GETL: An Extract-Transform-Load Framework Across Graph Models in Graph Warehouse

Feng Yao, Xiaokang Yang, Shufeng Gong, Qian Tao, Yanfeng Zhang, Wenyuan Yu, and Ge Yu, *Senior Member, IEEE*

***Abstract*—Various graph models have emerged to meet diverse application needs, each with unique characteristics and specialties. Managing and analyzing graph data inevitably requires interactions across different models to serve upstream business requirements. Therefore, an Extract-Transform-Load (ETL) tool designed to bridge different graph models is desired. In this paper, we propose GETL, a generalized graph ETL framework capable of automatically identifying graph model schemas and performing seamless data conversion among RDF, RDF-star, labeled property graph, and the relational model. This is attributed to GETL's unified graph representation model, constructed as nested <label, entity> pairs, offering powerful capabilities in graph representation and model compatibility. Additionally, we develop a unified programming interface to support complex graph transformation tasks. It is built upon the Gremlin syntax and provides strong expressive capabilities. Finally, our evaluation demonstrates that GETL outperforms state-of-the-art solutions in terms of model conversion efficiency and data manipulation language (DML) intelligibility.**

***Index Terms*—graph ETL framework, unified graph representation, programming interface.**

I. Introduction

GRAPH-structured data has now drawn widespread attention in various fields. Several models exist, including the Resource Description Framework (RDF), Labeled Property Graph (LPG), and the Relational Model, each with unique characteristics suited to different graph task scenarios. For example, RDF is well-suited for semantic linking and reasoning [1], as well as data integration [2]. LPG is ideal for representing complex network structures, excelling in graph traversal and analysis [3]. The relational model and its database systems is optimal for applications that require strict transaction control and well-structured data [4].

However, for recently emerging graph warehouses [5], [6], supporting diverse graph applications requires fusing and converting graph data across different models as a prerequisite for meaningful analysis. Moreover, graph-based retrieve-augmented generation (GraphRAG) [7], [8] also demands the fusion of various graph representations during graph construction to enhance expressiveness.

Example 1: Figure 1 provides a simplified portrayal of the fusion and conversion of data under different graph models residing in various business scenarios, using Alibaba as an example. (1) In scenarios involving massive data and high concurrency, *e.g.,* Taobao [9] generates 4 billion behavioral data records daily [10], relational tables are employed to maintain the user-product relationship network, which includes attributes such as user purchase behaviors and order information, to support high-concurrency processing (marked as Scenario 1 in Figure 1); (2) In logistics and warehousing operations, RDF is utilized to maintain a knowledge graph of product, inventory, and supplier, facilitating data integration from various channels (Scenario 2); (3) Built upon a vast user base, *e.g.,* with 400 million daily active users on Taobao [11], LPG is used to construct social networks within its ecosystem (Scenario 3). Each scenario is typically equipped with dedicated processing engines to perform specialized tasks, *e.g.,* performing semantic queries on RDF or graph analysis on LPG [12]. Enterprises often need to integrate data from multiple business systems to create a more comprehensive view, enabling deeper data analysis and decision-making. For example, integrating user purchase information from the relational tables in Scenario 1 into social LPG of Scenario 3 enhances the representation of user preferences and facilitates community discovery. Additionally, even a single dataset might be modeled in different ways based on specific business requirements [13]. For instance, converting part of the RDF graph from Scenario 2 into an LPG and utilizing the processing engine in Scenario 3 allows for detailed graph analysis of the flow of goods in the supply chain network [14]. In large enterprises, the fusion and conversion of different data models are essential to break down data silos, thereby promoting data sharing and collaboration across various departments. □

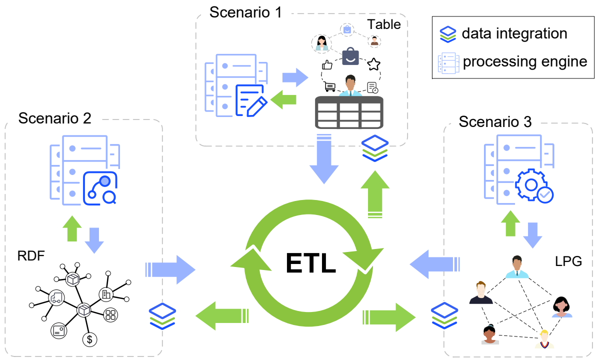


Fig. 1. A case of fusion and conversion involving various types of graph data across different business scenarios.

From the above example, a natural requirement arises: graph data needs to be interconnected and converted between different models across systems. Additionally, the graph often requires a series of transformation operations (*e.g.,* pruning of graph topology) before being loaded into the corresponding processing engines to meet the needs of the analysis task. Therefore, it is essential to develop a graph Extract-Transform-Load (ETL) tool that extracts graph data from source systems, seamlessly transforms it, and loads it into target systems, while concealing the complexity of the underlying models. This is beneficial for practitioners, as they are inclined to focus on the graph area they specialize in without having to manually program cross-model conversion routines.

Currently, there are some efforts on graph model conversion and interoperability. One line of research [4], [15]–[21] focuses on constructing unidirectional or bidirectional mappings between any two of LPG, RDF, and the relational model. However, integrating all pairwise mappings in graph ETL requires incorporating a large number of conversion rules and mapping logic, which would be both complex and rigid. Direct mapping also hinders transformation operations in the ETL process. Another interesting proposal is to employ an intermediary to reconcile the disparities among models.

Unfortunately, the current mainstream graph models are not well-suited to serve as this intermediary. Each model's design possesses unique characteristics while also revealing limitations in other aspects. Specifically, RDF represents data in the form of (*subject, predicate, object*) triples, while supporting global identifiers (IRIs) and a schema-less design [22], making it well-suited for efficient data integration. However, RDF lacks direct counterparts for higher-order relationships [23], *e.g.,* edge properties in LPG. Additionally, it does not support *multigraphs*, which allow multiple edges of the same type between two vertices [12]. Conversely, LPG is characterized as directed multigraphs, adopting the graph terminology to explicitly define labels and properties for vertices and edges. This makes LPG ideal for representing complex network structures and performing graph traversal. However, the flexibility of definitions on schema [21], [24] makes data integration challenging. The relational model can be used to represent graph data, where one table corresponds to a type of vertices in LPG, and edges are possible integrity constraints [25]. The structured organizational form makes various relationships transparent and normalized. While the relational model is valuable for well-structured data, its predefined schema limits the ability to accommodate dynamic changes in graph structures. In a nutshell, as an intermediary, the ability to accommodate various model representations and efficient data integration are goals pursued by the graph ETL. Clearly, none of them possess a fully compatible feature set.

On the other hand, Multilayer Graph [26] and Statement Graph [27] serve as intermediaries, exploring the representation of RDF (including RDF-star [28]) and LPG in *unified data model* manners. Multilayer Graph captures higher-order relationships by adding edge IDs to directed labeled edges for multilayer nesting. Statement Graph associates (*subject, predicate, object*) triples by setting internal nodes with statement identifiers and describes complex relationships through the connections between internal nodes. Both are designed with richer statements to support data representations from two graph models, but this in turn may require increased effort on data integration. Moreover, they cannot handle some of the more complex representational features specific to graph ETL, *e.g.,* paths as vertices, which enables users to manipulate graph data from different hierarchical views. The analysis above prompts us to design a unified data model to serve as an intermediary for graph ETL, which needs to accommodate representations of three mainstream graph models and the specific complex representational features required by graph ETL, while also providing robust data integration capabilities.

Another key requirement for building a graph ETL tool is the design of a *domain-specific* programming interface to perform a wide range of complex transformations during the ETL process. This differs significantly from existing graph query languages, such as Gremlin [29] and Cypher [30], which are proficient in basic graph pattern matching and can serve as core components in graph ETL pipelines, but have limitations in addressing ETL-specific needs. Firstly, these languages struggle to express *special mapping operations* in graph ETL, such as mapping subgraphs to vertices and condensing edges between subgraphs into an edge. Secondly, they appear less elegant in handling *user-defined recursive traversal* operations that involve complex computational logic. For example, constructing weighted shortest paths in pathfinding operations requires complex programming or rigid predefined functions (*e.g., Graph Data Science* library [31] in Neo4j and *VertexPrograms* [32] in TinkerPop).

In this paper, we develop GETL, a generalized graph ETL framework. Firstly, we propose a unified graph representation model. The model uses label-entity pairs to define all elements (*e.g.,* vertex, edge, property, and path) and employs nested binary relations to represent connections between entities at different levels, enhancing the graph representation capabilities. Furthermore, the overall consistent structural form emphasizes its data integration capabilities. Additionally, we define concise mapping rules that automatically identify the schemas of different models and enable seamless data conversion. Secondly, GETL features a three-layer architecture to accommodate different types of graph transformation operations, providing users with flexible and extensive operational space to customize the graph according to the requirements of upstream applications. Finally, we design a programming interface for GETL to facilitate user interactions with graph data within the three-layer architecture throughout the graph ETL process. It is built on Gremlin syntax to support complex mapping operations. Furthermore, by integrating Datalog's recursive querying capabilities with Gremlin's traversal query strengths, we introduce a hybrid Data Manipulation Language to optimize the needs of recursive aggregation operations.

Contributions. Overall, we make the following contributions:

- We develop GETL, a generalized graph ETL framework. To the best of our knowledge, it is the first graph ETL tool specifically designed to accommodate various graph models, serving graph warehouses and supporting broad types of upstream applications (§III).
- We propose a unified graph representation model as an intermediary for graph ETL, featuring robust representational power and strong data integration capabilities. (§IV-A).
- We design concise and intuitive mapping rules to enable the automatic identification of diverse graph model schemas and seamless data conversion. (§IV-B).
- We design unified programming interfaces tailored for the graph ETL process to meet users' needs for extensive graph transformation tasks (§V).

II. Problem Formulation and Model Analysis

In this section, we present the definition of the problem and analyze the characteristics of mainstream models for graphs.

A. Definition of Graph ETL Process

The ETL process involves extracting and transforming data from sources and then loading it into a target system. We define the *graph ETL process* as follows:

Extract. A set of graph sources is $S_G = \cup G_i$, where each G_i is a graph from a different system with a data model m_i (*e.g.,* LPG, RDF, and the relational model). Given a unified data model m_* and a graph form d, the extract function $\epsilon : G_i, m_i \xrightarrow{m_*} d$ converts the graph G_i into a consolidated data form d relying on m_*.

Transform. By integrating over $\cup \epsilon_{m_*}(G_i, m_i)$, a unified graph representation, D_G, is constructed. Graph transformations based on D_G are then executed according to the requirements of upstream applications, categorized into the following types.

