GalaxyWeaver: Autonomous Table-to-Graph Conversion and Schema Optimization with Large Language Models

Bing Tong
CreateLink & HKUST(GZ)
btong799@connect.hkust-gz.edu.cn

Yan Zhou*
CreateLink
zhouyan@createlink.com

Chen Zhang
CreateLink
zhangchen@createlink.com

Jianheng Tang
HKUST(GZ)
jtangbf@connect.ust.hk

Jia Li*
HKUST(GZ)
jialee@ust.hk

Lei Chen
HKUST(GZ)
leichen@ust.hk

ABSTRACT

Most enterprise graph data derives from relational databases, yet transforming relational tables into query-optimized graph schemas remains challenging. Existing approaches have notable limitations: (1) transformations based on primary and foreign keys often fail to generate schemas optimized for query performance; (2) manual schema design, although flexible, is costly and requires domain expertise; and (3) machine learning methods predict graph structures based on data patterns but heavily depend on large, high-quality training datasets. To address these challenges, we propose GalaxyWeaver, a framework to automate query-aware graph schema generation. GalaxyWeaver utilizes the reasoning power of Large Language Models (LLMs) to align graph schema designs with specific query requirements, effectively integrating domain knowledge with optimization strategies. The framework employs prompt-guided analysis to enhance the decision-making accuracy of LLM agents, facilitating iterative schema refinement. Experiments across diverse domains show that GalaxyWeaver simplifies transformation while improving query performance and reducing storage costs.

PVLDB Reference Format:
Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jia Li, and Lei Chen. GalaxyWeaver: Autonomous Table-to-Graph Conversion and Schema Optimization with Large Language Models. PVLDB, 18(12): 5100 - 5112, 2025.
doi:10.14778/3750601.3750630

1 INTRODUCTION

Graph databases [1, 15, 28, 34, 43] have emerged as a transformative paradigm for managing and analyzing complex network data. By representing information as vertices, edges, and properties, they enable efficient traversal and querying of multi-hop relationships, making them invaluable in domains such as social networking [8, 12, 17, 24], energy transmission [44], fraud detection [21, 25, 33], and knowledge graphs [7, 27, 32, 42].

However, this raises a fundamental question: where do graph data come from? Based on deployment experiences from Galaxybase [34], a distributed graph database, we found that nearly 80% of graph data utilized by enterprise users derives from relational databases. However, the process of converting relational tables into optimized graph structures remains a significant challenge. Many organizations lack the necessary tools, expertise, and methodologies for effective transformations, and most rely on external support to design graph structures that meet their analytical needs. This disconnect between relational data and graph representation often leads to suboptimal transformations, resulting in inflated storage costs, increased query latency, and diminished performance in graph-specific analytics. Consequently, the full potential of graph databases—particularly for multi-hop queries and complex relationship modeling—remains underutilized, hindering their widespread adoption across industries.

Existing approaches for table-to-graph conversion generally fall into three categories. The first relies on direct mapping of primary keys to vertices and foreign keys to edges [18, 35, 37], preserving the structure of the relational schema but often resulting in suboptimal graph structures for analytical tasks. The second approach involves manual design, where users define mappings interactively to suit specific requirements [6, 13]. While this allows for tailored solutions, it is labor-intensive and error-prone, especially for users with limited graph modeling expertise. The third approach leverages machine learning to predict graph structures based on data patterns, automating parts of the process [29, 38]. However, this method depends heavily on high-quality training data and struggles to scale across large, heterogeneous datasets.

The rapid advancement of state-of-the-art large language models (LLMs), such as ChatGPT [22], Claude [5], and DeepSeek[19], has accelerated progress toward general artificial intelligence. In the context of graph databases, LLMs hold promise for simplifying the transformation of relational data into optimized graph structures by interpreting and processing natural language instructions. This capability lowers technical barriers, enabling users to generate or refine graph schemas through descriptive prompts with minimal expertise. Despite their potential, current LLMs face significant limitations in addressing the complexities of graph structure optimization. First, they depend heavily on explicit and well-defined instructions; ambiguous or incomplete prompts often result in suboptimal or poorly structured graph schemas. Second, they struggle with multi-objective optimization tasks, which require balancing competing goals such as minimizing graph size, enhancing query performance, and maintaining schema compatibility.

To bridge this gap, we present GalaxyWeaver, an LLM-powered framework designed to automate the transformation of relational

* Corresponding Authors.

Proceedings of the VLDB Endowment, Vol. 18, No. 12 ISSN 2150-8097.
doi:10.14778/3750601.3750630

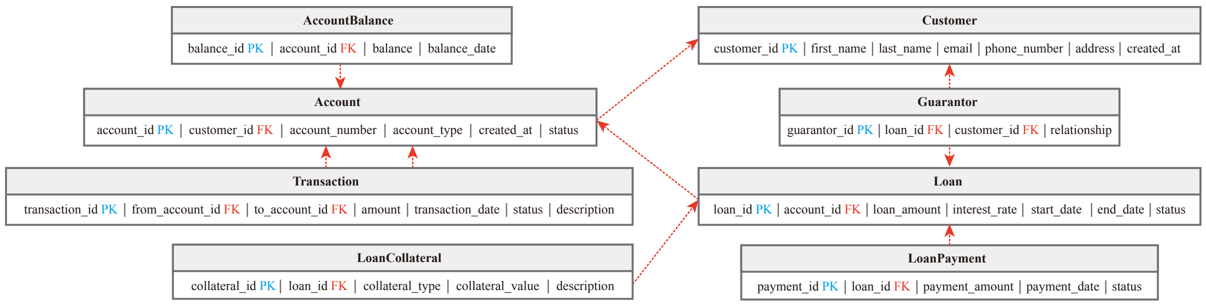

AccountBalance
balance_id PK | account_id FK | balance | balance_date
Account
account_id PK | customer_id FK | account_number | account_type | created_at | status
Transaction
transaction_id PK | from_account_id FK | to_account_id FK | amount | transaction_date | status | description
LoanCollateral
collateral_id PK | loan_id FK | collateral_type | collateral_value | description
Customer
customer_id PK | first_name | last_name | email | phone_number | address | created_at
Guarantor
guarantor_id PK | loan_id FK | customer_id FK | relationship
Loan
loan_id PK | account_id FK | loan_amount | interest_rate | start_date | end_date | status
LoanPayment
payment_id PK | loan_id FK | payment_amount | payment_date | status

Figure 1: Relational schema for financial data

data into optimized graph structures. Based on practical experience from commercial Galaxybase deployments [34], where each graph targets a specific business scenario with relatively stable query patterns. Leveraging the reasoning capabilities of large language models, GalaxyWeaver aligns graph designs with query intent and integrates domain knowledge with optimization strategies. This approach reduces operational complexity and empowers users to harness the analytical potential of graph databases without requiring advanced technical expertise. This work showcases how our framework simplifies data transformation and drives the evolution of graph-based solutions across industries. The main contributions of this work are as follows:

- **GalaxyWeaver Framework**: We propose **GalaxyWeaver**, an LLM-powered framework that automates the entire process of transforming relational data into optimized graph structures. By incorporating query-awareness and domain knowledge, GalaxyWeaver iteratively refines graph schemas to align with analytical requirements, reducing manual effort and enabling more efficient graph analysis.
- **Systematic Schema Optimization Methodology**: We revisit table-to-graph conversion techniques, and systematically summarize a set of graph schema optimization strategies based on extensive real-world deployments of **Galaxybase**, providing practical and reusable guidelines for graph modeling.
- **Prompt-Guided Optimization Workflow**: GalaxyWeaver unifies table-to-graph conversion and schema optimization into a structured stepwise workflow. The *Advisor* employs prompt-guided analysis to enhance the decision-making accuracy of LLM agents, while the *Executor* executes these decisions, enabling adaptive schema evolution and ensuring schema integrity.
- **Extensive Empirical Evaluation**: We conduct extensive experiments on datasets from diverse domains, including marketing, e-commerce, social networks, and academic knowledge graphs. Results show that GalaxyWeaver not only automates table-to-graph transformation but also significantly improves query performance and reduces storage overhead across all domains.

