlanner, the cost-based planner in Calcite, to optimize the `ScanGraphTableRelNode`. We devised a top-down search algorithm that assesses the most efficient physical plan based on a cost model outlined in Sec. 4.2.1, combined with high-order statistics from GLogue for more accurate cost estimation. While low-order statistics primarily focus on the cardinalities of relational tables, high-order statistics also include the frequencies of sub-patterns (can be seen as the joined results of multiple tables of vertices and edges), which aids in more accurate cost estimation. It is important to note that RelGo remains functional with only low-order statistics, but the efficiency of the generated plan may decrease due to less accurate cost estimation.

For the remaining relational operators in the query, we leverage Calcite's built-in optimizer, which already includes comprehensive relational optimization techniques. Lastly, the converged optimizer outputs an optimized and platform-independent plan formatted with Google Protocol Buffers (protobuf) [42], ensuring the adaptability of RelGo's output to various backend database systems.

We developed the RelGo framework's backend in C++ using DuckDB as the relational execution engine to showcase its optimization capabilities. We integrated graph index support in GRainDB [23]. With graph index, the `EXPAND`, `EXPAND_EDGE` and `GET_VERTEX` operators can be optimized by directly using the predefined join in GRainDB. Note that we craft a new join on DuckDB called *EI-Join* for the support of `EXPAND_INTERSECT`. Without graph index, the `HASH_JOIN` operator is used throughout the entire plan. To execute the optimized plans within DuckDB, we introduced a runtime module that translates the optimized physical plan into a sequence of DuckDB/GRainDB-compatible executable operators. This runtime module essentially bridges the gap between the optimized plans produced by RelGo and DuckDB's execution engine, thereby validating RelGo's practicality and potential performance improvements for SPJM queries on an established relational database system.

5 Evaluation

5.1 Experimental Settings

Benchmarks. Our experiments leverage two widely used benchmarks to assess system performance, as follows:

LDBC SNB. We use *LDBC*10, *LDBC*30, and *LDBC*100 with scale factors of 10, 30, and 100, generated by the official LDBC Data Generator. These datasets were chosen because they can be accommodated in the main memory of a single configured machine. We select 10 queries from the LDBC Interactive workload for evaluation, denoted as $\mathrm{IC}_{1,\ldots,9,11,12}$, with 10, 13, and 14 excluded since they involve either pre-computation or shortest-path that are not supported. To accommodate queries containing variable-length paths [23], we followed [23] to slightly modify them by separating each query into multiple individual queries with fixed-length paths. Each of these modified queries is denoted with a suffix "-l", where l represents the length of the fixed-length path. In addition, we carefully designed two sets of queries for the comprehensiveness of evaluation, including (1) $QR_{1...4}$ to test the effectiveness of FilterIntoMatchRule and TrimAndFuseRule in RelGo, and (2) $QC_{1...3}$, comprising three typical patterns with cycles including triangle, square, and 4-clique, to assess the efficiency of EXPAND_INTERSECT introduced in Sec. 3.2.

JOB. The Join Order Benchmark (JOB) [31] on Internet Movie Database (IMDB) is adopted. We select the variants marked with "a" of all JOB queries, referred to as $\text{JOB}_{1...33}$, without loss of generality. These queries are primarily designed to test join order optimization, with each query containing an average of 8 joins.

The largest dataset (i.e., $LDBC100$) contains 282 million tuples in vertex relations and 938 million tuples in edge relations. More detailed statistics of the datasets are available in the full version[35]. We manually implement the queries using SQL/PGQ, which are presented in the artifact [34]. Furthermore, we perform the RGMapping process in a manner that allows the construction of the same graph index on the LDBC and JOB datasets used in GRainDB's experiments [23]. Specifically, the EV-index and VE-index on potential edge relations are constructed on foreign keys and tables that depict many-to-many relationships.

Compared Systems. To ensure a fair comparison, all systems except Kùzu use DuckDB v0.9.2 as the relational execution engine, differing only in their optimizers. Since GRainDB was originally implemented on an older version of DuckDB, we have reimplemented it on DuckDB v0.9.2, which offers improved performance over the original version. Kùzu utilizes its own execution engine (v0.4.2) as a baseline of a graph database management system (GDBMS).

DuckDB [2]: This system optimizes queries using the graph-agnostic approach, leveraging DuckDB's built-in optimizer as described in Sec. 4.1. It serves as the naive baseline for extending a relational database system to support SPJM.

GRainDB [23]: This system uses same optimizer as DuckDB but employs the graph index (Sec. 3.2.1) for query execution. It acts as the baseline to demonstrate that solely using graph index is insufficient for optimizing SPJM.

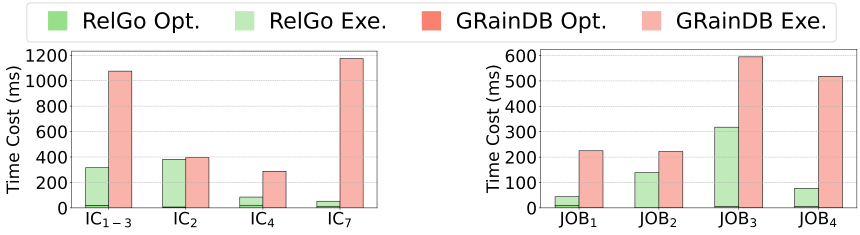
Umbra [15, 37]: This system features an advanced hybrid optimizer capable of generating wco join plans. We obtained the Umbra executable from the authors and configured its parameters according to their recommendations for computing the execution plan. The execution plan is then executed on DuckDB[2], utilizing the graph index when applicable, as done in GRainDB. This helps demonstrate that even with an advanced relational optimizer and the addition of a graph index, it can still fall short in optimizing SPJM.