(1) *Graph pattern matching*, which essentially matches a set of *path bindings* in a graph [33]. A path $\rho = (v_0, e_0, v_1, e_1, \ldots, e_{n-1}, v_n)$ in D_G represents a sequence of vertex v and edge e identifiers. The sequence is captured through pattern expressions, where each variable in the expression maps to a graph element (vertex, edge).

(2) *Special mapping*, which involves the abstraction of the graph at different layers. Specifically, the transformation aggregates different types of elements (vertices, edges, and *multidimensional* properties) to create an aggregate network [34]. For example, it can address questions such as, "*What is the network structure of a social graph when grouped by user gender?*" It makes sense for the user to customize graphs according to business needs. Additionally, consider a mapping function $f : G' \to v_{G'}$, which maps subgraph G', constructed in different dimensional spaces on D_G, to a new vertex $v_{G'}$ in a new graph layer. Correspondingly, the edge set between subgraphs (*e.g.,* $E(G'_1, G'_2)$) is condensed into a new edge, *i.e.,* $f : E(G'_1, G'_2) \to e(v_{G'_1}, v_{G'_2})$. The f is sufficiently flexible to accommodate various mapping possibilities, such as mapping from subgraph to property and from path to edge.

(3) *User-defined recursive graph traversal*, which requires tailored recursive/iterative configurations and aggregation operations based on the recursive/iterative nature of graph constructs. Standard algorithms often fail to fully satisfy the needs of practical applications, necessitating a degree of customization by the user [35]. Examples include path planning, heuristic search algorithms, personalized graph ranking, weighted random walk, and GNN stochastic sampling.

It is worth noting that the three types of transformation operations mentioned above can function independently or be utilized in conjunction to achieve the desired outcomes.

Load. Generate graph data in a specified data model and load it into upstream graph management or analysis systems.

B. Different Data Models for Graphs

In search of characteristics of data models to lay the foundation for GETL compatibility, we introduce LPG, RDF, RDF-star, and the relational model.

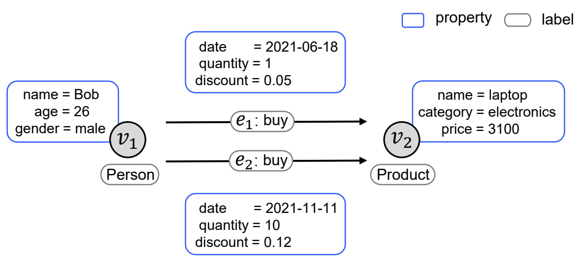


Fig. 2. An example of an LPG for online shopping.

Labeled Property Graph (LPG). The formal definition of LPG varies slightly across different literature, as do the functionalities implemented in various systems guided by it (*e.g.,* some support multi-valued properties or multiple labels, while others do not). In this paper, we follow the definition from [12] and have included adaptations for data type descriptions. Assume that four infinite sets: $\mathcal{L}$ (labels), $\mathcal{K}$ (property names), $\mathcal{V}$ (property values), and $\mathcal{T}$ (data types).

Definition 1 (LPG)**.** *An LPG is a tuple $(V, E, \pi, \lambda, \sigma)$ where:*

(1) V is a finite set of vertices;

(2) E is a finite set of edges such that $V \cap E = \emptyset$;

(3) $\pi : E \to (V \times V)$ is a total function that associates each edge in E with a pair of vertices in V;

(4) $\lambda : (V \cup E) \to \mathcal{L}$ is total function that associates labels from $\mathcal{L}$ with vertices/edges;

(5) $\sigma : (V \cup E) \times \mathcal{K} \to (\mathcal{T}, \mathcal{V})$ is a partial function that associates property pairs (name, value) with vertices/edges.

Example 2: Figure 2 shows an example of an LPG for online shopping, featuring two types of vertices: v_1, labeled as "Person", and v_2, labeled as "Product". v_1 refers to a user named "Bob", and v_2 is a "laptop" with a price of "3100", both possessing their own properties. Edges e_1 and e_2 denote that "Bob" buys the "laptop" twice, with each edge's properties capturing the respective order details. An example of mapping partial elements of LPG is as follows:

$$\begin{array}{l}
V = v_1, v_2, \quad \lambda(v_1) = \text{Person}, \quad \lambda(v_2) = \text{Product}, \\
E = e_1, e_2, \quad \lambda(e_1) = \text{buy}, \quad \pi(e_1) = (v_1, v_2), \\
\sigma(v_1, \text{name}) = (\text{String}, \text{Bob}), \quad \sigma(v_2, \text{category}) = (\text{String}, \text{electronics}), \\
\sigma(e_1, \text{date}) = (\text{DATATIME}, 2021-06-18).
\end{array}$$

Notably, the definition of LPG schemas is relatively flexible and typically vendor-specific [12], which can lead to inconsistencies in integrating LPG data from different sources. □

RDF & RDF-star. The RDF model consists of *subject-predicate-object* triples. Assume that three pairwise disjoint sets: $\mathcal{I}$ (IRIs), $\mathcal{N}$ (literals), and $\mathcal{B}$ (blank nodes).

Definition 2 (RDF)**.** *An RDF triple is a tuple $t = (s, k, o)$ where subject $s \in (\mathcal{I} \cup \mathcal{B})$, predicate $k \in \mathcal{I}$, and object $o \in (\mathcal{I} \cup \mathcal{B} \cup \mathcal{N})$. An RDF graph is a finite subset of RDF triples.*

Example 3: The following provides an RDF example related to Example 2, serialized in the RDF Turtle syntax [36], with namespaces declared using the @prefix directive.

```
@prefix ex: <http://example.org/> .
@prefix rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix xsd: <http://www.w3.org/2001/XMLSchema#> .

    _:v1  rdf:type     ex:Person ;
    _:v1  ex:name      "Bob" ;
    _:v1  ex:age       26^^xsd:integer .

    _:v2  rdf:type     ex:Product ;
    _:v2  ex:name      "laptop" ;
    _:v2  ex:category  "electronics" .

    _:v1  ex:buys      _:v2.
```

Here, the blank nodes _:v$_1$ and _:v$_2$ represent the vertices v_1 and v_2. The triple <_:v$_1$ ex:buys _:v$_2$ > corresponds to an edge in the LPG, and XML Schema datatypes (*e.g.,* xsd:integer) are used to define the data types. □

However, RDF does not provide built-in support for relationships involving more than two entities, *e.g.,* edge properties. RDF-star extends RDF with the *quoted triple*.

Definition 3 (RDF-star)**.** *An RDF-star triple is a tuple $t^* = (s^*, k, o^*)$ where subject $s^* \in \mathcal{I} \cup \mathcal{B} \cup \{t_1^*\}$, predicate $k \in \mathcal{I}$, and object $o^* \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{N} \cup \{t_2^*\}$, with t_1^* and t_2^* being given RDF-star triples not equal to t^*. An RDF-star graph is a finite subset of RDF-star triples.*

Here t_1^* and t_2^* are quoted triples, which are unique. By definition, an RDF graph is also an RDF-star graph.

Example 4: Building on Example 3, RDF-star represents edge properties with quoted triple as follows, serialized in Turtle-star format [37]:

```
≪ _:v1  ex:buys  _:v2 ≫   ex:date     "2021 − 06 − 18"^^xsd:date;
≪ _:v1  ex:buys  _:v2 ≫   ex:quantity   1^^xsd:integer;
≪ _:v1  ex:buys  _:v2 ≫   ex:discount   0.05^^xsd:decimal.
```

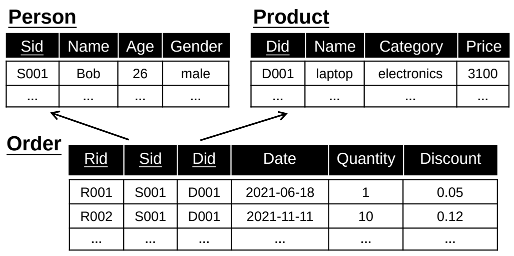
Here, ≪ _:v$_1$ ex:buys _:v$_2$ ≫ is a quoted triple acting as the subject, with order information serving as the predicate and object to form RDF-star triples. □

Regarding alignment with Example 2, RDF-star cannot effectively distinguish between edges e_1 and e_2 and their distinct properties. This is because they form the same triple, ≪ _:v$_1$ ex:buys _:v$_2$ ≫, but the *occurrences* they represent are different. For instance, when the properties of e_1 and e_2 are added to the triple, it is not clear which date corresponds to which discount.

Relational Model. The relational model organizes data into a set of tables, each representing a relation, which in turn is viewed as a collection of tuples.

Definition 4 (Relational Model)**.** *Let $R^* = (R_1, R_2...R_n)$ is a relational schema consisting of relations $R_i(A_1, A_2, ..., A_m)$, where $A_1, A_2..., A_m$ are attributes. Each instance of relation R_i is an ordered n-tuple $(a_1, a_2..., a_m)$, where each a_j is an element of attribute A_j.*

Example 5: Figure 3 illustrates the relational model representation corresponding to Example 2. There are three relational tables: Person, Product, and Order, each column representing an attribute, *e.g.,* "Name" in the Person table, with "Bob" being an element. Additionally, "Sid", "Did", and "Rid" are



Person

Sid	Name	Age	Gender
S001	Bob	26	male
...	...	...	...

Product

Did	Name	Category	Price
D001	laptop	electronics	3100
...	...	...	...

Order

Rid	Sid	Did	Date	Quantity	Discount
R001	S001	D001	2021-06-18	1	0.05
R002	S001	D001	2021-11-11	10	0.12
...	...	...	...	...	...

Fig. 3. The relational model corresponding to Example 2.

primary keys as unique identifiers. Instances of the Person and Product tables correspond to vertices v_1 and v_2 in Example 2, respectively. Instances of the Order table correspond to edges labeled as "buy", which are associated with the Person and Product tables through the foreign keys "Sid" and "Did". □

C. Model Design Philosophy and Insight

We discuss the unique characteristics of models from a design philosophy perspective, exploring insights for the underlying unified graph model design and revealing the pursuits of flexible transformation operations in the graph ETL process.

The RDF focuses on resource identification and format standardization to ensure generality, simplicity, and scalability. Each RDF triple is an independent unit of information, which means that integrating new data does not require complex schema matching or changes to the existing structure; it simply requires appending. However, RDF struggles to represent higher-order relationships, as described in Example 4. Conversely, LPG emphasizes the richness of topology. The model intuitively represents connections and interactions, enhancing efficiency for traversal and graph analysis. However, its flexibility in schema definition poses challenges for data exchange and integration, areas where RDF excels. Additionally, the relational model employs unique identifiers (keys) to ensure entity integrity. However, predefined schema restricts the ability to adapt to dynamic changes within graph structures.