2 PRELIMINARIES

2.1 Relational Database and Schema

A relational database [9, 14, 20] organizes data into a collection of relational tables, each consisting of tuples (rows) structured according to a predefined schema, as illustrated in Figure 1. The schema specifies the attributes (columns), their data types, and associated constraints, serving as a blueprint for data organization and integrity. Formally, a relational table $\mathcal{R}$ comprises the following components:

- $table\_name$: The unique identifier of a relational table in the database.
- $primary\_key$ (PK): A minimal set of attributes that uniquely identify each tuple in a relational table.
- $foreign\_key$ (FK): Attributes in one relational table that reference the primary key of another table, establishing inter-table relationships.
- a: Attributes, properties, or characteristics that describe each tuple in a relational table.
- c: Constraints such as uniqueness, referential integrity, and domain restrictions, which ensure data validity and maintain consistency across the database.

$$\mathcal{R} = \{table\_name, primary\_key, foreign\_key, a, c\}.$$

The schema of a relational database, comprising multiple relations, is formally represented as:

$$\mathcal{S} = \{\mathcal{R}_1, \mathcal{R}_2, \ldots, \mathcal{R}_m\},$$

where each relational table $\mathcal{R}_i$ is defined by its attributes, primary keys, foreign keys, and constraints. Relationships between relational tables are expressed through foreign keys. For instance, a foreign key $FK(\mathcal{R}_i.a_x) \rightarrow PK(\mathcal{R}_j)$ signifies that attribute a_x in $\mathcal{R}_i$ references the primary key of $\mathcal{R}_j$.

2.2 Graph Database and Schema

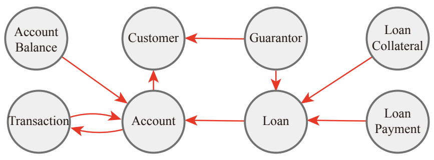


Figure 2: Graph schema for financial data

A graph database [1, 15, 28, 34, 43] is a system for storing, retrieving, and analyzing data represented as a graph. Unlike relational databases that use tables, graph databases model entities as vertices and relationships as edges, enabling efficient management of complex, interconnected data. Formally, a graph database $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consists of a set of vertices $\mathcal{V}$ and a set of edges $\mathcal{E}$. The property graph model [2–4], widely adopted in graph databases, extends this basic structure by associating properties with both vertices and edges.

To ensure consistency and robustness, we adopt a schema design with constraints, as illustrated in Figure 2. Both vertices and edges are represented as relations within a schema. The components of graph schema are defined as follows:

- v: Vertices, representing the nodes in the graph.
- e: Edges, representing the connections between vertices.
- $type$: The type or category of the vertex or edge.
- $primary\_key$: A unique identifier for each vertex.
- $property$: Attributes or properties associated with the vertex or edge.

$$\mathcal{V} = \{v \mid v = (type, primary\_key, property)\},$$

$$\mathcal{E} = \{e \mid e = (source\_v, target\_v, type, property)\}.$$

Each vertex is assigned a unique identifier ($primary\_key$), and each edge connects precisely one source vertex to one target vertex. Multiple edges may exist between any two vertices, even when these edges share the same type, and self-loops are permitted.

2.3 Large Language Models

Large Language Models (LLMs) are advanced artificial intelligence systems trained on extensive textual corpora to perform language understanding and generation tasks with human-like proficiency. These models, such as ChatGPT [41], Claude [5], and DeepSeek[19], are typically based on deep neural network architectures that leverage the attention mechanism [36]. This mechanism enables LLMs to focus on relevant parts of the input sequence while generating outputs, effectively capturing contextual relationships.

Prompt engineering is a key method for optimizing the performance of LLMs. It focuses on crafting effective prompts to guide the model toward desired outputs. Techniques such as few-shot prompting [39] and the chain of thought approach enable models to handle diverse tasks and complex reasoning. By shaping the input structure, prompt engineering maximizes task accuracy without modifying the underlying model.

LLM agents combine large language models with auxiliary modules—such as planning, memory, and external tools—to handle complex, multi-step tasks [45, 47]. The LLM acts as the core reasoning engine, coordinating actions and adapting to specific task needs. This architecture extends LLMs to more interactive and problem-oriented domains.

3 CHALLENGES AND MOTIVATIONS

To ensure data integrity while meeting query requirements, an effective table-to-graph conversion must balance two key objectives: optimizing query performance and minimizing storage overhead. Commonly used approaches include: (1) primary-foreign key-based conversion, (2) manual design, and (3) machine learning techniques that predict graph structures from data patterns, partially automating the process.

Manual methods require deep expertise in graph modeling, making them impractical for non-specialists. Existing machine learning approaches automate schema transformation but depend on large, high-quality datasets and primarily optimize for predictive tasks like node classification, often failing to refine schemas for efficient query performance.

Given these limitations, we focus on primary-foreign key-based conversions. Section 3.1 revisits and refines traditional methods grounded in this approach, while Section 3.2 delves into graph schema optimization, addressing inherent challenges and presenting practical solutions informed by insights from deploying Galaxybase.

3.1 Revisiting Standard Table-to-Graph Conversion

Traditional methods for converting relational data to graphs work well in simple cases with single primary key (PK) and foreign key (FK) relationships [18, 35, 37]. While these approaches maintain data integrity by mapping relational keys to graph structures, they struggle with the complexity of real-world schemas, which frequently involve multiple PKs, multiple FKs, and intricate interdependencies beyond the scope of standard conversion techniques. To address these limitations, we propose a standardized framework that extends traditional methods with a systematic, adaptive rule set capable of handling diverse PK-FK configurations. The conversion rules are defined as follows:

Tables with Primary Keys but No Foreign Keys: For a relational table $\mathcal{R}$ containing one or more PKs but no FKs, the PK attributes are merged to create a unique vertex v. All non-key attributes (a) of $\mathcal{R}$ are stored as properties of the vertex. This design ensures that each tuple in $\mathcal{R}$ is represented as a distinct vertex, preserving all associated attributes. Formally, the conversion of $\mathcal{R}$ into vertices $\mathcal{V}$ is expressed as:

$$v(\mathcal{R}) = (\mathcal{R}.\text{table\_name}, \text{merge}(\mathcal{R}.PK_i \mid i \in [1, n]), \{\mathcal{R}.a_i \mid i \in [1, k]\}),$$

where table_name is the name of the relational table, PK_i are the primary key attributes, and a_i are the non-key attributes. The resulting set of vertices is:

$$\mathcal{V}(\mathcal{R}) = \{v(\mathcal{R}) \mid \text{each tuple in } \mathcal{R}\}.$$

Tables with Primary Keys and Foreign Keys: For a relational table $\mathcal{R}$ containing both PKs and FKs, the primary keys are merged to form a unique vertex $v(\mathcal{R})$, with its non-key attributes (a) stored as properties. Additionally, edges $\mathcal{E}$ are created to link $v(\mathcal{R})$ to vertices of referenced tables $\mathcal{R}_{\text{ref}}$ via their FKs. This ensures that both PK-based queryability and FK-based connectivity are preserved in the graph representation. The vertex for $\mathcal{R}$ is defined as before, and the edges are represented as follows:

$$e(\mathcal{R}, \mathcal{R}_{\text{ref}}) = (v(\mathcal{R}), v(\mathcal{R}_{\text{ref}}), \text{merge}(\mathcal{R}.\text{table\_name}, \mathcal{R}_{\text{ref}}.\text{table\_name}), \varnothing),$$

where:

- $\mathcal{R}$: The table containing the FK that establishes the relationship.
- $\mathcal{R}_{\text{ref}}$: The table referenced by $\mathcal{R}$, where the corresponding PK resides.
- merge($\mathcal{R}$.table_name, $\mathcal{R}_{\text{ref}}$.table_name): A label indicating the relationship between $\mathcal{R}$ and $\mathcal{R}_{\text{ref}}$.
- $\varnothing$: Represents an edge without additional properties.