RelGo: This system optimizes queries using the converged optimizer presented in Sec. 4.2 and utilizes the graph index for query execution. It demonstrates the full range of techniques introduced in this paper. There are some variants of RelGo for verifying the effectiveness of the proposed techniques, which will be introduced in the corresponding experiments.

Kùzu [22]: This system is a GDBMS that adopts the property graph data model. We use it as a baseline to compare the performance gap between RelGo on relational databases and native graph databases.

Configurations. Our experiments were conducted on a server equipped with an Intel Xeon E5-2682 CPU running at 2.50GHz and 256GB of RAM, with parallelism restricted to a single thread. For a comprehensive performance analysis, each query from the LDBC benchmark was run 50 times using the official parameters, while each query from the JOB benchmark was executed 10 times. We report the average time cost for each query to mitigate potential biases. We imposed

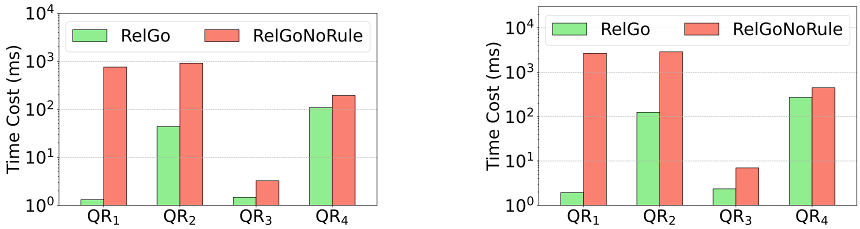
[2] Notably, all Umbra's plans for the benchmark queries exclude the multiway-join operator, allowing for direct transformation into DuckDB's runtime.



(a) E2E Time on *LDBC*30.

(b) E2E Time on IMDB.

Fig. 7. Experiments on optimization and execution cost



(a) Time Cost on *LDBC*10.

(b) Time Cost on *LDBC*30.

Fig. 8. Efficiency comparison of RelGo and RelGoNoRule

a timeout limit of 10 minutes for each query, and queries that fail to finish within the limit are marked as OT.

5.2 Micro Benchmarks on RelGo

In this subsection, we conducted three micro benchmarks to evaluate the effectiveness of RelGo, including assessing the efficiency of the optimizer, testing its advanced optimization strategies, and examining its effectiveness in optimizing join order.

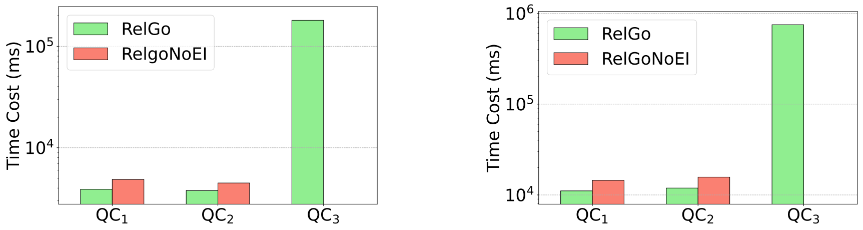
Optimization Efficiency Evaluation. First, we assessed the optimization efficiency by comparing RelGo with GRainDB[23]. We tested their optimization time and also evaluated the execution time for their optimized plans as a measure of the plan quality. We considered end-to-end time as optimization time plus execution time. We randomly selected two subsets of the LDBC and JOB queries, and conducted the experiments on *LDBC*30 and IMDB datasets.

The results in Fig. 7 reveal that RelGo significantly outperforms GRainDB in terms end-to-end time, achieving an average speedup of 7.5× on *LDBC*30 and 3.8× on IMDB. However, note that RelGo incurs a slightly higher optimization cost compared to GRainDB. Although RelGo theoretically has a narrower search space, as analyzed in Sec. 3.1.3, GRainDB benefits from DuckDB's optimizer, which includes very aggressive pruning strategies. Despite the slightly higher optimization cost, RelGo generates superior optimized plans, surpassing GRainDB by an average of 9.7× on LDBC30 and 4.3× on IMDB in execution time.

For fair comparison, in the subsequent experiments, we evaluate the efficiency of different systems using the end-to-end time.

Advanced Optimization Strategies. In this experiment, we assessed the advanced optimization strategies in RelGo, including the heuristic FilterIntoMatchRule and TrimAndFuseRule, and the optimized implementation of EXPAND_INTERSECT operator that aims to improve the efficiency of complete star join.

We began by testing heuristic rules FilterIntoMatchRule and TrimAndFuseRule. We conducted experiments on *LDBC*10 and *LDBC*30, using QR_1 and QR_2 to test FilterIntoMatchRule, and QR_3



(a) Time Cost on *LDBC*10.

(b) Time Cost on *LDBC*30.