Insight. In the graph ETL process, the underlying unified data model m_* (refer to Section II-A) must be capable of integrating data from different models, aligning with RDF's design philosophy for scalability. It employs a standardized format that does not interfere with existing data and structures. However, this alone is not sufficient; m_* must also cover the representation capabilities of all models, such as representing higher-order relationships in LPG. From this perspective, the design of unique identifiers (keys) in the relational model ensures data integrity, which is crucial for distinguishing and referencing relationships across different orders. On the other hand, LPG's efficient graph traversal capabilities effectively facilitate graph transformation operations during the Transform stage. Additionally, its flexible schema definition offers users a richer space for graph semantics and topological manipulation on integrated graphs.

III. System Overview

Architecture. We build GETL, a generalized graph ETL framework. Figure 4 shows its system architecture, which consists of the following three layers:

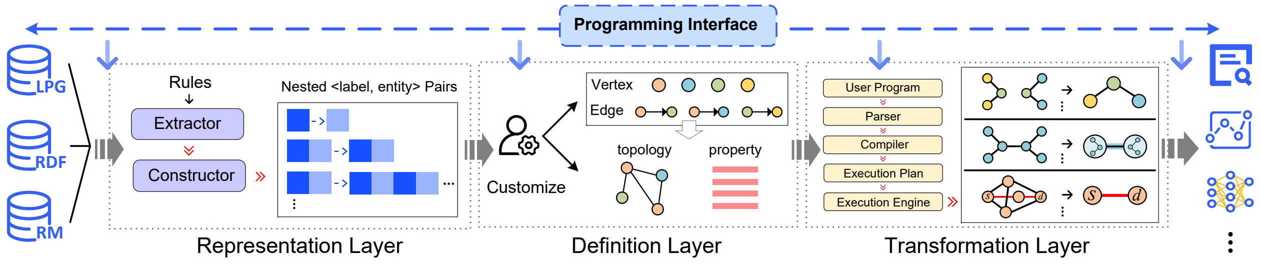


Fig. 4. The architecture overview of GETL.

Representation Layer. The representation layer provides a unified description of graphs from different sources across LPG, RDF, RDF-star, and the relational model (RM). In this layer, we design a unified graph representation model, constructed as nested <label, entity> pair, that acts as an intermediary for converting different data models, accommodating various graph representations. It employs a consistent representation form for different entities *without imposing a fixed schema*, facilitating data integration.

Definition Layer. The underlying model in the representation layer employs a schema-less design to enhance compatibility and integration. However, this flexibility conflicts with the structured and clearly defined syntax required in programming interfaces used for manipulating data. To bridge the gap between the two, we constructed the definition layer. This layer enables users to customize graph schemas using high-level language in the programming interface. Users can flexibly define different types of vertices, edges, and properties based on the underlying model, constructing a well-defined graph. It is worth noting that: 1) The graph defined based on the underlying model is not unique; it is customized by users, who can adapt the schema according to business needs. 2) The choice of property graph schema as the defining layer is due to its flexibility and superior graph traversal capabilities, which facilitate the execution of transformation operations. The design of this layer fulfills the programming interface's requirements for clear structuring, while simultaneously ensuring meaningful data semantics for the integrated graph.

Transformation Layer. Based on the user-customized graph in the definition layer, the transformation layer provides support for three categories of transformations. It features a built-in processing engine to execute transformation operations.

Workflow. We describe the workflow of GETL from the perspective of the graph ETL process.

Extract. Graphs from various sources based on LPG, RDF, and RM are automatically extracted into the representation layer. GETL establishes a set of rules, guiding the extractor in recognizing and extracting source graphs. Subsequently, the constructor unifies these into nested <label, entity> pairs.

Transform. After the extraction is complete, users can manipulate graph transformations through the programming interface. First, customize the schema using the *Data Definition Language* (DDL) statements through the programming interface. For transformations, one approach involves executing simple transformations directly during the graph schema definition phase, such as applying constraints and conducting filtering operations. Alternatively, the defined graph schema can enter the transformation layer as a base graph layer. The difference between the two approaches is that the former is more efficient as it does not require the construction of multi-layered indexes, but it may struggle to support some complex operations. In contrast, the latter approach enables richer and more complex transformations. GETL empowers users by offering them the autonomy to make informed decisions tailored to their specific needs. At the transformation layer, users write transformation scripts through the programming interface. These scripts direct the transformations on the user-customized graph. Upon receiving these scripts, the transformation processing engine handles a series of processes, such as parsing and compiling.

Load. Finally, GETL converts the transformed graph into the graph form under the user-specified data model and loads it into the upstream system. Additionally, the output can serve as an intermediate graph and reintegrate into GETL.

IV. Intermediary Design in GETL

This section presents the design of the unified graph representation model (UGR) within GETL's representation layer.

A. Unified Graph Representation Model (UGR)

Design Idea. We adopt a schema-less design that treats all components of the graph equally as entities and describes the associative relationships expressed by the entities in this perspective. Thus, each entity is an independent unit and forms a consistent expression, which helps avoid complex matching or modifications when integrating new data, thereby facilitating integration. Additionally, we use nested structures to represent higher-order relationships and extend set representations to support specific features.

We first introduce a *label-entity pair* structure, which serves as the basic building block of the model. Let $\mathcal{L}$ be a set of labels and $\mathcal{E}$ be a set of entities representing concepts, things, or objects in the physical world within a given domain $\mathcal{D}$.

Definition 5 (Label-Entity Pair)**.** *A label-entity pair is an ordered pair (lb, et) where $lb \in \mathcal{L}$ and $et \in \mathcal{E}$. The pair (lb, et) forms an association in which lb categorizes, describes, or tags the entity et within the context of $\mathcal{D}$.*

A label lb is a descriptor that can either be any element from $\mathcal{L}$ or be *null*. The flexibility within $\mathcal{L}$ allows for dynamic categorization of entities, accommodating various levels of granularity. An entity et can be identified in various formats, such as numbers, strings, tensors, and IRIs. $\mathcal{D}$ refers to the definition of labels and entity relationships within a particular domain. It encompasses the rules, context, and knowledge system that influence how labels classify or describe entities. Furthermore, label-entity pairs support the description of a class of entities, specifically represented as $(lb, *)$, which denotes the set of all entities under the label lb.

To represent higher-order relationships among components in the graph, we introduce a nested binary relation based on label-entity pairs, termed *Nested <label, entity> Pair*.

Definition 6 (Nested <label, entity> Pair)**.** *Let $\mathcal{P}$ be a set in which each element $p \in \mathcal{P}$ is an ordered pair (p_1, p_2). A nested binary relation $\mathcal{R}$ on $\mathcal{P}$ is defined recursively as follows:*

(1) A basic pair p in $\mathcal{R}$ is a label-entity pair (lb, et);

(2) A nested relation in $\mathcal{R}$ can be an ordered pair $p = (p_1, p_2)$, where p_1 and p_2 are either basic pairs or themselves nested relations.

Intuitively, everything (*e.g.,* vertices, edges, properties, and even paths or an entire graph) can be identified as entities with descriptive labels. The content of entities can be detailed using nested binary relations. The nesting structure still follows the semantics of the label-entity pair form, even at varying levels of nesting. This approach effectively unifies representations of different graph components in a concise way.

Example 6: We continue referencing the example in Figure 2, illustrating its representation through nested label-entity pairs. A partial representation of the elements is provided below:

The basic pair p in $\mathcal{R}$:
$\mathsf{Person} \to \mathsf{ps\_id1}$, $\mathsf{Product} \to \mathsf{pd\_id1}$, $\mathsf{buy} \to \mathsf{by\_id1}$, $\mathsf{buy} \to \mathsf{by\_id2}$,
$\mathsf{String} \to \mathsf{Bob}$, $\mathsf{Integer} \to 26$, $\mathsf{DATETIME} \to 2021-06-18$.
The nested relation pair p in $\mathcal{R}$:
#1 description of user and product information (vertex related)
$(\mathsf{name} : \mathsf{nm\_id1}) \to \big((\mathsf{Person} : \mathsf{ps\_id1}) \to (\mathsf{String} : \mathsf{Bob})\big)$,
$(\mathsf{age} : \mathsf{ag\_id1}) \to \big((\mathsf{Person} : \mathsf{ps\_id1}) \to (\mathsf{Integer} : 26)\big)$,
$(\mathsf{category} : \mathsf{cg\_id1}) \to \big((\mathsf{Product} : \mathsf{pd\_id1}) \to (\mathsf{String} : \mathsf{electronics})\big)$,
#2 description of purchase relationships and orders (edge related)
$(\mathsf{buy} : \mathsf{by\_id1}) \to \big((\mathsf{Person} : \mathsf{ps\_id1}) \to (\mathsf{Product} : \mathsf{pd\_id1})\big)$,
$(\mathsf{date} : \mathsf{dt\_id1}) \to \big((\mathsf{buy} : \mathsf{by\_id1}) \to (\mathsf{DATATIME} : 2021-06-18)\big)$,
$(\mathsf{buy} : \mathsf{by\_id2}) \to \big((\mathsf{Person} : \mathsf{ps\_id1}) \to (\mathsf{Product} : \mathsf{pd\_id1})\big)$,

Firstly, the model defines all components as entities, distinguishing them using unique identifiers, such as "Person → ps_id1". In terms of expression, there is no difference among these components; all can be described using label-entity pairs. For constants, the data type and constant value also form the label-entity pair. Secondly, binary relations between entities are likewise regarded as entities and are identified using a labeled identifier, forming a nested relation pair, such as vertex properties, edges, and properties on edges. Formally, there is no structural difference between vertex-related entities and edge-related entities, despite their differing semantics and relational compositions. The design philosophy behind the unified representation form for different entities is intended to facilitate effective data exchange and integration. □

We further provide set operations on nested <label, entity> pairs to support the representational features of different levels of abstraction specific to graph ETL.

Definition 7 (Set Operations)**.** *Given a nested binary relation $\mathcal{R}$ and an ordered pair $p = (p_1, p_2)$ in $\mathcal{P}$, p_1 and p_2 can be expressed using set operations based on $\mathcal{R}$, including union ($\cup$), intersection ($\cap$), and difference ($\setminus$).*

Set operations that combine or modify relationships in the nested structure can support specific representational features of graph ETL. An example (*path as vertex*) is provided below:

The set operation on pair p in $\mathcal{R}$:
$(\mathsf{specific\_promotion} : \mathsf{sp\_id1}) \to \big((\mathsf{start\_date} : \mathsf{sd\_id1}) \to (\mathsf{DATATIME}: 2021-06-01)\big) \cup \big((\mathsf{duration} : \mathsf{dr\_id1}) \to (\mathsf{Integer} : 18)\big) \cup \big((\mathsf{end\_date} : \mathsf{ed\_id1}) \to (\mathsf{DATATIME} : 2021-06-18)\big)$.