By assigning attributes (a) to vertices instead of edges, this design maintains a clear distinction between data and relationships: vertices store all data properties, while edges solely represent connections. This approach minimizes redundancy by ensuring each attribute is stored exactly once within the vertex it pertains to, streamlining data organization and integrity. The resulting set of edges is:

$$\mathcal{E}(\mathcal{R}, \mathcal{R}_{\text{ref}}) = \{e(\mathcal{R}, \mathcal{R}_{\text{ref}}) \mid \text{each foreign key in } \mathcal{R} \text{ referencing } \mathcal{R}_{\text{ref}}\}.$$

Tables without Primary Keys but with Foreign Keys: For a relational table $\mathcal{R}$ that lacks a PK but contains FKs, an auto-generated identifier (auto_id) is assigned to each tuple to uniquely identify the corresponding vertex $v(\mathcal{R})$. Non-key attributes (a) are stored as properties of the vertex, consistent with the design for tables that include PKs. Edges are then created between this vertex and the vertices of referenced tables using FKs, following the same methodology as described for tables with FKs. The vertex definition, with auto_id generated for $\mathcal{R}$, is:

$$v(\mathcal{R}) = (\mathcal{R}.\text{table\_name}, \text{auto\_id}, \{\mathcal{R}.a_i \mid i \in [1, k]\}),$$

where table_name represents the name of the relational table, auto_id is an auto-generated identifier, and a_i denotes the non-key attributes of $\mathcal{R}$.

Since $\mathcal{R}$ lacks an original PK, it will not be referenced by foreign keys from other tables. Consequently, edges involving $\mathcal{R}$ are exclusively outgoing, representing its relationships with the referenced tables.

Tables without Primary Keys or Foreign Keys: For a relational table $\mathcal{R}$ that has neither a PK nor an FK, a auto_id is generated for each tuple, creating an isolated vertex $v(\mathcal{R})$, consistent with the approach used for generating vertices with auto_id in relations without PKs. All attributes (a) in the relational table are stored as properties of the vertex. These vertices remain unconnected in the graph, mirroring the standalone nature of such tables in the relational schema.

3.2 Further Optimization on Graph Schema

While standard table-to-graph methods provide general solutions, they often fail to meet the query performance needs of specialized industry scenarios. Focusing on financial services, this section shows how context-aware schema restructuring—leveraging the flexibility of graph models—can improve efficiency for key tasks such as transaction analysis and guarantor retrieval.

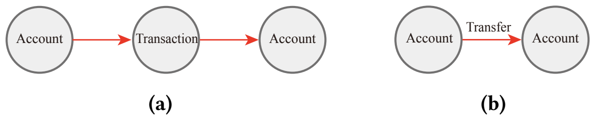


Figure 3: Optimization of transaction-based queries

In particular, we point out four issues on the standard table-to-graph conversion process:

(1) Suboptimal Traversal Complexity: One of the most prominent challenges in the initial graph schema is the inefficiency introduced by multi-hop traversals, particularly for queries involving indirect relationships. These inefficiencies increase query execution time and computational overhead, making the schema less suitable for high-performance applications.

Example 3.1. In the context of *Transaction-Based Queries*, schema design plays a pivotal role in determining query efficiency. Consider a query that retrieves a transaction based on its unique identifier. In this case, preserving the `Transaction` vertex, as shown in Figure 3a, allows for a direct lookup using the primary key (`transactionId`), ensuring efficient query execution. However, for queries that analyze transaction flows between accounts, this schema design introduces unnecessary intermediate steps. By replacing the intermediate `Transaction` vertex with a direct `Transfer` edge between `Account` vertices, as illustrated in Figure 3b, the traversal complexity is significantly reduced. This optimization effectively halves the traversal steps required to analyze transaction chains.

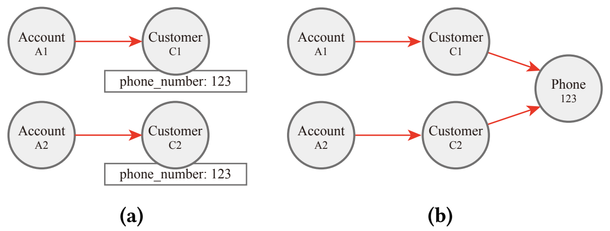


Figure 4: Optimization of phone number-based queries

(2) Inefficient Property Utilization: Another limitation lies in the suboptimal use of vertex and edge properties, often resulting in inefficient filtering and degraded query performance—especially when properties must be accessed or filtered at scale.

Example 3.2. Phone Number-Based Queries serve as a clear example of this limitation. When retrieving a customer's phone number associated with a specific account, the initial schema, as shown in Figure 4a, is sufficient. The query involves traversing from the `Account` vertex to the `Customer` vertex and accessing the phone number as a vertex property. However, for reverse queries—such as identifying all accounts linked to a specific phone number—the schema requires scanning all `Customer` vertices and filtering based on the `phone_number` property, leading to inefficiencies. To address this issue, representing `Phone` as a standalone vertex directly connected to `Customer` vertices, as illustrated in Figure 4b, eliminates the need for extensive property-based filtering. Additionally, this transformation introduces explicit relationships that support richer graph analytics.

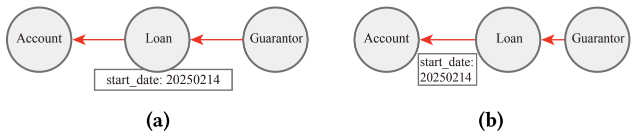


Figure 5: Enhancing query awareness in loan-based queries

(3) Lack of Query Awareness: Initial graph schemas are often designed without sufficient consideration of specific query contexts, such as temporal constraints or relational dependencies. This lack of adaptability can result in suboptimal performance for targeted queries that require more sophisticated filtering or traversal strategies.

Example 3.3. Loan-Based Queries highlight opportunities for schema optimization in query-aware contexts. For instance, retrieving guarantors for loans associated with a specific account involves traversing from the `Account` vertex to the `Loan` vertex and then to the Guarantor vertices, as shown in Figure 5a. While this design is adequate for basic queries, it becomes inefficient when temporal constraints are introduced. For example, filtering loans based on a specific issuance date requires scanning all Loan vertices and applying a filter on the temporal property. An optimized schema, as illustrated in Figure 5b, stores temporal information as an edge property between Account and Loan vertices. This design enables direct filtering on edges during traversal, reducing computational overhead.

(4) Data Redundancy: Data redundancy, such as unused vertices, edges, or properties, not only complicates the graph structure but also increases storage costs. This redundancy can negatively affect both query performance and schema maintainability. For example, in *Transaction-Based Queries*, retaining both the Transaction vertex and direct Account-to-Account edges within the same schema leads to duplicated information. Similarly, in *Phone Number-Based Queries*, storing phone numbers as both vertex properties and separate vertices introduces unnecessary redundancy.

These issues highlight a fundamental limitation in the standard table-to-graph conversion process: it often overlooks the specific demands of target queries and application scenarios, leading to inefficiencies such as redundant storage, traversal complexity, and suboptimal query performance. To address these challenges, we propose a targeted optimization framework that aligns the graph schema with query requirements. This process involves analyzing query patterns to identify bottlenecks and refining the schema to enhance efficiency and maintainability. Key strategies include:

- **Convert Vertices to Edges:** Transform less significant vertices into edges to simplify relationships and reduce traversal steps, particularly for queries focused on pathfinding or connectivity.
- **Convert Vertex Properties to Edge Properties:** Shift properties that are more relevant to relationships, such as weights or timestamps, to edges to enable faster filtering.
- **Extract Vertex Properties into Separate Vertices:** Promote frequently queried properties into dedicated vertices to improve direct accessibility and support property-specific queries.
- **Remove Redundant Vertices, Edges, and Properties:** Eliminate unnecessary elements to streamline the graph structure, reducing storage overhead.
- **Adjust Edge Directions:** Orient edges to align with typical query traversal patterns.
- **Rename Vertex and Edge Types:** Use clear, domain-relevant naming conventions to enhance schema readability and facilitate intuitive query construction.

4 SYSTEM DESIGN AND IMPLEMENTATION

4.1 Overview

We present **GalaxyWeaver**, a system designed to transform relational schemas into graph schemas optimized for query performance by leveraging strategies informed by domain expertise and large language models (LLMs). As shown in Figure 6, the system architecture consists of two primary modules: the *Table-to-Graph Module* and the *Graph Schema Optimization Module*. Together, these modules enable the seamless transformation and iterative refinement of graph schemas tailored to specific application needs. The system requires two inputs: (1) query specifications, defining the queries that the graph schema must support efficiently, and (2) a relational database schema, providing the structure of the source data. The output is an optimized graph schema that aligns with the input queries, ensuring efficiency and relevance.