Fig. 9. Efficiency comparison of RelGo and RelGoNoEI

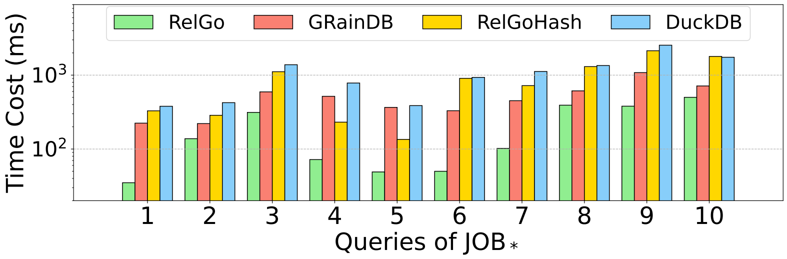
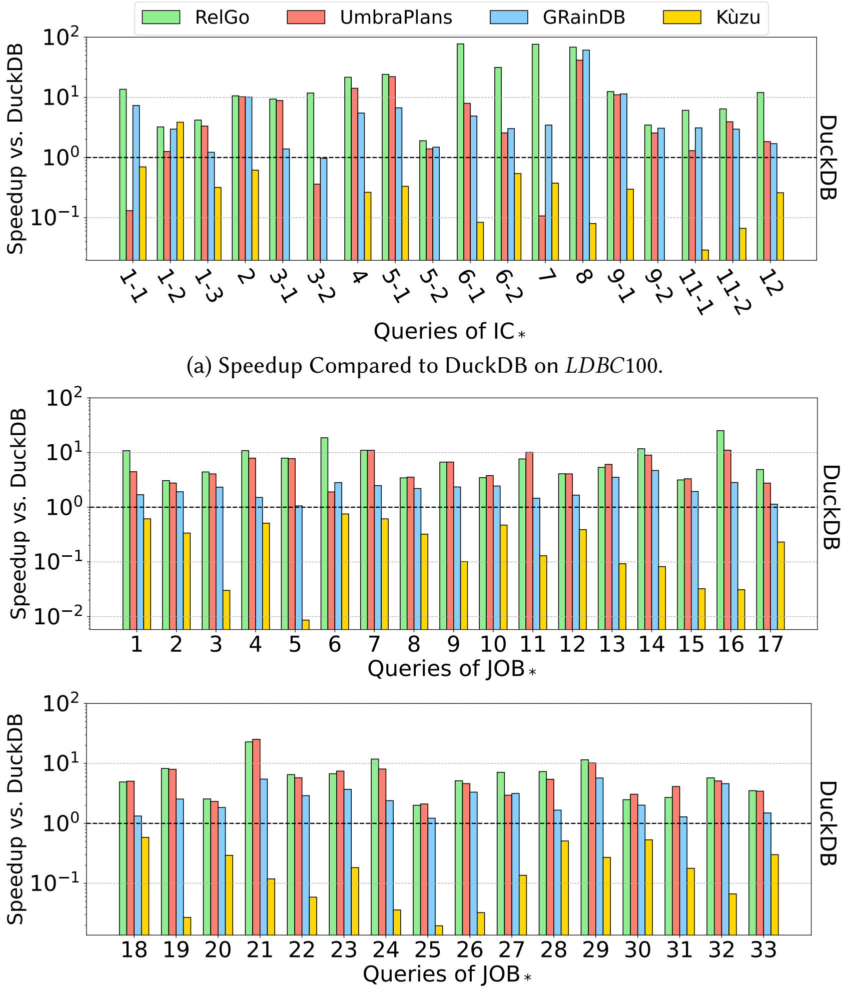


Fig. 10. Experiments on join order efficiency

and QR_4 to test TrimAndFuseRule. The results in Fig. 8 compared the performance of RelGo with and without applying these rules, denoted as RelGo and RelGoNoRule, respectively. The results show that FilterIntoMatchRule significantly improves query performance, providing an average speedup of 299.4× on *LDBC*10 and 699.8× on *LDBC*30. With TrimAndFuseRule, query execution is accelerated by an average of 2.0× on *LDBC*10 and 2.3× on *LDBC*30. These findings suggest that the heuristic rules, particularly FilterIntoMatchRule, are highly effective in enhancing query execution efficiency.

Next, we evaluated the effectiveness of the EXPAND_INTERSECT, which focuses on improving the efficiency of complete star join. Without this optimization strategy, the EXPAND_INTERSECT operator would be implemented as a traditional multiple join, and we denote this variant as RelGoNoEI. Queries $QC_{1\ldots3}$ that contain cycles are used to compare the performance of RelGo and RelGoNoEI. The performance results in Fig. 9 suggest that, compared to RelGoNoEI, RelGo achieves an average speedup of 1.22× on *LDBC*10 and 1.31× on *LDBC*30 (excluding QC_3). Notably, for QC_3, which is a complex 4-clique, the plans optimized by RelGoNoEI confront an out-of-memory (OOM) error. The results indicate that EXPAND_INTERSECT with an optimized implementation not only enhances query performance but also significantly reduces the spatial overhead.

Efficiency of Join Order. We compared RelGo with GRainDB and DuckDB, focusing on the efficiency of the join order. For this purpose, we introduced a variant of RelGo called RelGoHash, which optimizes the plan in a converged manner like RelGo but deliberately bypasses the use of graph index. We selected 10 queries from the JOB benchmark and showed the performance results in Fig. 10. The results demonstrate that RelGo outperforms GRainDB on all the queries, accelerating the execution time by factors ranging from 1.4× to 7.5×, with an average speedup of 4.1×. Additionally, the plans optimized with RelGoHash are at least as good as those optimized by DuckDB, achieving an average speedup of 1.6×. The effectiveness of RelGo and RelGoHash stems from their use of advanced graph-aware optimization techniques in optimizing the matching operator, resulting in good join order and thus robust performance regardless of graph index. It is worth noting that RelGo does not always generate the absolute best join orders, as it relies on the



(a) Speedup Compared to DuckDB on *LDBC*100.

(b) Speedup Compared to DuckDB on IMDB.

Fig. 11. Results of the comprehensive experiments. The speedup is computed as $\frac{\text{Time(DuckDB)}}{\text{Time(Compared Method)}}$.

estimated cost of the plans. However, its optimized plans generally remain competitive in most cases, thanks to its integration of GLogue that use high-order statistics for cost estimation.