We extend the example of the online shopping graph to describe promotion information with a two-hop path, which represents an 18-day promotion from the "start_date : sd_id1" to the "end_data : ed_id1". This path entity can be treated as a vertex for establishing further associations, such as the entity is linked as a vertex to a promotional product category ("category : cg_id1") or a discount ("discount : dc_id1"). Additionally, we can use $(lb, *)$ to represent a class of entities and use nested relations on it to represent graph entities, *e.g.,* $(\mathsf{graph} : \mathsf{pg\_id1}) \to \big((\mathsf{Person} : *) \to (\mathsf{Product} : *)\big)$, represents a user-product graph entity. Clearly, higher-order associations can be established on the abstraction layer of graph entities.

B. Automated Graph Model Conversion

GETL automatically identifies various graph model schemas and converts them using the concise and natural rules prescribed by UGR. Here, we provide a detailed discussion of the schemas and mapping rules for different models.

Labeled Property Graph. According to Definition 1, for the *LPG* tuple $(V, E, \pi, \lambda, \sigma)$, there exists the following mapping Φ_{LPG} between UGR and LPG:

(R1) $\lambda(V \cup E) \to$ pair (lb, et), where $lb \in \mathcal{L}$ and $et \in (V \cup E)$.

(R2) $\pi(E) \to$ pair $p = (p_1, p_2)$, where p_1 is a label-entity pair (lb, et), with $p_1 \in (\mathcal{L}, E)$, and p_2 is a nested label-entity pair $(p_1^{'}, p_2^{'})$, with $p_1^{'}, p_2^{'} \in (\mathcal{L}, V)$.

(R3) $\sigma\big((V \cup E), \mathcal{K}\big) \to$ pair $p = (p_1, p_2)$, where p_1 is a label-entity pair (lb, et), with $p_1 \in (\mathcal{K}, IDs)$, and p_2 is a nested label-entity pair $(p_1^{'}, p_2^{'})$, with $p_1^{'} \in (\mathcal{L}, (V \cup E))$ and $p_2^{'} \in (\mathcal{T}, \mathcal{V})$.

As demonstrated in Example 6, all entities are identified by labeled unique identifiers, forming label-entity pairs. It is worth noting that these unique identifiers can distinguish *multiple edges* of the same type between vertices. The mapping rule R1 specifies the labeled vertex and edge entities themselves. In R2, p_1 specifies the edge entity, while p_2 represents the binary relation between the two vertex entities that constitute the edge. R3 is a mapping of properties, where p_1 designates the property entity, and p_2 establishes the binary relation between the vertex or edge entity and its corresponding property value.

RDF & RDF-star. Based on Definitions 2 and 3, the mapping Φ_{RDF} between UGR and RDF & RDF-star is as follows:

(R1) Given a tuple $t = (s, k, o)$ with $k = \mathsf{rdf} : \mathsf{type}$, then $t \to$ pair $p^s = (o, s)$, where p^s is a label-entity pair on entity s, and $s, o \in (\mathcal{I} \cup \mathcal{B})$.

(R2) Given a tuple $t = (s, k, o)$ with $k \in \{\mathcal{I} \setminus \{\mathsf{rdf : type}\}\}$, then $t \rightarrow$ pair $p = (p_1, p_2)$, where p_1 is a label-entity pair (lb, et), with $p_1 = (k, id)$, and p_2 is a nested label-entity pair $(p_1^{'}, p_2^{'})$, with $p_1^{'} \in p^s \cup (null, s)$ and $p_2^{'} \in p^o \cup (null, o)$.

(R3) Given a tuple $t^* = (s^*, k, o^*)$, where $s^* \in \mathcal{I} \cup \mathcal{B} \cup \{t_1^*\}$, $k \in \mathcal{I}$, and $o^* \in \mathcal{I} \cup \mathcal{B} \cup \mathcal{N} \cup \{t_2^*\}$, with t as specified in cases (R1) and (R2). There exists $\Phi'_{RDF} : t_1^*, t_2^* \rightarrow p'$, where $p' = p^s$ in case (R1) or $p' = p_1$ in case (R2). Then, $t^* \rightarrow$ pair p^*, where p^* is a nested binary relation pair that embeds p' into the corresponding position within the mapping results of case (R1) or case (R2).

The RDF syntax does not explicitly specify labels, but objects linked by the predicate "rdf : type" can semantically be interpreted as labels. Correspondingly, for R1, triples with the predicate "rdf : type", whose subject s serves as the entity identifier, and the object o serves as the label, form label-entity pairs. Here, p^s is a label-entity pair denoting a labeled entity with the entity identifier positioned at the subject s. When the predicate k is not "rdf : type", the triple in R2 is interpreted as a binary relation between the subject s and the object o concerning the predicate k. Therefore, p_1 specifies the entity k and assigns a unique identifier $id \in IDs$, while p_2 represents the binary relation between the entities s and o. In R3, RDF-star allows triples to be contained within the subject s and object o. Consequently, according to RDF mapping rules R1 and R2, the quoted triple t is mapped first, followed by the mapping of the outer triple. Specifically, the quoted triple t is initially mapped to p^s in R1 or to p_1 in R2. Subsequently, t is replaced at the corresponding positions in the mapping functions of R1 or R2. For example, in an RDF-star triple $t^* = ((s_1, k_1, o_1), k, o)$, the triple (s_1, k_1, o_1) is identified by R1 as $p^{s_1} = (o_1, s_1)$, and then t^* is mapped to $p^* = (k, id) \rightarrow \big((o_1, s_1) \rightarrow p^o\big)$ after R2 quoting p^{s_1} as subject. Note that R3 describes only first-order RDF-star nesting; multi-layer nesting can be handled by a layer-by-layer mapping decomposition.

Relational Model. Following Definition 4, the mapping Φ_{RM} between UGR and RM can be expressed as follows:

(R1) Given a relational table $R_i(A_1, A_2, ..., A_m)$, where attributes A_j and A_{j+1} are set as foreign keys, then $R_i(A_j, A_{j+1}) \rightarrow$ set of pairs $P = \{p | p = (p_1, p_2)\}$, where p_1 is a label-entity pair, $p_1 \in (R_i, PK)$, and p_2 is a nested label-entity pair (p'_1, p'_2), with $p'_1 \in (A_j, a_j)$ and $p'_2 \in (A_{j+1}, a_{j+1})$.

(R2) Given a relational table $R_i(A_1, A_2, ..., A_m)$ excluding the attributes A_j and A_{j+1}, then, $R_i(A_1, A_2, ..., A_m) \rightarrow$ set of pairs $P = \{p | p = (p_1, p_2)\}$, where p_1 is a label-entity pair, $p_1 \in (A, IDs)$, and p_2 is a nested label-entity pair (p'_1, p'_2), with $p'_1 \in (R_i, PK)$ and $p'_2 \in (\mathcal{T}, a)$.

The mapping in rule R1 specifies the binary relation between entities, denoted by foreign keys (*i.e.,* p_2), and uniquely identifies the relation using the table name R_i and primary keys PK (*i.e.,* p_1). In R2, non-foreign key attributes are specified as entities by combining attribute names A and unique identifiers IDs in p_1. p_2 establishes a binary relation between instances identified by the primary key PK of R_i and attribute values a with data types $\mathcal{T}$.

TABLE I
SUMMARY OF VARIOUS MODEL FEATURES

	LPG	RDF	RDF-star	RM	MG	SG	UGR
Vertex Label	✓					✓	✓
Edge Label	✓	✓	✓		✓	✓	✓
Vertex Property	✓	✓	✓	✓	✓	✓	✓
Edge Property	✓		✓	✓	✓	✓	✓
Multiple Edges	✓			✓	✓	✓	✓
Edge as Vertex			✓	✓	✓	✓	✓
Path as Vertex							✓
Path as Edge							✓
Graph as Vertex							✓

C. Comparison of Representational Features between Models

UGR provides support for a rich set of graph representation features. In Table I, we compare the representational capabilities of various models [26], including LPG, RDF, RDF-star, relational model (RM), Multilayer Graphs (MG) [26], Statement Graphs (SG) [27], and UGR used in GETL. Additionally, we provide the following introduction to the features specifically required for graph ETL.

- *Edge/Path/Graph as Vertex*: Edges, specific paths, and entire graphs can be referenced as vertices.
- *Path as Edge*: Specific paths can be referenced as edges.

We emphasize the importance of these features for graph ETL. They are built from different abstraction perspectives. For instance, *Edge as Vertex* allows edges to connect to other vertices within the graph [26], while *Path as Edge* enables the assignment of properties to specific paths. These features allow users to customize graphs at various abstraction layers in the definition layer without the need to initiate complex constructions from scratch. In UGR, edges as vertices are naturally supported by nested label-entity pairs. Entities like paths and graphs are represented through sets of entities, *i.e.,* $(lb, *)$, and set operations, as outlined in Definition 7.

V. PROGRAMMING INTERFACE FOR GETL

GETL serves GraphScope [35], [38], which supports a wide range of computational paradigms and handles various graph tasks, including graph analysis, graph learning, and interactive querying, all within a single unified system designed for rich graphs. GETL is built upon GraphScope's significant investment in the Gremlin [29] stack and introduces a dedicated programming interface that simplifies the development of user-defined ETL operations. This section introduces a high-level overview of the programming interface, which supports graph schema definition and transformation manipulation.

A. Graph Schema Definition

The definition layer allows users to customize graph schemas and perform simple data filtering. To facilitate this, we design a *Data Definition Language* (DDL) as part of GETL's programming interface, utilizing a syntax similar to Gremlin. We illustrate the DDL syntax through Figure 5, which defines the underlying data model described in Example

6 as a shopping graph. First, register a graph named "shopping_graph" using `g.register`. The `Config()` specifies the nested label-entity pairs in the model associated with this graph, such as "{ 'Person': $*$ }", representing all entities associated with the "Person" label. Second, define vertices with `putV()` and indicate the labels of entities set as vertices (*e.g.,* "Person"). Define edges with `putE()`, specifying the edge label (*e.g.,* "buy"), direction `Direction.OUT`, and endpoints. Follow these with `property()` to assign properties and `label()` to reset labels if necessary. Third, DDL supports pruning the newly registered graph using `where()`, filtering structures that meet specific conditions, such as vertices where "age" is greater than 18. Finally, construct identifiers for the newly registered graph with `setIdTransform()`.

```
# define a shopping graph configured by three types of label-entity sets
shopping_graph = g.register.Config({'Person':*}, {'Product':*}, {'buy':*})
      # define vertex with the 'Person' label and add properties
      .putV('Person')
      .property('name').property('age').property('gender')
      .where(values('age').is(gt(18)))
      # define vertex with the 'Product' label and add properties
      .putV('Product')
      .property('name').property('category').property('price')
      # define edge labeled 'buy' from 'Person' to 'Product'
      .putE('buy', Direction.OUT, 'Person', 'Product')
      .property('date').property('quantity').property('discount')
      .where(values('date').is(lt('2021-06-20')))
      .setIdTransform()
```

Fig. 5. Example DDL for defining a shopping graph.