The *Table-to-Graph Module* handles the initial transformation of relational data into a graph structure using the *Converter* component. This step follows the strategies detailed in Section 3.1, where primary keys are converted into vertices, foreign keys into edges, and other attributes into properties. The module supports direct integration with relational databases such as MySQL and Oracle, extracting primary and foreign key relationships from schema metadata. For cases where data is not available in traditional databases, the system supports the direct input of schema information through table schemas or SQL creation statements. Additionally, an intuitive interface allows users to define relationships manually, ensuring compatibility with diverse data sources.

The *Graph Schema Optimization Module* iteratively refines the initial graph schema using strategies outlined in Section 3.2, supported by LLMs that simulate human reasoning. The optimization process prioritizes query performance, storage efficiency, and structural simplicity. The workflow is composed of four key components. First, the *Viewer* component visualizes the graph schema, providing users with insights into the current structure. Second, the *Advisor* component prompts the LLM with a combination of the current schema, input queries, and strategy-specific prompts to generate optimization decisions. Third, the *Executor* applies these decisions using rule-based functions to modify the schema. Finally, the *Adjuster* provides an interactive interface for users to review and refine the schema manually if needed.

4.2 Table-to-Graph Module

The transformation from relational data to graph data is critical for ensuring data accuracy and completeness. To meet the diverse needs of users and maintain the integrity of data relationships, we design a system: the *Table-to-Graph Module*. This module supports three distinct transformation modes: (1) automatic conversion by connecting to relational databases, (2) automatic schema recognition from SQL database creation scripts, and (3) interactive modeling for manually defining relationships when source data is unavailable.

Automatic Conversion by Connecting to Relational Databases. For well-structured relational databases, primary and foreign keys are the foundation of data relationships. The *Table-to-Graph Module* leverages these existing relationships to automatically construct graph schemas. By extracting schema metadata directly from databases such as MySQL or Oracle, the module can seamlessly map primary keys to vertices, foreign keys to edges, and other attributes to properties. This approach minimizes manual intervention while ensuring the completeness and integrity of the graph structure.

Schema Recognition from SQL Creation Scripts. When schema information is provided in the form of SQL creation statements, the module parses these scripts to identify primary and foreign key elements. This enables automated graph construction even without a direct connection to a live database. By interpreting schema definitions, the module generates an initial graph schema that accurately reflects the structure and relationships defined in the original data.

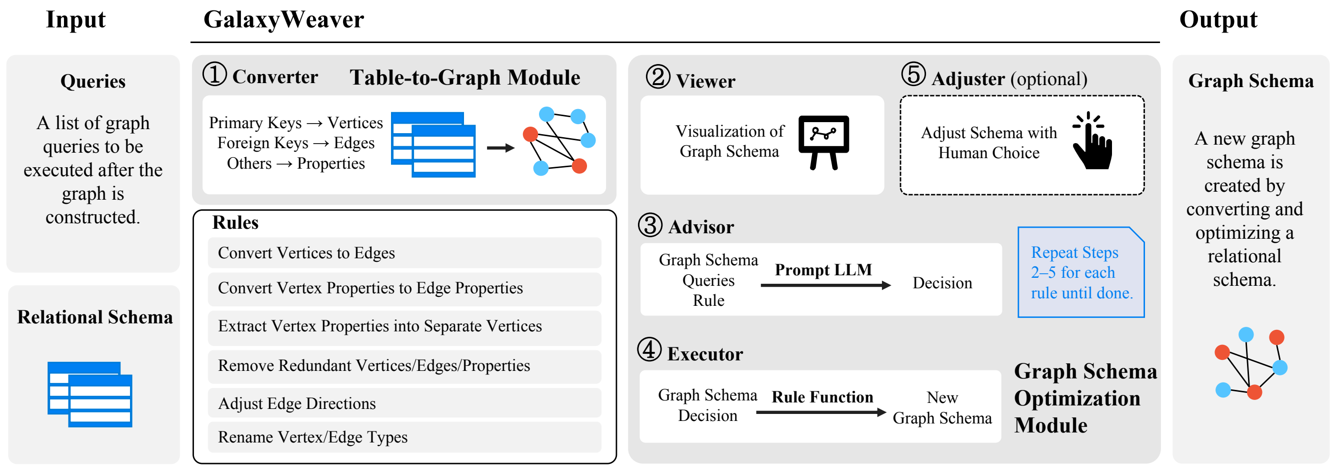

Input
GalaxyWeaver
Output
Queries
A list of graph queries to be executed after the graph is constructed.
Relational Schema
① Converter
Table-to-Graph Module
Primary Keys → Vertices
Foreign Keys → Edges
Others → Properties
Rules
Convert Vertices to Edges
Convert Vertex Properties to Edge Properties
Extract Vertex Properties into Separate Vertices
Remove Redundant Vertices/Edges/Properties
Adjust Edge Directions
Rename Vertex/Edge Types
② Viewer
Visualization of Graph Schema
⑤ Adjuster (optional)
Adjust Schema with Human Choice
③ Advisor
Graph Schema
Queries
Rule
Prompt LLM
Decision
Repeat Steps 2–5 for each rule until done.
④ Executor
Graph Schema
Decision
Rule Function
New Graph Schema
Graph Schema Optimization Module
Graph Schema
A new graph schema is created by converting and optimizing a relational schema.

Figure 6: The architecture of GalaxyWeaver

Interactive Modeling for Custom Relationship Input. In scenarios where data lacks a well-defined schema or exists in semi-structured formats (e.g., CSV or JSON files), the module offers an interactive interface that allows users to manually define relationships and construct the desired graph schema. Although this process requires user input, users typically have deep domain knowledge of their data and its inherent relationships, even if they are not experts in graph schema design. For example, a user managing an e-commerce dataset inherently understands that "User ID" links to "Order ID" . This domain knowledge enables them to accurately define key relationships without requiring advanced technical expertise.

4.3 Graph Schema Optimization Module

Efficient design and optimization of graph schemas are critical for achieving high performance in graph database applications. However, for users without prior experience in graph databases, tailoring a schema to specific use-case requirements can be a challenging and complex process, as discussed in Section 3.2. To address this, we introduce the *Graph Schema Optimization Module*, an intelligent module provided by GalaxyWeaver. This module is designed to guide users through the schema optimization journey, combining intuitive visualization tools with intelligent refinement techniques.

Integrating LLMs into the Optimization Process. Large language models (LLMs) offer promising capabilities for automating schema optimization by leveraging their natural language understanding and reasoning abilities. However, applying LLMs directly to this task reveals several challenges: (1) Inconsistent Optimization Results: LLM-generated plans often vary widely and may fail to balance multiple optimization objectives. (2) Schema Incompleteness: In complex scenarios, LLMs may overlook critical schema elements, leading to incomplete or suboptimal designs.

These limitations underscore the need for a more structured approach to harness LLM capabilities effectively. The *Graph Schema Optimization Module* addresses this by adopting a human-inspired optimization framework. By deconstructing the optimization process into smaller, sequential tasks, it allows LLMs to focus on specific, well-defined decisions. This targeted approach not only improves the precision of LLM-generated recommendations but also integrates execution and validation mechanisms to ensure the resulting schema is both complete and efficient.

At the heart of this module are four tightly integrated components, each serving a distinct yet complementary function:

- *Viewer*: Provides an intuitive visualization of the graph schema, giving users a clear and immediate understanding of its current structure.
- *Advisor*: Analyzes both the schema and user query requirements to generate tailored optimization recommendations.
- *Executor*: Applies the recommended changes to the graph schema, ensuring that schema integrity and correctness are maintained throughout the process.
- *Adjuster*: Allows users to manually refine the schema, enabling adjustments to accommodate specific use-case requirements or domain-specific knowledge.

Among these, the *Advisor* serves as the cornerstone of the module. It is tasked with generating actionable recommendations by analyzing both user requirements and the graph's structural characteristics, ensuring that optimization decisions are both relevant and impactful.