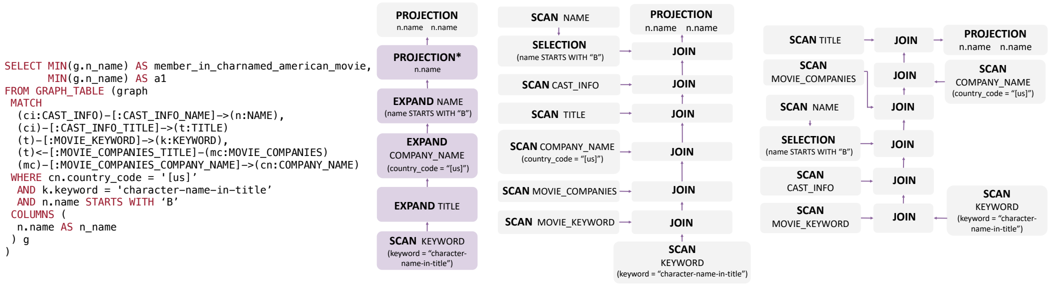
5.3 Comprehensive Experiments

We conducted comprehensive experiments on the LDBC and JOB benchmarks to comprehensively evaluate the performance of RelGo compared to DuckDB, GRainDB, Umbra, and Kùzu. The experimental results are shown in Fig. 11. The results on LDBC10 and LDBC30 are omitted because they are comparable to those on LDBC100. Complete results are provided in the full version[35].

5.3.1 Comparison with DuckDB and GRainDB. Firstly, we compared the performance of RelGo with DuckDB and GRainDB. Specifically, for the LDBC benchmark, the execution time of the plans optimized by RelGo is about 21.9× and 5.4× faster on average than those generated by DuckDB and GRainDB on *LDBC*100. It is important to note that RelGo is especially effective for queries containing cycles, which can benefit more from graph optimizations. For example, in query IC_7, which contains a cycle, RelGo outperforms DuckDB and GRainDB by 76.3× and 22.0×, respectively.

Conversely, the JOB benchmark, established for assessing join optimizations in relational databases, lacks any cyclic-pattern queries. Despite this, RelGo still achieves better performance compared to DuckDB and GRainDB, with an average speedup of 8.2× and 4.0×, respectively.

The experimental results reflect our discussions in Sec. 3.1.2. We summarize RelGo's superiority as follows. First, RelGo is designed to be aware of the existence of graph index in query optimization and can leverage the index to effectively retrieve adjacent edges and vertices. In contrast, for GRainDB, relational optimizers can occasionally alter the order of EVJoin operations, making graph index ineffective. DuckDB, on the other hand, does not consider graph index in query optimization and executes queries using conventional hash joins, which are often less efficient compared to graph-aware approaches. Second, by incorporating a matching operator in SPJM queries to capture the graph query semantics, RelGo is able to leverage advanced graph optimization techniques to optimize matching operators. These techniques include using high-order statistics to estimate the cost of plans more accurately and employing wco join implementations to optimize cyclic patterns. In contrast, DuckDB and GRainDB cannot benefit from these graph-specific optimizations, which may lead to suboptimal plans and inefficient execution. Third, RelGo considers optimization opportunities across both graph and relational query semantics, introducing effective heuristic rules such as FilterIntoMatchRule and TrimAndFuseRule. These rules can significantly improve the efficiency of the generated plans.



```
SELECT MIN(g.n_name) AS member_in_charnamed_american_movie,
       MIN(g.n_name) AS a1
FROM GRAPH_TABLE (graph
 MATCH
  (ci:CAST_INFO)-[:CAST_INFO_NAME]->(n:NAME),
  (ci)-[:CAST_INFO_TITLE]->(t:TITLE)
  (t)-[:MOVIE_KEYWORD]->(k:KEYWORD),
  (t)<-[:MOVIE_COMPANIES_TITLE]-(mc:MOVIE_COMPANIES)
  (mc)-[:MOVIE_COMPANIES_COMPANY_NAME]->(cn:COMPANY_NAME)
 WHERE cn.country_code = '[us]'
  AND k.keyword = 'character-name-in-title'
  AND n.name STARTS WITH 'B'
 COLUMNS (
  n.name AS n_name
 ) g
)
```

PROJECTION n.name n.name
PROJECTION* n.name
EXPAND NAME (name STARTS WITH "B")
EXPAND COMPANY_NAME (country_code = "[us]")
EXPAND TITLE
SCAN KEYWORD (keyword = "character-name-in-title")
SCAN NAME
SELECTION (name STARTS WITH "B")
SCAN CAST_INFO
SCAN TITLE
SCAN COMPANY_NAME (country_code = "[us]")
SCAN MOVIE_COMPANIES
SCAN MOVIE_KEYWORD
PROJECTION n.name n.name
JOIN
JOIN
JOIN
JOIN
JOIN
JOIN
SCAN KEYWORD (keyword = "character-name-in-title")
SCAN TITLE
JOIN
PROJECTION n.name n.name
SCAN MOVIE_COMPANIES
JOIN
SCAN COMPANY_NAME (country_code = "[us]")
SCAN NAME
JOIN
SELECTION (name STARTS WITH "B")
JOIN
SCAN CAST_INFO
JOIN
SCAN MOVIE_KEYWORD
JOIN
SCAN KEYWORD (keyword = "character-name-in-title")

(a) JOB$_{17}$ query (b) Query Plan of RelGo (c) Query Plan of GRainDB (d) Query Plan of Umbra

Fig. 12. JOB$_{17}$'s plans given by RelGo, GRainDB and Umbra. JOINs are implemented as GRainDB's predefined joins if possible.