DDL also provides graph schema definitions at various abstraction levels, through different model representational features, such as *Edge/Path/Graph as Vertex* and *Path as Edge*.

B. Graph Transformation Manipulation

GETL defines a set of syntactic rules to describe three types of transformation operations. We demonstrate the programming syntax features and commands for each type.

Basic Syntax. We utilize Gremlin to perform basic transformation operations, *i.e., graph pattern matching*. The Gremlin syntax supports high-level and declarative programming, enabling users to define traversal instructions easily. The traversal source, `g`, serves as an interface for accessing data in the graph, using the operators `V()` and `E()` to select vertices and edges. Moreover, Gremlin provides a comprehensive set of operators for filtering (`has`, `where`), projection (`select`, `by`), looping (`repeat`), aggregation (`group`), sorting (`order`, `limit`), and more. Further details are available in [39].

```
#Q1: query "Person" who purchased "ProductA"
g.V().has('Product','name','productA').in('buy').dedup()
#Q2: query "Person" who purchased both "productA" and "productB"
g.V().match(
      __.as('person').out('buy').has('Product', 'name', 'productA'),
      __.as('person').out('buy').has('Product', 'name', 'productB'))
      .select('person').values('name').dedup()
```

Fig. 6. Gremlin for graph pattern matching.

Figure 6 demonstrates using Gremlin for graph pattern matching. Query `Q1` searches for individuals who purchased "productA". Here, `has()` filters and locates vertices labeled "productA". The `in()` traverses incoming edges labeled "buy" to find vertices connected to "ProductA" vertices. The `dedup()` operator removes duplicates. Query `Q2` uses `match()` for declarative pattern matching, containing two clauses that both start with the alias "person" and confirm whether the person purchased specific products. Finally, `select()` extracts the "name" from matched "Person".

Layer Building. GETL introduces the `layer()` operator to construct new layers that support *special mapping* and rewrites existing Gremlin steps to support its execution semantics. We illustrate this enhancement with the following two use cases.

Figure 7 shows an example of extracting subgraphs from the "Person" graph based on "age" ranges and mapping these to "Cohort" vertices. The process includes several steps: Initially, `layer()` creates a new graph layer, grouping "Person" vertices by "age" intervals of 10 to form vertices labeled "Cohort". The `groupBy()` enables grouping operations using embedded expressions. Subsequently, vertex properties are defined to specify the maximum and minimum ages within the "Cohort". The `unfold()` is then used to expand the "Cohort" vertices and extract their "age" properties. Edges are constructed between "Cohort" vertices, incorporating the original edges between "Person" vertices, with an edge property labeled as "weight" that counts all original edges. Finally, the `finish()` completes all the layer-building operations.

```
# define the graph's abstraction at different layers
cohort_graph = layerGraphBuilder(person_graph)
      .layer(g.V().hasLabel('Person'))
      .label('Cohort').as('cohort')
      # group 'Person' by 'age' range interval of 10
      .groupBy('age', v -> Math.floor(v.get().value('age') / 10) * 10))
      # add maximum and minimum age as vertex property
      .property('maxAge', select('cohort').unfold().values('age').max())
      .property('minAge', select('cohort').unfold().values('age').min())
      .addE('connection', Direction.OUT, 'cohort')
      # add count of original edges as edge property
      .property('weight', select('cohort').bothE().unfold().count())
      .setIdTransform().finish()
```

Fig. 7. Define the graph's abstraction at different layers.

```
path_graph = layerGraphBuilder(person_graph)
      .layer(g.V().hasLabel('Person'))
      .label('Person')
      # map 3-hop paths to edges
      .addE(g.V().as('start')
              .repeat(out().simplePath()).times(3).as('end')
              .path(),'connection', Direction.OUT, 'start', 'end')
      .setIdTransform().finish()
```

Fig. 8. Map three-hop paths to edges.

Figure 8 provides another example, where three-hop paths are mapped directly as edges. The `addE()` accepts a `path()` parameter, utilizing the "start" and "end" vertices of the path as the two endpoints of the new edge.

User-defined Recursive Graph Traversal. User-defined recursive graph traversals are an essential tool for complex transformation operations, yet current approaches lack adequate support. Figure 9 illustrates a standard method, ShortestPathVertexProgram, provided by TinkerPop [39] for solving the single-source shortest path problem (SSSP), which includes several configurable parameters. However, this method is overly rigid, thereby restricting users' operational flexibility. In addition, as depicted in Figure 10, implementing SSSP (*i.e.,* vertex-centric iterative algorithms) using Gremlin scripts requires maintaining additional vertex states, along with complex iteration control and arithmetic expressions, significantly increasing the program's complexity.

```
# create a shortest path vertex program and configure it
spvp = ShortestPathVertexProgram.build()
            .source(sourceId).distanceProperty('weight').create()
result = graph.compute().program(spvp).submit().get()
# output the shortest path from the results memory
result.memory().get(ShortestPathVertexProgram.SHORTEST_PATHS)
```

Fig. 9. ShortestPathVertexProgram for SSSP.

```
# define the source vertex and initialize distances
def sourceId = 0
g.V().property('distance',
                __.choose(__.id().is(sourceId), 0, Double.MAX_VALUE))
                .iterate()
def vertexCount = g.V().count().next()
# |V| - 1 iterations to find the shortest path
g.V().repeat(
     __.as('v').outE().as('e').inV().as('w')
     .select('e', 'w').by('weight').as('b').by('distance').as('c')
     .by(select('v').values('distance')).as('a')
     .where(__.math('a + b').is(lt('c')))
     .select('w').property('distance', __.math('a + b')))
.times(vertexCount - 1).iterate()
```

Fig. 10. An implementation of SSSP via a Gremlin script.

To address the above issues, we develop a hybrid Data Manipulation Language (DML) that combines Datalog and Gremlin. We choose the Datalog + Gremlin combination based on the following observations. First, Datalog and Gremlin have been widely used in knowledge-oriented applications [40] and graph applications [35] domains, making them easy for new users to learn. Second, Gremlin can easily express complex navigational graph queries, excelling in path filtering and control, but handling recursive/iterative logic is challenging, which is where Datalog excels. Datalog uses concise rule definitions for recursion and possesses logical reasoning capabilities. Moreover, its rule definition approach is well-suited for managing intermediate states and arithmetic expressions in iterative computations. Thus, combining Gremlin and Datalog leverages the strengths of both. The syntax is as follows:

```
FROM GRAPH example_graph
....
R(x_1, ..., x_n, [Aggregate(z)]) :- B_1(x_1, ..., x_n, z);
                                    ...;
                                 :- B_m(x_1, ..., x_n, z).
-------------------------------------------------------------
 where ∀_{1≤i≤m} B_i(x_1, ..., x_n, z): R(x_1, ..., x_n, y),
                                       <Gremlin-expression>,
                                       P_1(x_1, ..., x_n,  c),
                                       ...,
                                       P_l(x_1, ..., x_n,  c),
                                       z = Function(y,  c),
                                       [Condition(z)];
```

The program begins with the FROM GRAPH clause, specifying the input graph view. The core of the program consists of a finite set of *rules*. Within these rules, the predicate R on the left side of ":-" serves as the rule head. Aggregate specifies an aggregation function applied to the arguments z. The right side of ":-" contains multiple rule bodies B, separated by semicolons, where each B can include multiple predicates separated by commas. In the rule body B, the predicate R with the same name as the head predicate forms a recursive rule. The Gremlin-expression guides recursive traversal on the graph view, while the predicate P provides parameters for the computation of z-Function. In addition, some programs include rules with conditions for terminating recursion, *e.g.*, [sum[Δrank] $<$ 0.001] in delta-based PageRank [41].

Figure 11 presents an example program using hybrid DML for SSSP, consisting of two rules. The first rule identifies the source vertex and sets the initial distance. The second rule recursively calculates the distance dis_y from the source to y, based on the path length dis_x from vertex x and the edge weight between x and y. Gremlin expressions guide edge filtering and path matching throughout this process. The aggregation function min determines the current shortest path to y. Compared to the previously mentioned Gremlin implementation of SSSP, the hybrid DML requires only two SSSP rules but enhances the flexibility of algorithm design.

```
FROM GRAPH road_graph
SSSP(x, dis)          :- x = 0, dis = 0.
SSSP(y, min[dis_y])   :- SSSP(x, dis_x),
                         g.V(x).outE().has('weight').as('e')
                          .inV().hasLabel('location').as('y'),
                         dis_y = dis_x + e.weight.
```

Fig. 11. Example SSSP via hybrid DML.

```
# weighted random walk with a maximum step count of i + 1
FROM GRAPH social_graph
degree(x, count[y])          :- g.V(x).bothE().as('y').
WRW(0, x)                    :- x = 0.
WRW(i + 1, random[y, w_y])   :- WRW(i, x),
                                g.V(x).outE().has('weight').as('e')
                                 .inV().hasLabel('Person').as('y'),
                                degree(y, d),
                                w_y = 0.35 * e.weight + 0.65 * d.
```

Fig. 12. Example weighted random walk via hybrid DML.

Another example, shown in Figure 12, involves a user-defined weighted random walk algorithm. This algorithm adjusts the transition probabilities of the random walk based on the user-defined importance of each vertex, aiming to identify interest groups within social networks. Specifically, the algorithm incorporates both vertex degree and edge weight to determine the importance w_y of vertex y. Then, w_y is used as a parameter to adjust the random selection process for determining the next vertex in the walk.

VI. System Implementation

We propose a graph ETL framework, GETL. Here, we discuss the details of some components.

Interface. GETL's data interface supports multiple input and output data formats for graph models, such as CSV, JSON, XML, and TTL. Additionally, GETL facilitates direct integration with data management and analysis systems, such as GraphScope [35], MySQL [42], Neo4j [31], Amazon Neptune [43], and TinkerPop [32]. GETL also provides programming interfaces for the entire ETL process, with the main components listed in Table II. These interfaces include system-level APIs and extended Gremlin steps based on Gremlin v3.4.9 [29]. Furthermore, GETL leverages existing entity alignment approaches [44], [45] to address data consistency.

Pipelining. GETL employs a pipeline design that overlaps the data loading and conversion to the UGR steps to enhance extraction efficiency. We use the Disruptor [46], where the producer asynchronously inserts loaded batch data into a ring

TABLE II
GETL PROGRAMMING INTERFACES.