Enhancing Optimization through Prompt-Guided Analysis. A distinctive feature of the *Advisor* is its innovative use of prompt-guided analysis to refine decision-making accuracy. Unlike traditional approaches that rely on direct schema optimization by LLMs, this module strategically leverages prompts to evaluate specific schema aspects, enabling more informed and targeted transformations. Below, we demonstrate this process with key examples of graph schema optimization.

4.3.1 Converting Vertices to Edges.

One optimization strategy targets scenarios where intermediate vertices exist solely to represent 1-to-1 relationships — vertices with exactly two outgoing edges and no incoming edges. In these cases, collapsing such vertices into direct edges between the connected entities reduces traversal complexity and improves query efficiency by eliminating unnecessary hops.

For example, consider the vertex type `Guarantor`, which connects `Loan` and `Customer` in a 1-to-1 relationship (`Loan` ← `Guarantor`

$\rightarrow$ `Customer`). The decision to retain `Guarantor` as vertices or transform them into edges depends on their usage patterns:

- **Retain as a vertex:** If queries frequently retrieve `Guarantor` using their primary key, keeping them as vertices ensures efficient direct lookups.
- **Transform into an edge:** If `Guarantor` primarily serve as an intermediary in traversal operations, converting them into edges (e.g., `Loan` $\rightarrow$ `Customer` with edge properties) eliminates an unnecessary vertex lookup and improves query performance.

To guide this decision-making, the *Advisor* constructs a structured prompt for the LLM, incorporating the graph schema and queries at the beginning. This provides contextual information before presenting the decision-making problem to the model. We outline the key decision-making prompt below:

> *I would like to understand how my vertex type [Guarantor] is being used according to my query requirements. Which of the following scenarios do you think it fits? You may write the query in Cypher to distinguish whether it is being located or traversed.*
> *(1) The requirement is to locate this vertex using the primary key.*
> *(2) The requirement is to traverse through neighbors to reach this vertex.*
> *(3) Both (1) and (2).*
> *(4) Neither (1) nor (2).*
> *For the final result, you should output the following format: 'I will choose 1/2/3/4. '*

If the response indicates scenario (2), where `Guarantor` are accessed solely through traversal, the *Advisor* recommends converting them into edges. The *Executor* then applies this transformation, preserving schema integrity while enhancing traversal efficiency. The resulting output may look as follows:

> *I will choose 2.*

4.3.2 Converting Vertex Properties to Edge Properties. Another optimization approach addresses cases where vertex properties are exclusively used during traversal. In such scenarios, transferring these properties to edges can reduce query complexity and improve performance, especially when dealing with isolated vertices with an out-degree of 1 and an in-degree of 0.

For example, consider the vertex type `LoanPayment`, which includes properties such as `payment_amount`, `payment_date`, and `status`. These properties are associated with edges connecting `Loans` to `Customers`. The decision to retain or transfer these properties depends on their usage patterns:

- **Retain as vertex properties:** If queries frequently locate `LoanPayment` by its primary key, keeping the properties on the vertex ensures efficient lookups.
- **Transfer to edge properties:** If the properties are accessed exclusively during traversal, transferring them to the edges eliminates unnecessary vertex lookups, enhancing query performance.

To evaluate these patterns, the *Advisor* uses the following prompt:

> *I would like to understand how my vertex type [LoanPayment] and its properties [payment_amount, payment_date, status] are being used according to my query requirements. Which of the following scenarios do you think it fits? You may write the query in Cypher to distinguish whether it is being located or traversed.*
> *(1) The requirement is to locate this vertex using the primary key.*
> *(2) The requirement is to traverse through neighbors to reach this vertex.*
> *(3) Both (1) and (2).*
> *(4) Neither (1) nor (2).*
> *For the final result, you should output the following format: 'I will choose 1/2/3/4. The properties [xxx, xxx] are retrieved by locating the vertex, and the properties [xxx, xxx] are retrieved by traversing through neighbors.'*
> *Note: You cannot alter the final result format, and the lists [xxx, xxx] may be empty.*

If the response indicates scenario (3), the *Advisor* recommends transferring the relevant properties to edges, as illustrated below:

> *I will choose 3. The properties [payment_amount, status] are retrieved by locating the vertex, and the properties [payment_amount, payment_date] are retrieved by traversing through neighbors.*

Since `payment_date` is only accessed through traversal, the *Executor* moves this property from the vertex to the corresponding edges, ensuring a more efficient query structure.

4.3.3 Extracting Vertex Properties into Separate Vertices. Another optimization strategy focuses on cases where specific vertex properties are frequently used for traversal or filtering rather than for direct lookups. In such scenarios, extracting these properties into separate vertices can improve indexing efficiency and reduce computational overhead associated with property scans.

For instance, consider the vertex type `User`, which contains properties such as `name` and `phone`. If queries often involve filtering or traversing `User` based on these properties, restructuring the schema by promoting them to independent vertices can enhance performance. The decision to extract these properties depends on their usage patterns:

- **Retain as vertex properties:** If queries primarily retrieve `name` and `phone` by directly looking up the `User` vertex, keeping them as vertex properties ensures efficient access.
- **Extract into separate vertices:** If these properties are frequently used for filtering or traversal, converting them into independent vertices and linking them to `User` can optimize query performance by leveraging edge relationships.

To evaluate whether a property should be extracted, the *Advisor* generates the following prompt:

Please write the Cypher query for my requirements and answer the following questions: In each requirement, is there a query that requires finding a vertex by iterating over it based on a specific property (not the primary key)? If so, please identify this property and determine whether it can serve as a vertex.
For the final result, you should output the following format: 'The result is [VertexTypeName-PropertyName, VertexTypeName-PropertyName...]'.
For example, 'The result is [User-name, User-phone]'.
Note: The lists [] may be empty.

If the LLM identifies properties such as `name` and `phone` as traversal-critical, they are extracted into separate vertices. The response might appear as follows:

The result is [User-name, User-phone].

Based on this analysis, the *Advisor* recommends restructuring the schema. The *Executor* then applies the transformation by creating new vertex types for these properties and establishing edges between them and the original `User` vertex.

4.4 Running Example

To demonstrate how GalaxyWeaver transforms relational data into optimized graph structures, this section presents a step-by-step walkthrough using the finance dataset shown in Figure 1. We focus on the core decision-making process within the *Graph Schema Optimization Module*, where the *Advisor* analyzes query workloads to propose optimization decisions, and the *Executor* applies the corresponding schema transformations.

We select four representative analytical queries based on this dataset:

- Query 1: Retrieve the total loan amount issued to each customer, along with the total loan amount they have guaranteed.
- Query 2: Compute the proportion of repayment amounts made by each guarantor relative to the total loans they have guaranteed.
- Query 3: Determine the number of customers associated with a given phone number.
- Query 4: Identify transaction relationships between two specified accounts.

Table 1 summarizes the core steps of the optimization process, with the final optimized schema shown in Figure 7.

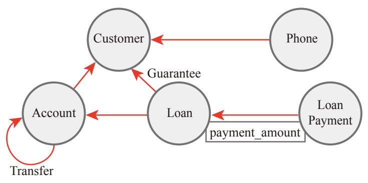


Figure 7: Graph schema after optimization

Table 1: Schema optimization process in GalaxyWeaver

Step	Target Data	Decision Details
Convert Vertices to Edges		
1	Account ← Transaction → Account	Account → Account
2	Loan ← Guarantor → Customer	Loan → Customer
Convert Vertex Properties to Edge Properties		
3	AccountBalance → Account	No properties moved
4	LoanPayment → Loan	Move [payment_amount]
5	LoanCollateral → Loan	No properties moved
Extract Vertex Properties into Separate Vertices		
6	Whole Vertex Types	Extract [phone_number]
Remove Redundant Edges		
7	Whole Edge Types	Remove [AccountBalanceToAccount, LoanCollateralToLoan]
Remove Redundant Vertices		
8	Whole Vertex Types	Remove [AccountBalance, LoanCollateral]

5 EVALUATION

In this section, we evaluate GalaxyWeaver's schema generation and optimization capabilities by comparing three query-based approaches: (1) executing queries on Relational Table data; (2) running queries on Basic Graph data, constructed by directly converting primary and foreign key relationships into graph structures; and (3) assessing the performance on Optimized Graph data refined by GalaxyWeaver.