5.3.2 Comparsion with Umbra. We then compared the performance of RelGo and Umbra. In detail, the plans optimized by RelGo are about 49.9× faster on average than those generated by Umbra on *LDBC*100. On JOB benchmark, the plans generated by RelGo are on average 1.7× more efficient than those given by Umbra. Several factors contribute to the results: (1) Umbra, due to its lack of a graph perspective, might generate query plans that encounter challenges in utilizing graph indexes effectively, similar to GRainDB; (2) Although Umbra's optimizer supports generating worst-case optimal plans that include multiway joins, none of Umbra's optimized plans for the tested queries in our experiments contained multiway joins. In contrast, RelGo excels at identifying opportunities to effectively utilize graph indices and adheres to worst-case optimality.

There are instances where Umbra outperforms RelGo in execution plans. For example, when querying JOB$_{30}$ on IMDB, the execution time of the plan generated by RelGo is approximately 1.2× slower than that of Umbra. A potential reason is that RelGo has not yet considered the distributions of attribute values. For example, when the predicate "t.production_year > 2000" is present, knowing the distribution of the attribute "production_year" can help better estimate the results after filtering by the predicate. Hence, Umbra can sometimes estimate cardinalities more accurately when such predicates exist. Addressing this will be an important future work.

5.3.3 Comparison with Kùzu. Finally, we compared RelGo with the GDBMS, Kùzu. The experimental results show that RelGo is approximately 188.7× faster on average than Kùzu on *LDBC*100 and 136.1× faster on the JOB benchmark. Some results of Kùzu are omitted (e.g., IC_{3-1} on *LDBC*100) due to OOM errors. As Kùzu is also developed based on DuckDB, we speculated that Kùzu may not sufficiently exploit graph-specific optimizations as RelGo does.

5.4 Case Study

To further illustrate why the plans generated by RelGo are superior to those produced by the baseline optimizers, we conducted a case study on JOB_{17} as an example, shown in Fig. 12(a). The optimized query plans by RelGo, GRainDB, and Umbra for this query are presented in Fig. 12(b)-(d). Fig. 11b shows that RelGo's plan runs 4.3× and 1.8× faster than those optimized by GRainDB and Umbra, respectively.

A key difference between the plan of RelGo and those of GRainDB and Umbra is that RelGo can consistently follow the graph query semantics by continuously expanding from a starting vertex to its neighbors, leveraging the graph index. For example, RelGo's plan begins with scanning R_{KEYWORD}, then expands to its neighbors R_{TITLE}, followed by $R_{\mathrm{COMPANY\_NAME}}$, and finally R_{NAME}. In this order, the graph indices (both EV-index and VE-index) introduced in Sec. 3.2.1 are fully utilized to efficiently retrieve neighboring vertices. In contrast, GRainDB and Umbra, as relational optimizers, may not always adhere to this semantics. For instance, in GRainDB's plan, after joining R_{KEYWORD} with $R_{\mathrm{MOVIE\_KEYWORD}}$, the plan misses the opportunity to immediately join R_{TITLE}, thus failing to use the EV-index constructed between $R_{\mathrm{MOVIE\_KEYWORD}}$ and R_{TITLE} right away. A similar situation occurs in Umbra's plan.

6 Related Work

Query Optimization for Relational Databases. Various studies of query optimization for relational databases were proposed to find the optimal join order [18, 21, 25, 26]. For example, Haffner et al. [18] converted join order optimization into finding the shortest path on directed graphs and used the A* algorithm to solve it. Kossmann et al. [25] summarized the methods to optimize queries with data dependencies, such as uniqueness constraints, foreign key constraints, and inclusion dependencies. Recently, researchers attempt to incorporate wco joins into plans to better handle queries with cycles and reduce the size of intermediate results [4, 50]. CLFTJ [24] introduces caching into trie join to reuse previously computed results. Umbra [15] proposes a new hash trie data structure and further reduces the cost of set intersection. All these techniques can be orthogonally adopted in RelGo's relational optimization.

Query Optimization for Graph Databases. Graph pattern matching, a fundamental problem in graph query processing, has been extensively studied [6]. In sequential settings, Ullmann's backtracking algorithm [49] has been optimized using various techniques, such as tree indexing [44], symmetry breaking [19], and compression [7]. Join-based algorithms have been developed for distributed environments. These algorithms use cost estimation to optimize join order, with binary-join algorithms[27, 28] estimating costs using random graph models and worst-case-optimal join algorithms [5] ensuring a worst-case upper bound on the cost. Hybrid approaches[22, 36, 51] adaptively select between binary and wco joins based on the lower cost. Recent studies have focused on improving cost estimation in graph pattern matching, including decomposing graphs into star-shaped subgraphs [38] and comparing different cardinality estimation methods [41]. Some optimizers, like GLogS [29], search for the optimal plan by representing edges as binary joins or vertex-expansion subtasks. We follow the join-based methods such as [29, 51] due to their compatibility with the relational context for which RelGo is designed.

Bridging Relational and Graph Models. There is a growing interest in studying the interaction between relational and graph models. DuckPGQ [47, 48] has demonstrated support for SQL/PGQ within the DuckDB [2], utilizing the straightforward, graph-agnostic approach to transform and process pattern matching. Hence, DuckPGQ loses the opportunity to optimize the query from a graph query perspective. Index-based methods, such as GQ-Fast [33] and GRainDB [23], work towards construct graph-like index on relational databases to improve the performance of join execution. RelGo leveraged GRainDB's indexing technique for implementing physical graph operations. In contrast, methods like GRFusion [20] and Gart [45] work towards materializing graph from the relational tables, so that graph queries can be executed directly on the materialized graph. Such methods incur additional storage costs and potential inconsistencies between relational and graph data.