Function Name	Description
refreshLPG()/refreshRM()/refreshRDF()	Convert UGR to LPG / RM / RDF.
loadLPG(LPGGraph lpgGraph)/loadRM(RMGraph rmGraph)/loadRDF(RDFGraph rdfGraph)	Load LPG / RM / RDF into UGR.
addRMSchema(RMSchema rmSchema)	Add the RM schema for mapping rules with UGR.
addLPGMappingConfig(LPGConfig lpgConfig)	Add user-customized graph configuration.
readRDFFile(RdfFormat rdfFormat, String filePath)	Read RDF files / MySQL to RMGraph / LPG vertex and edge (including Property) files.
queryFromMySQL(MysqlSessions sessions)	
loadVertex(String fileName, String vertexLabel, String...properties)	
loadEdge(String filePath, String edgeLabel, String from, String to, String...properties)	
Modified or Extended Gremlin Steps	**Enhanced Gremlin Functions**
register(), Config(), putV(), putE(), property(), label(), setIdTransform()	Define the graph schema.
layer(), groupBy(), unfold(), addE(), drop(), path(), groupCount(), count(), materialize(), finish()	Build a new graph layer.

buffer, and the consumer retrieves the data from the ring buffer in a non-blocking manner and converts it into UGR.

Indexes. GETL does not allocate new storage for user-customized graph schemas but instead constructs indexes based on underlying nested pairs. The advantage lies in a more streamlined and efficient construction of graph schemas, which also reduces memory consumption. Additionally, since the definition layer supports only simple transformations, using indexes does not significantly impact transformation performance. A prudent approach involves adding indexes when constructing nested pairs if the user has already defined the graph schema, thereby minimizing redundant data traversal. In addition, GETL provides the materialize() interface, which allows intermediate graph layers to physically materialize the constructed temporary graphs.

Garbage Collection (GC). Constructing different graph layers typically incurs significant storage overhead. Additionally, GETL forgoes the optimization of grouped management [47] for pair structures to preserve efficient integration capabilities, resulting in redundant storage. Therefore, as an in-memory framework, effective GC is crucial for GETL to prevent excessive memory consumption during continuous operations. GETL performs data cleanup in stages. After constructing a new graph layer, GETL traverses the previous layer to identify unnecessary data objects based on the connections between the current and previous layers. The GC thread then releases these objects. Furthermore, GETL also supports selective retention of data from any layer to serve data integration and reuse.

VII. Experimental Evaluation

A. Experimental Setup

Environments. All the experiments in this section are conducted on an AliCloud ECS server (ecs.c7.16xlarge instance) equipped with 64 vCPU cores and 512GB of memory.

Datasets. We use MovieLens [48] under the relational model, which contains 18 columns and 41,741,454 rows. We select Wikipedia [49] for the RDF graph, consisting of 22,791,171 triples. As a widely used benchmark, the LDBC SNB graph [50] serves as the LPG. It comprises 3,966,203 vertices and 23,031,794 edges, along with 19,736,348 vertex properties and 6,854,988 edge properties. Furthermore, we use three sub-datasets from the LDBC graph to validate data integration: LPG $G1$, RDF graph $G2$, and graph $G3$ under the relational model. Specifically, edges with and without properties in the LDBC graph are respectively distributed in $G1$ and $G2$. $G2$ partially holds vertex properties, while the remainder is maintained in $G3$. The detailed data scales are as follows: $G1$ contains 3,966,203 vertices, 19,740,638 edges, and 6,854,988 edge properties. $G2$ consists of 14,883,937 triples, and $G3$ includes 23 columns and 3,948,592 rows.

Competitors. In the conversion performance comparison of the unified model (UM), we use Multilayer Graphs (MG) and Statement Graphs (SG) as competitors. Both support representations for LPG, RDF, and RDF-star. We choose GraphScope [35] and Ra-SQL [51] for comparison, both of which have designed DMLs capable of handling recursive algorithms based on traversal queries. GraphScope extends the "Scatter-Gather" step on Gremlin to support vertex-centric computation logic. RaSQL supports recursive aggregation through an extension of the SQL standard.

B. Model Conversion Performance

We first evaluate the performance of MG, SG, and UGR in bidirectional mapping between the LPG, RDF (including RDF-star), and the relational model (RM). As shown in Figure 13, the mapping is divided into two directions: from graph models to the unified model (UM), *e.g.,* LPG2UM, and from the unified model to specific graph models, *e.g.,* UM2LPG. In most cases, UGR demonstrates good performance. When mapping graph models to the unified model, the SG performs a detailed categorization of different elements within the graph models. Although this increases overhead, it also enhances the efficiency of mapping operations to specific graph models. The MG feature that assigns IDs to relationships can be naturally built into RDF triples, resulting in significant benefits. The UGR benefits from the simplicity of its mapping rules. Additionally, UGR explicitly provides a bidirectional mapping with RM. We also evaluate the runtime for conversions between LPG and RDF through the unified model, *i.e.,* LPG2RDF and RDF2LPG, and UGR outperforms others in both cases.

C. Graph ETL Execution Performance

We introduce seven queries to describe the graph ETL process under diverse requirements and evaluate the overall performance of GETL in executing these queries. These queries involve input and output formats across different models and

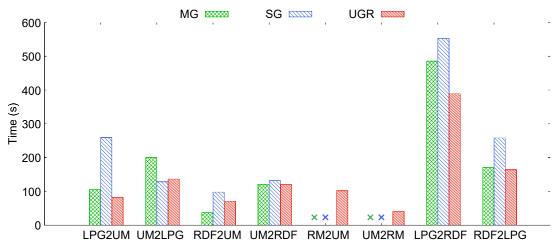


Fig. 13. Runtime of model conversion.

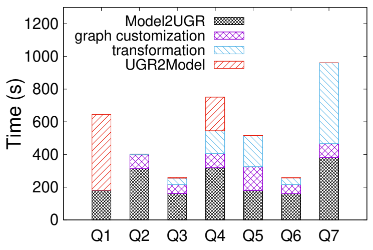


Fig. 14. Runtime breakdown.

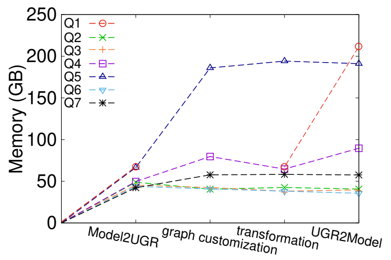


Fig. 15. Memory cost by stage.

various types of transformation operations, encompassing the core functionalities of GETL and hold practical relevance.

Q1: Directly convert MovieLens from RM to an RDF graph.

Q2: In the LDBC graph, execute a custom graph schema operation to identify vertices representing individuals who posted on forums with more than 20 members from 2010-01 to 2010-05, and retrieve the received comments. Output the results in RM.

Q3: In Wikipedia, construct graph schema based on IRIs, filter vertices with a degree greater than 20 along with connected edges, and convert these results to LPG.

Q4: In the LDBC graph, identify the paths: personA $\rightarrow$ comment $\rightarrow$ post $\rightarrow$ personB, and map the path to edge labeled "isFanOf" from personA to personB. Then, output in RDF.

Q5: In MovieLens, categorize movies into different subgraphs based on the "genome-scores.relevance" property and map each subgraph to a vertex. Construct edges between these subgraphs by identifying users who rate movies in both subgraphs with a score of 5, labeling these edges as "relevance" to reflect the relevance of user preferences. Additionally, count the occurrences of such rating patterns to use as an edge property, representing the weight of movie category relevance. Finally, export the graph layer as LPG.

Q6: In Wikipedia, similar to query Q3, extract vertices with a degree greater than 20, along with their connecting edges, to construct an undirected graph. Apply the Connected Component algorithm [52] to identify subgraphs. Finally, output these connected subgraphs as relational tables.

Q7: Integrating $G1$, $G2$, and $G3$ from different models enriches the properties on vertices and edges. Subsequently, for community detection among user groups, conduct a biased two-hop random walk on the integrated graph to sample paths with edge labels "likes" or "hasCreator": person $\rightarrow$ post $\rightarrow$ person. Calculate transition probabilities based on tag similarity (intersection of person and post tags) and the number of comments and likes on the post, which influence the selection of vertices at each hop. The sampled paths are then mapped to new edges, *i.e.,* person $\rightarrow$ person, with "common_interest" labels, and the results are converted into RM.

Table III shows the overall runtime for seven queries in GETL. The results indicate that the performance of GETL is within an acceptable range for all cases. Q1 involves only model conversion operations yet incurs significant runtime due to substantial conversion overhead. This is because RDF requires a large number of triples to adequately describe the contents of relational tables. Q2 showcases a user customizing a graph schema at the definition layer, while Q3 further executes graph pattern matching on the customized graph. Both queries involve basic graph ETL operations, resulting in relatively shorter runtimes. Q4 and Q5 are designed to construct complex graph layers to perform special mappings, which are built upon a higher level of transformation over the customized graph, resulting in greater overhead. The operations involved in Q6 are straightforward, and compared to Q3, it incurs less model conversion time, resulting in minimal time overhead. Q7 demonstrates GETL's capability to integrate data from different models and executes a biased graph sampling on the integrated graph. The query involves complex multi-layer transformation operations, including customizing the integrated graph schema, executing a user-defined recursive query based on specific paths, and mapping the sampled paths to edges, making it time-consuming.

TABLE III
RUNTIME OF THE GRAPH ETL PROCESS

Query	Dataset	Mapping	Time (s)	Output Scale
Q1	MovieLens	RM2RDF	**645.36**	triples: 108,962,057
Q2	LDBC	LPG2RM	**402.83**	rows: 105,430
Q3	Wikipedia	RDF2LPG	**259.49**	vertices: 225,415 edges: 1,915,096
Q4	LDBC	LPG2RDF	**751.06**	triples: 54,009,423
Q5	MovieLens	RM2LPG	**517.26**	vertices: 841 edges: 298,297
Q6	Wikipedia	RDF2RM	**257.75**	rows: 1,713,388
Q7	$G1$, $G2$, $G3$	(LPG, RM, RDF)2RM	**961.03**	rows: 19,717

D. Runtime Breakdown

We next evaluate the runtime of GETL at each stage throughout the execution of various queries. The total runtime includes the following stages: data loading and extraction (Model2UGR), graph schema customization (graph customization), transformation operations (Transformation), and output model conversion (UGR2Model). The results are shown in Figure 14. During the Model2UGR stage, GETL leverages pipeline optimization to overlap data loading and conversion to UGR steps, significantly reducing runtime. In Q1, the mapping from UGR to RDF occupies the majority of the time due to the generation of a large volume of RDF triples. A similar phenomenon is observed in Q4. For Q4 and Q5, the transformation stage involves searching for path and subgraph patterns and constructing new graph layers, which results in longer times. The overhead in Q7 primarily arises from the serial integration of three independent datasets and the execution of complex, multi-layer transformation operations.