5.1 Experimental Setup

Our experiments utilize Galaxybase [34] (version 3.5.2) for storing and processing both Basic Graph and Optimized Graph representations, while MySQL (version 8.0.41) serves as the relational database for Relational Table representations. All experiments are conducted on a machine equipped with an Intel(R) Xeon(R) Gold 5218R CPU (20 cores, 40 threads), 128 GB of RAM, running Ubuntu 20.04.6 LTS with HDD storage.

Table 2: Statistics of each dataset

Dataset	# Tables	# Columns	# Rows
AVS	3	24	349,967,371
AB	3	15	24,291,489
SE	7	49	5,399,818
MAG	5	13	21,847,396

To evaluate the effectiveness of GalaxyWeaver across diverse data domains, we select four representative datasets from 4DBInfer [38]: Acquire Valued Shoppers (AVS) [16], Amazon Book Reviews (AB) [23], StackExchange (SE) [31], and Microsoft Academic Graph (MAG) [30]. Table 2 provides a summary of their key characteristics. For each dataset, we design three representative queries. The details of these datasets and queries are outlined below. Due to space constraints, only the key attributes are depicted in the schema diagrams.

5.1.1 Acquire Valued Shoppers (AVS). The AVS dataset [16], sourced from the Kaggle Acquire Valued Shoppers Challenge, contains transactional and promotional data from an e-commerce platform. It includes three core tables:

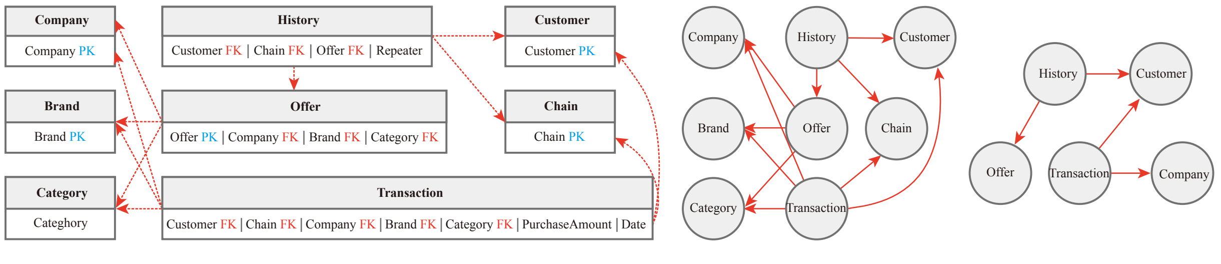

Company
Company PK
History
Customer FK | Chain FK | Offer FK | Repeater
Customer
Customer PK
Brand
Brand PK
Offer
Offer PK | Company FK | Brand FK | Category FK
Chain
Chain PK
Category
Categhory
Transaction
Customer FK | Chain FK | Company FK | Brand FK | Category FK | PurchaseAmount | Date
(a)
(b)
(c)

Figure 8: Schema transformation process for the AVS dataset

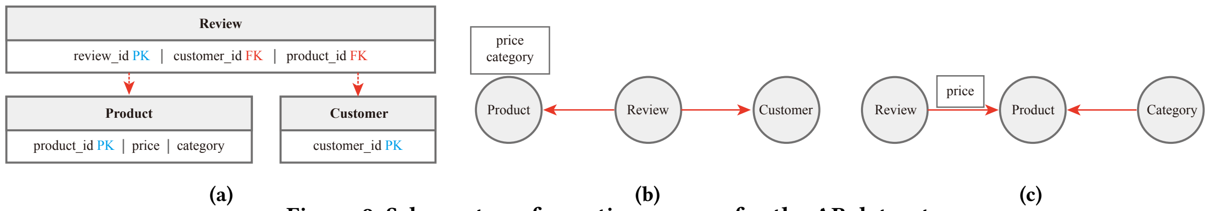

Review
review_id PK | customer_id FK | product_id FK
Product
product_id PK | price | category
Customer
customer_id PK
price category
price
Category
(a)
(b)
(c)

Figure 9: Schema transformation process for the AB dataset

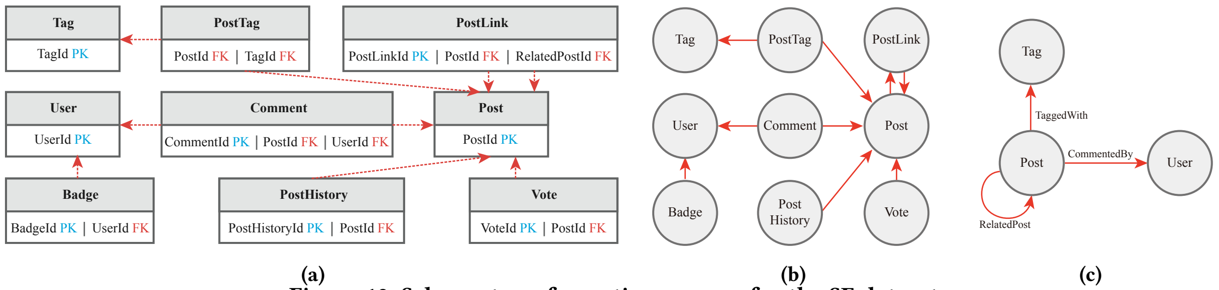


Figure 10: Schema transformation process for the SE dataset

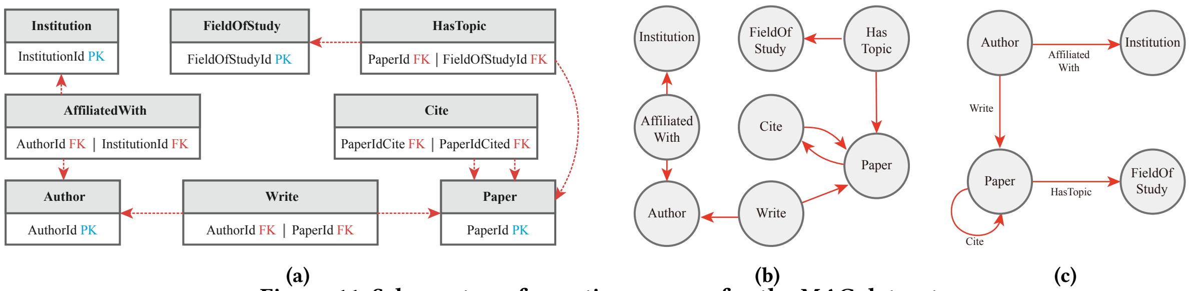


Figure 11: Schema transformation process for the MAG dataset

- `History`: Tracks promotional offers received by customers.
- `Offer`: Details of each promotional offer.
- `Transaction`: Records product purchases by customers.

The relational schema is shown in Figure 8a. To assess the impact of schema transformation and optimization, we design three queries:

- Query 1: Identify the most frequently purchased product categories among customers who have redeemed offers. This query ranks product categories by purchase frequency within the target customer group.
- Query 2: Compute the total purchase amount and transaction count for a specific company within a defined time range. This query aggregates transactions filtered by company and time period.
- Query 3: Analyze repeat purchase behavior among customers post-offer redemption. The query identifies customers who made repeat purchases in the same product category after using an offer.

5.1.2 *Amazon Book Reviews (AB).*

The AB dataset [23] contains product reviews from Amazon, organized into three tables:

- `Customer`: Unique customer identifiers.
- `Product`: Detailed product information, including ID, brand, category, description, price, and title.

- `Review`: Links customers to products and includes review attributes such as rating, review text, submission time, and verification status.

The relational schema is shown in Figure 9a. We evaluate query performance across different representations using the following queries:

- Query 1: Retrieve review details by review ID. This is a simple primary key lookup.
- Query 2: Count the number of reviews for products in a specific category. This query filters products by category and aggregates the reviews.
- Query 3: Identify products with reviews where the price exceeds a specified threshold. This query filters products by price and retrieves the associated reviews.