7 Conclusions and Discussion

In this paper, we introduce RelGo, a converged relational-graph optimization framework designed for SQL/PGQ queries. We formulate the SPJM query skeleton to better analyze and optimize the relational-graph hybrid queries introduced by SQL/PGQ. After discovering that a graph-agnostic approach can result in a larger search space and suboptimal query plans, we design RelGo to optimize the relational and graph components of SPJM queries using dedicated relational and graph optimization modules, respectively. Additionally, RelGo incorporates optimization rules, such as FilterIntoMatchRule, which optimize queries across the relational and graph components, further enhancing overall query efficiency. We conduct extensive experiments comparing RelGo to graph-agnostic baselines, demonstrating its superior performance and confirming the effectiveness of our optimization techniques.

One interesting future direction is to extend RelGo to directly process existing SPJ queries as inputs, enabling the automatic conversion from SPJ to SPJM queries while being aware of the presence of graph indices. Boudaoud et al. [8] may have discussed potential methods for such conversion. However, designing a global solution to determine which parts of an SPJ query can be converted into a matching operator is challenging. This decision involves exhaustively exploring the search space, now including both join and pattern matching options. Given the high cost of optimizing joins alone, an exhaustive search could become prohibitively expensive. Therefore, it is necessary to carefully balance and select appropriate global and local optimization rules for given queries.

References

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

[2] 2024. DuckDB. https://duckdb.org/.

[3] 2024. openCypher. https://opencypher.org/.

[4] Christopher R. Aberger, Susan Tu, Kunle Olukotun, and Christopher Ré. 2016. EmptyHeaded: A Relational Engine for Graph Processing. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 431–446. https://doi.org/10.1145/2882903.2915213

[5] Khaled Ammar, Frank McSherry, Semih Salihoglu, and Manas Joglekar. 2018. Distributed Evaluation of Subgraph Queries Using Worst-Case Optimal Low-Memory Dataflows. *Proc. VLDB Endow.* 11, 6 (oct 2018), 691–704. https://doi.org/10.14778/3184470.3184473

[6] Renzo Angles, Marcelo Arenas, Pablo Barceló, Aidan Hogan, Juan Reutter, and Domagoj Vrgoč. 2017. Foundations of Modern Query Languages for Graph Databases. *ACM Comput. Surv.* 50, 5, Article 68 (sep 2017), 40 pages.

[7] 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.

[8] Abdelkrim Boudaoud, Houari Mahfoud, and Azeddine Chikh. 2022. Towards a Complete Direct Mapping from Relational Databases to Property Graphs. In *Model and Data Engineering: 11th International Conference, MEDI 2022,*

Cairo, Egypt, November 21-24, 2022, Proceedings (Lecture Notes in Computer Science, Vol. 13761), Philippe Fournier-Viger, Ahmed Hassan Yousef, and Ladjel Bellatreche (Eds.). Springer, 222–235. https://doi.org/10.1007/978-3-031-21595-7_16

[9] Donald D. Chamberlin and Raymond F. Boyce. 1974. SEQUEL: A structured English query language. In *Proceedings of the 1974 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control* (Ann Arbor, Michigan) *(SIGFIDET '74)*. Association for Computing Machinery, New York, NY, USA, 249–264.

[10] S. Chatterji, S. S. K. Evani, S. Ganguly, and M. D. Yemmanuru. 2002. On the complexity of approximate query optimization. In *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS '02)*. Association for Computing Machinery, New York, NY, USA, 282–292.

[11] Surajit Chaudhuri. 1998. An overview of query optimization in relational systems. In *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '98)*. Association for Computing Machinery, New York, NY, USA, 34–43.

[12] Surajit Chaudhuri and Kyuseok Shim. 1999. Optimization of queries with user-defined predicates. *ACM Trans. Database Syst.* 24, 2 (jun 1999), 177–228.

[13] Jin Chen, Guanyu Ye, Yan Zhao, Shuncheng Liu, Liwei Deng, Xu Chen, Rui Zhou, and Kai Zheng. 2022. Efficient Join Order Selection Learning with Graph-based Representation. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining* (Washington DC, USA) *(KDD '22)*. Association for Computing Machinery, New York, NY, USA, 97–107.

[14] Peter Pin-Shan Chen. 1983. English sentence structure and entity-relationship diagrams. *Information Sciences* 29, 2-3 (1983), 127–149.

[15] Michael J. Freitag, Maximilian Bandle, Tobias Schmidt, Alfons Kemper, and Thomas Neumann. 2020. Adopting Worst-Case Optimal Joins in Relational Database Systems. *Proc. VLDB Endow.* 13, 11 (2020), 1891–1904. http://www.vldb.org/pvldb/vol13/p1891-freitag.pdf

[16] Jonathan Goldstein and Per-Åke Larson. 2001. Optimizing queries using materialized views: a practical, scalable solution. *SIGMOD Rec.* 30, 2 (may 2001), 331–342.

[17] Goetz Graefe. 1995. The Cascades Framework for Query Optimization. *IEEE Data Eng. Bull.* 18, 3 (1995), 19–29. http://sites.computer.org/debull/95SEP-CD.pdf

[18] Immanuel Haffner and Jens Dittrich. 2023. Efficiently Computing Join Orders with Heuristic Search. *Proc. ACM Manag. Data* 1, 1, Article 73 (may 2023), 26 pages.