E. Memory Cost

We further evaluate the memory cost. GETL performs a phased garbage collection (GC) strategy to optimize memory

TABLE IV
INTELLIGIBILITY METRICS (P_E) FOR VARIOUS DMLs

	SSSP	CC	PR	CP
Gremlin	6939.3	568.87	8401.77	**114.12**
GraphScope	3233	1512.67	5614.41	240.53
RaSQL	2712.26	1580.23	7227.4	1856.95
GETL	**1437.39**	**317.84**	**3849.66**	990.4

usage. Figure 15 reports the trend of memory cost changes at different stages of query execution. It can be observed that GETL maintains stable memory usage in most cases. Additionally, the method of building indexes based on underlying nested pairs ensures that memory growth is not significant for most queries during the graph customization phase. The sharp increase in memory during the UGR2Model stage of Q1 is due to the mapping generating a large number of RDF triple results. In Q5, we materialize graph schema to efficiently group subgraphs and condense edges. Moreover, these grouped vertices and edges need to be preserved to support the newly mapped layer. Thus leading to limited memory savings.

F. Intelligibility of DMLs

We finally analyze the intelligibility of programs for user-defined recursive queries. Intelligibility refers to the conciseness of DML expressions, which is crucial for understanding the program and enabling non-expert users to master it quickly. We use *Halstead Metrics* [53] for quantitative measurement, which analyzes program effort, denoted as P_E (where $P_E = P_D \cdot P_V$), by combining the operators and operands in the code. It involves two metrics: the program volume $P_V = (N_1 + N_2) \cdot log_2(n_1 + n_2)$ and the program difficulty $P_D = (n_1/2) \cdot (N_2/n_2)$, where n_1 and n_2 are the numbers of distinct operators and operands, and N_1 and N_2 are the total numbers of operators and operands, respectively.

We compare the program effort of algorithms implemented using the Gremlin, GraphScope, RaSQL, and hybrid DML of GETL. The algorithms include Single-Source Shortest Path (SSSP) [54], Connected Component (CC) [52], PageRank (PR) [55], and Count Paths (CP) [56]. Partial examples of the algorithms are available in their respective papers. The results are shown in Table IV. It can be observed that in most cases, GETL requires the least program effort P_E. Gremlin and GraphScope excel at the CP algorithm because CP does not require complex recursive aggregation logic or temporary variables, relying primarily on traversal operations, which is a core strength of Gremlin-like syntax. RaSQL demonstrates strong intelligibility by extending SQL's declarative syntax, but it struggles to handle complex join operations succinctly.

VIII. RELATED WORKS

Conversion between data models. Numerous efforts have been dedicated to establishing direct mappings between the LPG, RDF, RDF-star, and the relational model [12], [16], [17], [19]–[21], [57]–[61]. Although these studies offer valuable insights into model-to-model conversions, they are not the ideal solution for GETL 's unified approach to supporting interoperability among multiple models. On the other hand, OneGraph [62], Multilayer Graphs [26], and Statement Graphs [27] achieve interoperability between LPG, RDF, and RDF-star from the perspective of a unified data model. OneGraph *outlines a vision* for the unification of graph models and explores the technical challenges involved. Multilayer Graph builds relationships across multiple layers by adding edge IDs as connectors. Statement Graph constructs a directed acyclic graph centered on statements. While they are intuitive and general, they lack direct support for specific features in graph ETL, such as paths as vertices. Therefore, we emphasize that UGR is a domain-specific data model designed to address these specialized requirements.

Programming Interfaces. Some efforts have been made for graph queries from two perspectives: (1) Describing graph analysis algorithms. StarPlat [63] enables code generation for multi-core, distributed, and many-core systems based on the same interface specification. Fregel [64] proposes a functional approach to vertex-centric graph processing, abstracting the computation of each vertex into a high-order function. GraphIt [65] facilitates the rapid implementation of algorithms with varying performance characteristics. (2) Developing graph query languages for graph databases [29], [30], [33], [66]–[71]. However, graph analytics programming interfaces fail to effectively support analysis on graph traversal queries, as they typically require predefined graphs as input. Graph database query languages, although effective for graph pattern matching, lack adequate support for special mappings and user-defined recursive queries. Conversely, GETL introduces a more targeted programming interface, specifically tailored for the graph ETL process.

ETL tools for graph data. Several ETL tools have been developed for graph data. Neo4j [31] is equipped with an ETL tool that enables the importation of data from relational tables to Neo4j. However, this tool supports only a single mapping for LPG. It does not accommodate special mappings, such as Q4 and Q5. Table2Graph [72] facilitates the conversion of multiple relational tables into an LPG-based ETL process. Berro et al. [73] proposes an ETL chain for warehousing based on RDF. GraphBuilder [74] is designed to alleviate many complexities of graph construction, such as graph formation, tabulation, and partitioning. The motivation behind GETL fundamentally differs from these tools. GETL seeks to bridge the graph ETL process across diverse data models, thereby enhancing interactions between applications and systems.

IX. CONCLUSIONS

This work has made the first effort to develop a graph ETL framework specialized for various graph models. The highlights of GETL include: (1) the design of a unified graph representation model that is compatible with various graph models while ensuring robust data integration capabilities; (2) a layered architecture that offers considerable flexibility, enabling the customization of graph schemas and progressive execution of graph transformations with varying complexity; (3) an expressive programming interface tailored for the graph ETL process, designed to support specialized operations inherent to the graph ETL workflow.

References

[1] M. Mountantonakis and Y. Tzitzikas, “Large-scale semantic integration of linked data: A survey,” *CSUR*, vol. 52, no. 5, pp. 1–40, 2019.

[2] M. Buron, F. Goasdoué, I. Manolescu, and M.-L. Mugnier, “Obi-wan: ontology-based rdf integration of heterogeneous data,” in *PVLDB*, 2020.

[3] R. Angles, “The property graph database model.” in *AMW*, 2018.

[4] H. R. Vyawahare, P. P. Karde, and V. M. Thakare, “A hybrid database approach using graph and relational database,” in *RICE*, 2018.

[5] P. Yi, L. Liang, D. Zhang, Y. Chen, J. Zhu, X. Liu, K. Tang, J. Chen, H. Lin, L. Qiu *et al.*, “Kgfabric: A scalable knowledge graph warehouse for enterprise data interconnection,” in *PVLDB*, 2024.

[6] X. Li, W. Zeng, Z. Wang, D. Zhu, J. Xu, W. Yu, and J. Zhou, “Graphar: An efficient storage scheme for graph data in data lakes,” *PVLDB*, vol. 18, no. 3, p. 530–543, 2024.

[7] H. Han, Y. Wang, H. Shomer, K. Guo, J. Ding, Y. Lei, M. Halappanavar, R. A. Rossi, S. Mukherjee, X. Tang *et al.*, “Retrieval-augmented generation with graphs (graphrag),” *arXiv preprint arXiv:2501.00309*, 2024.

[8] K. Guo, H. Shomer, S. Zeng, H. Han, Y. Wang, and J. Tang, “Empowering graphrag with knowledge filtering and integration,” *arXiv preprint arXiv:2503.13804*, 2025.

[9] Taobao, “Taobao,” https://www.taobao.com/, 2025.

[10] S. Yu, S. Gong, Q. Tao, S. Shen, Y. Zhang, W. Yu, P. Liu, Z. Zhang, H. Li, X. Luo *et al.*, “Lsmgraph: A high-performance dynamic graph storage system with multi-level csr,” *SIGMOD*, vol. 2, no. 6, pp. 1–28, 2024.

[11] H. Li, Q. Tao, S. Yu, S. Gong, Y. Zhang, F. Yao, W. Yu, G. Yu, and J. Zhou, “Gastcoco: Graph storage and coroutine-based prefetch co-design for dynamic graph processing,” *PVLDB*, vol. 17, no. 13, p. 4827–4839, 2025.

[12] S. Khayatbashi, S. Ferrada, and O. Hartig, “Converting property graphs to RDF: a preliminary study of the practical impact of different mappings,” in *SIGMOD*, 2022.

[13] A. Group, “Graphscope,” https://graphscope.io/journey/, 2025.

[14] A. Amiri, “Designing a distribution network in a supply chain system: Formulation and efficient solution procedure,” *Eur. J. Oper. Res.*, vol. 171, no. 2, pp. 567–576, 2006.

[15] D. Tomaszuk, “RDF data in property graph model,” in *MTSR*, 2016.

[16] D. Tomaszuk, R. Angles, and H. Thakkar, “PGO: describing property graphs in RDF,” *IEEE Access*, vol. 8, pp. 118 355–118 369, 2020.

[17] V. M. de Sousa and L. M. del Val Cura, “Logical design of graph databases from an entity-relationship conceptual model,” in *iiWAS*, 2018.

[18] E. Bytyçi, L. Ahmedi, and G. Gashi, “RDF mapper: Easy conversion of relational databases to RDF,” in *WEBIST*, 2018.

[19] G. Abuoda, D. Dell’Aglio, A. Keen, and K. Hose, “Transforming rdf-star to property graphs: A preliminary analysis of transformation approaches,” in *ISWC*, 2022.

[20] R. Angles, H. Thakkar, and D. Tomaszuk, “Mapping rdf databases to property graph databases,” *IEEE Access*, vol. 8, pp. 86 091–86 110, 2020.

[21] K. Rabbani, M. Lissandrini, A. Bonifati, and K. Hose, “Transforming rdf graphs to property graphs using standardized schemas,” *SIGMOD*, vol. 2, no. 6, pp. 1–25, 2024.

[22] J. E. L. Gayo, E. Prud’Hommeaux, I. Boneva, and D. Kontokostas, *Validating RDF data*. Morgan & Claypool Publishers, 2017.

[23] D. Wu, H.-T. Wang, and A. U. Tansel, “A survey for managing temporal data in rdf,” *Information Systems*, vol. 122, p. 102368, 2024.

[24] R. Angles, A. Bonifati, S. Dumbrava, G. Fletcher, A. Green, J. Hidders, B. Li, L. Libkin, V. Marsault, W. Martens *et al.*, “Pg-schema: Schemas for property graphs,” *SIGMOD*, 2023.

[25] D. W. Wardani and J. Kiing, “Semantic mapping relational to graph model,” in *IC3INA*, 2014.

[26] R. Angles, A. Hogan, O. Lassila, C. Rojas, D. Schwabe, P. A. Szekely, and D. Vrgoc, “Multilayer graphs: a unified data model for graph databases,” in *GRADES & NDA*, 2022.

[27] E. Gelling, G. Fletcher, and M. Schmidt, “Statement graphs: Unifying the graph data model landscape,” in *DASFAA*, 2024, pp. 364–376.

[28] R.-D. C. Group, “Rdf-star and sparql-star,” https://www.w3.org/2021/12/rdf-star.html, 2021.