5.1.3 *StackExchange (SE).*

The SE dataset captures interactions on the StackExchange platform [31], including questions, answers, comments, and user activities. It consists of several interconnected tables:

- `Badge`: Records badges awarded to users.
- `Comment`: Stores comments associated with posts.
- `Post`, `Tag`, `PostLink`, `PostHistory`: Store data related to posts, including content, post relationships, and editing history.
- `User`: Contains user profile information.
- `Vote`: Tracks votes cast on posts.

The relational schema is shown in Figure 10a. To evaluate performance on this relational dataset, we design the following queries:

- Query 1: Find all posts related to a given post within *n* hops, returning post IDs and the shortest path length.
- Query 2: Identify the most active users based on the number of comments posted, returning user IDs, display names, and total comment counts.
- Query 3: Retrieve the most frequently used tags across all posts, returning the tag name and associated post counts.

5.1.4 *Microsoft Academic Graph (MAG).*

The MAG [30] dataset is a comprehensive knowledge graph encompassing academic publications, research topics, authors, and institutions. It consists of several interconnected tables:

- `Paper`, `FieldOfStudy`, `Author`, `Institution`: Represent key academic entities.
- `Cite`, `HasTopic`, `Write`, `AffiliatedWith`: Capture relationships between these entities, such as citation links and author affiliations.

The relational schema is shown in Figure 11a. To evaluate GalaxyWeaver's performance on this academic graph, we design the following queries:

- Query 1: Explore the relationship between two papers (A and B), retrieving all papers forming citation paths between the two within *n* hops.
- Query 2: Identify the top co-author with the most papers in a given institution, returning the co-author with the highest number of papers published with authors from the target institution.
- Query 3: Retrieve the top *n* papers associated with the largest number of research domains, listing each paper with its associated domain count.

5.2 Experimental Methodology

To assess performance impacts across data representations, we generate three versions per dataset: Relational Tables, Basic Graphs, and Optimized Graphs. The Optimized Graphs are produced by GalaxyWeaver, with schemas refined and tailored to match the specific query workload. In this experiment, GalaxyWeaver utilizes GPT-4o as the underlying LLM API.

To evaluate the effectiveness and robustness of GalaxyWeaver, we conduct an LLM-based schema generation experiment across four datasets. For each dataset, GalaxyWeaver is executed 10 times, and we report (1) the average number of prompt and completion tokens, (2) average generation latency, (3) schema correctness, and (4) schema optimization rate, as summarized in Table 3. For the downstream experiments, we select a schema from runs that successfully produced both a correct and fully optimized design.

Table 3: LLM generation statistics and schema quality metrics for GalaxyWeaver (averaged over 10 runs per dataset)

Dataset	Prompt Tokens	Completion Tokens	Generation Latency(s)	Schema Correctness	Schema Opt. Rate
AVS	6,463	2,249	35.7	9/10	74.4%
AB	5,575	2,653	45.8	10/10	84.4%
SE	17,974	5,218	91.1	9/10	97.5%
MAG	8,708	4,198	61.9	9/10	100%

* **Schema Correctness** indicates whether the generated schema supports all predefined graph queries. Failures typically result from major semantic issues such as collapsed vertices or missing vertex types.
* **Schema Opt. Rate** is computed only on correct schemas and reflects how many expert-identified schema refinements (e.g., Convert Vertices to Edges) are successfully applied.

Table 4: Statistics of basic and optimized graphs

Dataset	Basic Graph		Optimized Graph	
	|V|	|E|	|V|	|E|
AVS	350,196,856	1,748,759,227	350,160,197	699,631,692
AB	15,964,090	27,435,504	14,114,827	14,204,911
SE	6,140,680	6,839,823	841,982	1,541,125
MAG	23,050,750	42,222,014	1,939,743	21,111,007

The structural evolution from Relational Tables to Basic Graphs for each dataset is illustrated in Figures 8b, 9b, 10b, and 11b. GalaxyWeaver applies a query-aware transformation process, generating Optimized Graphs that are tailored to the analytical requirements of the workload. The resulting Optimized Graphs are shown in Figures 8c, 9c, 10c, and 11c. To enhance readability, only key edge types and representative properties are retained in these visualizations. Table 4 presents the summary statistics for Basic Graph and Optimized Graph representations across all datasets.

To evaluate query performance across these representations, we apply the queries designed in Section 5.1 to each dataset under all three formats. Specifically, queries are executed using SQL for Relational Tables, and Cypher for both Basic Graphs and Optimized Graphs. For example, the Cypher and SQL implementations for Query 2 on the Amazon Book Reviews dataset are shown below.

SQL query on the Relational Table:

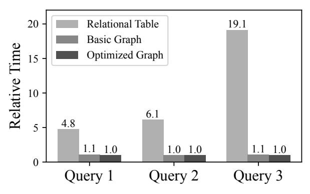
```
SELECT p.category, COUNT(r.review_id) AS review_count
FROM Product p
JOIN Review r ON p.product_id = r.product_id
WHERE p.category = 'Books Sports';
```

Cypher query on the Basic Graph:

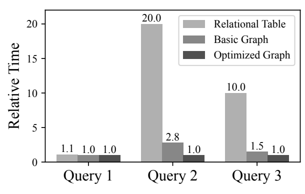
```
MATCH (p:Product)<-[r:ReviewToProduct]-(:Review)
WHERE p.category = 'Books Sports'
RETURN p.category AS product_category, COUNT(r) AS
review_count;
```

Cypher query on the Optimized Graph:

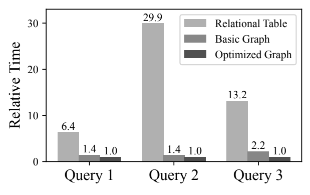
```
MATCH (c:Category category: 'Books Sports')
-[:CategorizedProduct]->(p:Product)
<-[r:ReviewedProduct]-(:Review)
RETURN c.category AS product_category, COUNT(r) AS
review_count;
```



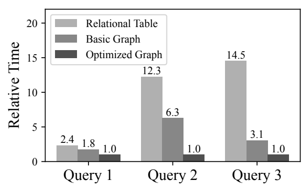
(a) AVS



(b) AB



(c) SE



(d) MAG

Figure 12: Comparative query performance across different data formats for various datasets

Each query is executed 10 times, and the average execution time of all post-warm-up runs is reported to minimize the impact of caching and execution bias. To facilitate direct performance comparison across data representations, we normalize the execution time of each query by setting the execution time on the Optimized Graph to 1. The relative performance of Relational Tables and Basic Graphs is then computed accordingly. The comparative query performance results across all datasets are presented in Figure 12.

5.3 Main Results

Section 5.2 presented schema evolution and query performance across data models. Here, we summarize key findings demonstrating GalaxyWeaver's effectiveness in transforming and optimizing graph schemas for complex analytics.

5.3.1 Graphs Outperform Tables for Multi-Table Join Queries. One of the key advantages of transforming relational tables into graphs is the improved handling of multi-table join queries. GalaxyWeaver is specifically designed as a transformation framework tailored for analytical workloads that involve complex join operations across multiple tables. To reflect this focus, the query workloads in our experiments were deliberately designed to emphasize such scenarios.

As shown in Figure 12, for almost all multi-table join queries, both Basic Graphs and Optimized Graphs consistently outperform the corresponding Relational Tables. The only exception is Query 1 from the AB dataset, which retrieves review details by review ID — a simple primary key lookup. For this type of direct entity retrieval, relational tables and graph-based representations show comparable performance, as illustrated in Figure 12b. This further highlights that the performance benefits of graph-based approaches emerge primarily in queries requiring complex relationship traversals, rather than in isolated entity lookups.

5.3.2 Converting Vertices to Edges Reduces Query Hops. For datasets with rich relational structures, such as SE and MAG, converting certain vertices into edges proves particularly beneficial for reducing query complexity. Taking MAG as an example, the Basic Graph (Figure 11b) models foreign key relationship tables — such as `Cite`, `HasTopic`, `Write`, and `AffiliatedWith` — as individual vertex types. As a result, a query requiring a 6-table join in the relational schema translates into a graph traversal spanning at least 6 hops in the Basic Graph.