[19] Wook-Shin Han, Jinsoo Lee, and Jeong-Hoon Lee. 2013. Turboiso: Towards Ultrafast and Robust Subgraph Isomorphism Search in Large Graph Databases. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data* (New York, New York, USA) *(SIGMOD '13)*. Association for Computing Machinery, New York, NY, USA, 337–348. https://doi.org/10.1145/2463676.2465300

[20] Mohamed S. Hassan, Tatiana Kuznetsova, Hyun Chai Jeong, Walid G. Aref, and Mohammad Sadoghi. 2018. Extending In-Memory Relational Database Engines with Native Graph Support. In *Proceedings of the 21st International Conference on Extending Database Technology, EDBT 2018, Vienna, Austria, March 26-29, 2018*, Michael H. Böhlen, Reinhard Pichler, Norman May, Erhard Rahm, Shan-Hung Wu, and Katja Hose (Eds.). OpenProceedings.org, 25–36. https://doi.org/10.5441/002/EDBT.2018.04

[21] Toshihide Ibaraki and Tiko Kameda. 1984. On the optimal nesting order for computing N-relational joins. *ACM Trans. Database Syst.* 9, 3 (sep 1984), 482–502.

[22] Guodong Jin, Xiyang Feng, Ziyi Chen, Chang Liu, and Semih Salihoglu. 2023. KÙZU Graph Database Management System. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023*. www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p48-jin.pdf

[23] Guodong Jin and Semih Salihoglu. 2022. Making RDBMSs Efficient on Graph Workloads Through Predefined Joins. *Proc. VLDB Endow.* 15, 5 (2022), 1011–1023. https://doi.org/10.14778/3510397.3510400

[24] Oren Kalinsky, Yoav Etsion, and Benny Kimelfeld. 2017. Flexible Caching in Trie Joins. In *Proceedings of the 20th International Conference on Extending Database Technology, EDBT 2017, Venice, Italy, March 21-24, 2017*, Volker Markl, Salvatore Orlando, Bernhard Mitschang, Periklis Andritsos, Kai-Uwe Sattler, and Sebastian Breß (Eds.). OpenProceedings.org, 282–293. https://doi.org/10.5441/002/EDBT.2017.26

[25] Jan Kossmann, Thorsten Papenbrock, and Felix Naumann. 2022. Data dependencies for query optimization: a survey. *VLDB J.* 31, 1 (2022), 1–22. https://doi.org/10.1007/s00778-021-00676-3

[26] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. 1986. Optimization of Nonrecursive Queries. In *VLDB'86 Twelfth International Conference on Very Large Data Bases, August 25-28, 1986, Kyoto, Japan, Proceedings*, Wesley W. Chu, Georges Gardarin, Setsuo Ohsuga, and Yahiko Kambayashi (Eds.). Morgan Kaufmann, 128–137. http://www.vldb.org/conf/1986/P128.PDF

[27] Longbin Lai, Lu Qin, Xuemin Lin, and Lijun Chang. 2015. Scalable subgraph enumeration in mapreduce. *Proceedings of the VLDB Endowment* 8, 10 (2015), 974–985.

[28] Longbin Lai, Zhu Qing, Zhengyi Yang, Xin Jin, Zhengmin Lai, Ran Wang, Kongzhang Hao, Xuemin Lin, Lu Qin, Wenjie Zhang, Ying Zhang, Zhengping Qian, and Jingren Zhou. 2019. Distributed subgraph matching on timely dataflow. *Proc. VLDB Endow.* 12, 10 (jun 2019), 1099–1112. https://doi.org/10.14778/3339490.3339494

[29] Longbin Lai, Yufan Yang, Zhibin Wang, Yuxuan Liu, Haotian Ma, Sijie Shen, Bingqing Lyu, Xiaoli Zhou, Wenyuan Yu, Zhengping Qian, Chen Tian, Sheng Zhong, Yeh-Ching Chung, and Jingren Zhou. 2023. GLogS: Interactive Graph Pattern Matching Query At Large Scale. In *2023 USENIX Annual Technical Conference, USENIX ATC 2023, Boston, MA, USA, July 10-12, 2023*, Julia Lawall and Dan Williams (Eds.). USENIX Association, 53–69. https://www.usenix.org/conference/atc23/presentation/lai

[30] LDBC Social Network Benchmark. 2022. https://ldbcouncil.org/benchmarks/snb/. [Online; accessed 20-October-2022].

[31] Viktor Leis, Andrey Gubichev, Atanas Mirchev, Peter A. Boncz, Alfons Kemper, and Thomas Neumann. 2015. How Good Are Query Optimizers, Really? *Proc. VLDB Endow.* 9, 3 (2015), 204–215. https://doi.org/10.14778/2850583.2850594

[32] Feifei Li, Bin Wu, Ke Yi, and Zhuoyue Zhao. 2016. Wander Join: Online Aggregation via Random Walks. In *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, Fatma Özcan, Georgia Koutrika, and Sam Madden (Eds.). ACM, 615–629. https://doi.org/10.1145/2882903.2915235

[33] Chunbin Lin, Benjamin Mandel, Yannis Papakonstantinou, and Matthias Springer. 2016. Fast In-Memory SQL Analytics on Typed Graphs. *Proc. VLDB Endow.* 10, 3 (2016), 265–276. https://doi.org/10.14778/3021924.3021941

[34] Yunkai Lou, Longbin Lai, Bingqing Lyu, Yufan Yang, XiaoLi Zhou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2024. *Towards a Converged Relational-Graph Optimization Framework (Artifact).* https://anonymous.4open.science/r/relgo-artifact2-C4F0

[35] Yunkai Lou, Longbin Lai, Bingqing Lyu, Yufan Yang, XiaoLi Zhou, Wenyuan Yu, Ying Zhang, and Jingren Zhou. 2024. *Towards a Converged Relational-Graph Optimization Framework (Full Version).* https://anonymous.4open.science/r/relgo-artifact2-C4F0/paper/paper.pdf

[36] Amine Mhedhbi and Semih Salihoglu. 2019. Optimizing subgraph queries by combining binary and worst-case optimal joins. *Proc. VLDB Endow.* 12, 11 (jul 2019), 1692–1704.