[29] M. A. Rodriguez, “The gremlin graph traversal machine and language (invited talk),” in *DBPL*, 2015.

[30] N. Francis, A. Green, P. Guagliardo, L. Libkin, T. Lindaaker, V. Marsault, S. Plantikow, M. Rydberg, P. Selmer, and A. Taylor, “Cypher: An evolving query language for property graphs,” in *SIGMOD*, 2018.

[31] “Neo4j,” https://neo4j.com/, 2025.

[32] “Tinkerpop,” https://tinkerpop.apache.org/, 2025.

[33] A. Deutsch, N. Francis, A. Green, K. Hare, B. Li, L. Libkin, T. Lindaaker, V. Marsault, W. Martens, J. Michels *et al.*, “Graph pattern matching in gql and sql/pgq,” in *SIGMOD*, 2022.

[34] P. Zhao, X. Li, D. Xin, and J. Han, “Graph cube: on warehousing and olap multidimensional networks,” in *SIGMOD*, 2011.

[35] W. Fan, T. He, L. Lai, X. Li, Y. Li, Z. Li, Z. Qian, C. Tian, L. Wang, J. Xu, Y. Yao, Q. Yin, W. Yu, K. Zeng, K. Zhao, J. Zhou, D. Zhu, and R. Zhu, “Graphscope: A unified engine for big graph processing,” in *PVLDB*, 2021.

[36] D. Beckett, T. Berners-Lee, E. Prud’hommeaux, and G. Carothers, “Rdf 1.1 turtle,” https://www.w3.org/TR/2014/REC-turtle-20140225/, 2014.

[37] O. Hartig, P.-A. Champin, G. Kellogg, and A. Seaborne, “Rdf-star and sparql-star,” https://www.w3.org/2021/12/rdf-star.html, 2021.

[38] T. He, S. Hu, L. Lai, D. Li, N. Li, X. Li, L. Liu, X. Luo, B. Lyu, K. Meng *et al.*, “Graphscope flex: Lego-like graph computing stack,” in *SIGMOD*, 2024.

[39] “Tinkerpop documentation,” https://tinkerpop.apache.org/docs/3.7.2/reference/, 2025.

[40] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo, “Big data analytics with datalog queries on spark,” in *SIGMOD*, 2016.

[41] Y. Zhang, Q. Gao, L. Gao, and C. Wang, “Maiter: An asynchronous graph processing framework for delta-based accumulative iterative computation,” *TPDS*, vol. 25, no. 8, pp. 2091–2100, 2013.

[42] “Mysql,” https://www.mysql.com/, 2025.

[43] “Amazon neptune,” https://aws.amazon.com/cn/neptune/, 2023.

[44] C. Ge, X. Liu, L. Chen, B. Zheng, and Y. Gao, “Largeea: Aligning entities for large-scale knowledge graphs,” in *PVLDB*, 2022.

[45] K. Xin, Z. Sun, W. Hua, W. Hu, J. Qu, and X. Zhou, “Large-scale entity alignment via knowledge graph merging, partitioning and embedding,” in *CIKM*, 2022.

[46] M. Thompson, D. Farley, M. Barker, P. Gee, and A. Stewart, “Disruptor: High performance alternative to bounded queues for exchanging data between concurrent threads,” *Technical paper. LMAX*, vol. 206, 2011.

[47] W. Martens, M. Niewerth, T. Popp, C. Rojas, S. Vansummeren, and D. Vrgoc, “Representing paths in graph database pattern matching,” in *PVLDB*, 2023.

[48] “Movielens,” https://grouplens.org/datasets/movielens/25m/, 2019.

[49] “Wikipedia,” https://databus.dbpedia.org/dbpedia/mappings/mappingbased-objects/2022.12.01/mappingbased-objects_lang=en.ttl.bz2, 2022.

[50] “Ldbc social network benchmark graphs,” https://repository.surfsara.nl/datasets/cwi/ldbc-snb-interactive-v1-datagen-v100, 2022.

[51] J. Gu, Y. H. Watanabe, W. A. Mazza, A. Shkapsky, M. Yang, L. Ding, and C. Zaniolo, “Rasql: Greater power and performance for big data analytics with recursive-aggregate-sql on spark,” in *SIGMOD*, 2019.

[52] T. Hsu, V. Ramachandran, and N. Dean, “Parallel implementation of algorithms for finding connected components in graphs,” in *DIMACS*, 1994.

[53] M. H. Halstead, *Elements of Software Science (Operating and programming systems series)*. Elsevier Science Inc., 1977.

[54] V. T. Chakaravarthy, F. Checconi, P. Murali, F. Petrini, and Y. Sabharwal, “Scalable single source shortest path algorithms for massively parallel systems,” *TPDS*, vol. 28, no. 7, pp. 2031–2045, 2017.

[55] M. Bianchini, M. Gori, and F. Scarselli, “Inside pagerank,” *ACM Trans. Internet Techn.*, vol. 5, no. 1, pp. 92–128, 2005.

[56] E. T. Bax, “Algorithms to count paths and cycles,” *Inf. Process. Lett.*, vol. 52, no. 5, pp. 249–252, 1994.

[57] G. Abuoda, D. Dell’Aglio, A. Keen, and K. Hose, “Transforming rdf-star to property graphs: A preliminary analysis of transformation approaches - extended version,” *CoRR*, vol. abs/2210.05781, 2022.

[58] O. Hartig, “Reconciliation of rdf* and property graphs,” *CoRR*, vol. abs/1409.3288, 2014. [Online]. Available: http://arxiv.org/abs/1409.3288

[59] R. Angles, H. Thakkar, and D. Tomaszuk, “RDF and property graphs interoperability: Status and issues,” in *AMW*, 2019.

[60] G. Yuan, J. Lu, Z. Yan, and S. Wu, “A survey on mapping semi-structured data and graph data to relational data,” *ACM Comput. Surv.*, vol. 55, no. 10, pp. 218:1–218:38, 2023.

[61] O. Hartig, “Foundations to query labeled property graphs using SPARQL,” in *SEMANTiCS*, 2019.

[62] O. Lassila, M. Schmidt, O. Hartig, B. Bebee, D. Bechberger, W. Broekema, A. Khandelwal, K. Lawrence, C. López-Enríquez, R. Sharda, and B. B. Thompson, “The onegraph vision: Challenges of breaking the graph model lock-in^1,” *Semantic Web*, vol. 14, no. 1, pp. 125–134, 2023.

[63] N. Behera, A. Kumar, E. R. T, S. Nitish, R. P. Muniasamy, and R. Nasre, “Starplat: A versatile DSL for graph analytics,” *CoRR*, vol. abs/2305.03317, 2023.
[64] K. Emoto, K. Matsuzaki, Z. Hu, A. Morihata, and H. Iwasaki, “Think like a vertex, behave like a function! a functional DSL for vertex-centric big graph processing,” in *ICFP*, 2016.
[65] Y. Zhang, M. Yang, R. Baghdadi, S. Kamil, J. Shun, and S. P. Amarasinghe, “Graphit: a high-performance graph DSL,” in *OOPSLA*, 2018.
[66] W3C, “Sparql 1.1 query language,” https://www.w3.org/TR/sparql11-query/, 2013.
[67] A. Deutsch, Y. Xu, M. Wu, and V. Lee, “Tigergraph: A native mpp graph database,” in *arXiv:1901.08248*, 2019.
[68] O. Van Rest, S. Hong, J. Kim, X. Meng, and H. Chafi, “Pgql: a property graph query language,” in *GRADES*, 2016.
[69] Z. Pan, T. Wu, Q. Zhao, Q. Zhou, Z. Peng, J. Li, Q. Zhang, G. Feng, and X. Zhu, “Geaflow: A graph extended and accelerated dataflow system,” *Proc. ACM Manag. Data*, vol. 1, no. 2, pp. 191:1–191:27, 2023.
[70] R. Angles, M. Arenas, P. Barceló, P. A. Boncz, G. H. L. Fletcher, C. Gutierrez, T. Lindaaker, M. Paradies, S. Plantikow, J. F. Sequeda, O. van Rest, and H. Voigt, “G-CORE: A core for future graph query languages,” in *SIGMOD*, 2018.
[71] “A new standard for a property graph database language,” https://www.iso.org/standard/76120.html, 2024.
[72] S. Lee, B. H. Park, S. Lim, and M. Shankar, “Table2graph: A scalable graph construction from relational tables using map-reduce,” in *Big-DataService*, 2015.
[73] A. Berro, I. Megdiche, and O. Teste, “Graph-based ETL processes for warehousing statistical open data,” in *ICEIS*, 2015.
[74] N. Jain, G. Liao, and T. L. Willke, “Graphbuilder: scalable graph ETL framework,” in *GRADES*, 2013.

Feng Yao received the MS degree in computer science from Northeastern University, China, in 2021. He is currently working toward a PhD degree in computer science at Northeastern University, China. His research interests include distributed graph processing and data mining.

Xiaokang Yang is currently a Master’s student in Computer Science at Northeastern University, China. His research interests include distributed and parallel computation and data mining.

Shufeng Gong received the PhD degree in computer science from Northeastern University, China, in 2021. He is currently a lecturer with Northeastern University, China. His research interests include cloud computing, distributed graph processing, and vector databases.

Qian Tao recieved the PhD degree in computer science and technology from Beihang University, China in 2021. He is currently an engineer in Alibaba Group, China. His research interests include graph neural networks, graph computations, and large language models.

Yanfeng Zhang received the PhD degree in computer science from Northeastern University, China, in 2012. He is currently a professor with Northeastern University, China. His research consists of distributed systems and big data processing. He has published many papers in the above areas. His paper in SoCC 2011 was honored with ”Paper of Distinction”.

Wenyuan Yu is a Senior Staff Engineer and Director at Alibaba Group and a Ph.D. graduate from the University of Edinburgh. At Alibaba, he leads the Fusion Computing team at the Institute for Intelligent Computing, focusing on machine learning systems and graph computing. Wenyuan is the founder and project lead of GraphScope, Alibaba’s open-source large-scale graph computing system, and the CNCF’s data sharing system, Vineyard. His research, published in top-tier international conferences and journals, has earned him recognition including Best Paper at SIGMOD 2017 and VLDB 2010, and the SIGMOD Research Highlight Award in 2018. Prior to Alibaba, Wenyuan was a founding member of 7Bridges Ltd and a Research Scientist at Facebook.

Ge Yu (Senior Member, IEEE) received the PhD degree in computer science from the Kyushu University of Japan, in 1996. He is now a professor with Northeastern University, China. His current research interests include distributed and parallel systems, cloud computing, big data management, and blockchain techniques and systems. He has published more than 200 papers in refereed journals and conferences. He is the CCF fellow and the ACM member.