[37] Thomas Neumann and Michael J. Freitag. 2020. Umbra: A Disk-Based System with In-Memory Performance. In *10th Conference on Innovative Data Systems Research, CIDR 2020, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings.* www.cidrdb.org. http://cidrdb.org/cidr2020/papers/p29-neumann-cidr20.pdf

[38] Thomas Neumann and Guido Moerkotte. 2011. Characteristic sets: Accurate cardinality estimation for RDF queries with multiple joins. In *Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, April 11-16, 2011, Hannover, Germany*, Serge Abiteboul, Klemens Böhm, Christoph Koch, and Kian-Lee Tan (Eds.). IEEE Computer Society, 984–994. https://doi.org/10.1109/ICDE.2011.5767868

[39] Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. 2018. Worst-case optimal join algorithms. *Journal of the ACM (JACM)* 65, 3 (2018), 1–40.

[40] Oracle. 2023. *Property Graph Queries (SQL/PGQ).* International Organization for Standardization. Retrieved June, 2023 from https://www.iso.org/standard/79473.html

[41] Yeonsu Park, Seongyun Ko, Sourav S. Bhowmick, Kyoungmin Kim, Kijae Hong, and Wook-Shin Han. 2020. G-CARE: A Framework for Performance Benchmarking of Cardinality Estimation Techniques for Subgraph Matching. In *Proceedings of the 2020 International Conference on Management of Data, SIGMOD Conference 2020, online conference [Portland, OR, USA], June 14-19, 2020*, David Maier, Rachel Pottinger, AnHai Doan, Wang-Chiew Tan, Abdussalam Alawini, and Hung Q. Ngo (Eds.). ACM, 1099–1114. https://doi.org/10.1145/3318464.3389702

[42] Protocol Buffers. 2024. https://protobuf.dev/overview/.

[43] Yuan Qiu, Yilei Wang, Ke Yi, Feifei Li, Bin Wu, and Chaoqun Zhan. 2021. Weighted Distinct Sampling: Cardinality Estimation for SPJ Queries. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 1465–1477.

[44] Haichuan Shang, Ying Zhang, Xuemin Lin, and Jeffrey Xu Yu. 2008. Taming Verification Hardness: An Efficient Algorithm for Testing Subgraph Isomorphism. *Proc. VLDB Endow.* 1, 1 (aug 2008), 364–375. https://doi.org/10.14778/1453856.1453899

[45] Sijie Shen, Zihang Yao, Lin Shi, Lei Wang, Longbin Lai, Qian Tao, Li Su, Rong Chen, Wenyuan Yu, Haibo Chen, Binyu Zang, and Jingren Zhou. 2023. Bridging the Gap between Relational OLTP and Graph-based OLAP. In *2023 USENIX Annual Technical Conference (USENIX ATC 23).* USENIX Association, Boston, MA, 181–196. https://www.usenix.org/conference/atc23/presentation/shen

[46] Il-Yeol Song, Mary Evans, and Eun K Park. 1995. A comparative analysis of entity-relationship diagrams. *Journal of Computer and Software Engineering* 3, 4 (1995), 427–459.

[47] Daniel ten Wolde, Tavneet Singh, Gábor Szárnyas, and Peter A. Boncz. 2023. DuckPGQ: Efficient Property Graph Queries in an analytical RDBMS. In *13th Conference on Innovative Data Systems Research, CIDR 2023, Amsterdam, The Netherlands, January 8-11, 2023.* www.cidrdb.org. https://www.cidrdb.org/cidr2023/papers/p66-wolde.pdf

[48] Daniel ten Wolde, Gábor Szárnyas, and Peter A. Boncz. 2023. DuckPGQ: Bringing SQL/PGQ to DuckDB. *Proc. VLDB Endow.* 16, 12 (2023), 4034–4037. https://doi.org/10.14778/3611540.3611614

[49] Julian R Ullmann. 1976. An algorithm for subgraph isomorphism. *Journal of the ACM (JACM)* 23, 1 (1976), 31–42.

[50] Yisu Remy Wang, Max Willsey, and Dan Suciu. 2023. Free Join: Unifying Worst-Case Optimal and Traditional Joins. *Proc. ACM Manag. Data* 1, 2 (2023), 150:1–150:23. https://doi.org/10.1145/3589295

[51] Zhengyi Yang, Longbin Lai, Xuemin Lin, Kongzhang Hao, and Wenjie Zhang. 2021. HUGE: An Efficient and Scalable Subgraph Enumeration System. In *SIGMOD '21: International Conference on Management of Data, Virtual Event, China, June 20-25, 2021*, Guoliang Li, Zhanhuai Li, Stratos Idreos, and Divesh Srivastava (Eds.). ACM, 2049–2062. https://doi.org/10.1145/3448016.3457237

Received April 2024; revised July 2024; accepted August 2024