In contrast, the Optimized Graph (Figure 11c) flattens these intermediate relationship vertices into direct edges between entity vertices, reducing the required traversal length to just 3 hops. This restructuring not only simplifies query execution but also significantly enhances performance. As shown in Figures 12c and 12d, the execution time of the Basic Graph is 1.4 to 6.3 times that of the Optimized Graph, depending on the complexity of the query—such as the number of hops required.

5.3.3 Converting Vertex Properties to Edge Properties Reduces Redundant Access. For queries involving property-based filtering along relational paths, relocating key properties from vertices to edges streamlines execution. This is exemplified by Query 3 in the AB dataset, which identifies products with reviews where the product price exceeds a specified threshold.

In the Basic Graph (Figure 9b), the `price` property resides on the `Product` vertex. Executing this query requires traversing from `Review` vertices to `Product` vertices in order to access the `price` property, introducing unnecessary traversal overhead.

In the Optimized Graph (Figure 9c), GalaxyWeaver relocates the `price` property directly onto the edges connecting `Review` vertices to `Product` vertices. This allows the query to filter directly on the edge property during traversal, eliminating the need for extra vertex lookups. As shown in Figure 12b, the execution time on the Basic Graph is 1.5 times that of the Optimized Graph for Query 3, which relies on property filtering.

5.3.4 Extracting Key Properties into Independent Vertices Enhances Filtering Efficiency. Another beneficial optimization involves elevating important properties into standalone vertices to facilitate more efficient filtering and indexing. This technique is demonstrated by Query 2 of the AB dataset, which counts reviews for products within a specific `category`.

In the Basic Graph (Figure 9b), `category` exists as a non-key property on the `Product` vertex. Without an additional index, evaluating this query requires scanning all products to find those matching the desired category.

GalaxyWeaver addresses this inefficiency by promoting `category` to a separate vertex (Figure 9c), directly linking products to their categories. This allows for efficient category-based filtering. As shown in Figure 12b, the execution time on the Basic Graph is 2.8 times that of the Optimized Graph for Query 2.

5.3.5 Eliminating Redundant Vertices, Edges, and Properties Streamlines the Schema. Schema simplification is another critical aspect of GalaxyWeaver's optimization process. In the AVS dataset (Figure 8), entities such as `Brand`, `Chain`, and `Category`, while present in the original relational schema, are unused by the actual query workload. GalaxyWeaver automatically identifies and removes these redundant elements during schema transformation, resulting in a more compact and query-efficient graph schema.

This reduction in unnecessary vertices, edges, and properties not only simplifies the schema but also reduces memory usage and storage overhead, as illustrated by the changes in graph statistics from the Basic Graph to the Optimized Graph in Table 4.

5.4 Case Study

A manufacturing client migrating from a legacy ERP system to Galaxybase faced significant challenges in data modeling, with over 40 relational tables and more than 300 attributes. The analytical workload primarily involved complex multi-table joins covering production orders, inventory movements, supplier relationships, and maintenance logs—requiring frequent traversals across business entities.

Traditionally, Galaxybase experts manually analyzed table relationships and business logic to design the graph schema from scratch. This process took approximately 20 person-days due to the need to understand domain-specific semantics and iteratively refine the schema. With GalaxyWeaver, the initial schema transformation was completed within tens of minutes. Graph experts then reviewed and fine-tuned the automatically generated schema, completing the entire design process in just a few days—achieving over 80% time savings. The final schema included 11 vertex types and 17 edge types.

GalaxyWeaver's optimizations also resulted in significant performance improvements. In one representative scenario, a query traversing four hops from production orders to supplier vertices—was automatically simplified to a three-hop traversal using the *Convert Vertices to Edges* rule. We benchmark this improvement across several production orders. On average, query latency dropped from around 2 seconds to just 20 milliseconds. In the most extreme case, the query execution time was reduced from approximately 15 seconds to 50 milliseconds.

However, due to the inherent instability and non-determinism of current LLMs, GalaxyWeaver is positioned as a decision-support tool rather than a fully autonomous solution for production environments. By integrating automated schema generation with expert validation, GalaxyWeaver significantly reduces the cost and complexity of table-to-graph transformation while ensuring schema quality and reliability for critical business applications.

6 RELATED WORK

Table-to-Graph Conversion. Existing table-to-graph conversion approaches can be broadly categorized into three types. Direct mapping methods [18, 35, 37] convert primary keys into vertices and foreign keys into edges, preserving the original relational structure within the graph. Manual design approaches [6, 13] allow users to define customized mappings, offering greater flexibility to meet specific analytical needs. More recently, machine learning techniques have been employed to predict graph structures based on data patterns, thereby automating parts of the process [29, 38]. These methods offer varying levels of automation and flexibility, laying the groundwork for GalaxyWeaver's approach to table-to-graph conversion.

Graph Schema Optimization. Graph schema optimization focuses on improving both query performance and data interpretability by restructuring graph elements. Prior work has explored static schema transformations [10], the conversion of RDF data into property graphs [27], and techniques for refining property graph structures to enhance query efficiency [11]. Building on this foundation, GalaxyWeaver further integrates query workload analysis into the optimization process, enabling the graph schema to be tailored to the specific characteristics of analytical queries.

LLM Agents. LLM agents have demonstrated significant potential in automating complex reasoning and decision-making tasks across a wide range of domains. AutoGen [40] provides a versatile framework for developing applications of varying complexity and LLM capabilities, while ChatDev [26] offers a highly customizable and extendable multi-agent framework designed to explore collaborative intelligence among LLM agents. In the domain of database management, D-Bot [46] leverages LLM agents to automatically extract diagnostic knowledge and assist with query optimization. Building on these developments, GalaxyWeaver employs LLM agents to support graph schema design and adaptive optimization, utilizing their natural language understanding and reasoning capabilities to lower the barrier to graph modeling. To further improve reliability, we plan to incorporate reflection mechanisms into the agent workflow, allowing the LLM to assess its own rule-based decisions, identify uncertainties, and support expert validation more effectively.

7 CONCLUSION

This paper introduces GalaxyWeaver, an LLM-powered framework for automating table-to-graph transformation with query-driven schema optimization. Combining prompt-guided analysis, domain knowledge, and proven optimization strategies from real-world deployments, GalaxyWeaver generates efficient graph schemas tailored to analytical needs. Experiments across diverse domains demonstrate that GalaxyWeaver reduces manual effort, improves query performance, and lowers storage overhead, providing an effective solution for bridging relational data and graph analytics.

ACKNOWLEDGMENTS

This work was supported by NSFC Grant No. 62206067 and the HKUST(GZ)-Chuanglin Graph Data Joint Lab. We sincerely thank the anonymous reviewers for their valuable feedback. We are also grateful to the entire team at CreateLink (Chuanglin) for their design and development of GalaxyWeaver and its underlying distributed graph database, Galaxybase. Additionally, we appreciate the professional guidance and insightful feedback from the team at the HKUST(GZ)-Chuanglin Graph Data Joint Lab.

REFERENCES

[1] 2024. *Neo4j*. https://neo4j.com/

[2] Renzo Angles. 2018. The Property Graph Database Model.. In *AMW*.

[3] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Alastair Green, Jan Hidders, Bei Li, Leonid Libkin, Victor Marsault, Wim Martens, et al. 2023. PG-Schema: Schemas for property graphs. *Proceedings of the ACM on Management of Data* 1, 2 (2023), 1–25.

[4] Renzo Angles, Angela Bonifati, Stefania Dumbrava, George Fletcher, Keith W Hare, Jan Hidders, Victor E Lee, Bei Li, Leonid Libkin, Wim Martens, et al. 2021. Pg-keys: Keys for property graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 2423–2436.

[5] Anthropic. [n.d.]. The Claude 3 Model Family: Opus, Sonnet, Haiku. https://api.semanticscholar.org/CorpusID:268232499. Accessed: 2, 6.

[6] Nafisa Anzum. 2020. *Systems for Graph Extraction from Tabular Data*. Master's thesis. University of Waterloo.

[7] Marcelo Arenas, Claudio Gutiérrez, and Juan F Sequeda. 2021. Querying in the age of graph databases and knowledge graphs. In *Proceedings of the 2021 International Conference on Management of Data*. 2821–2828.

[8] Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. 2013. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 1185–1196.

[9] Paolo Atzeni and Valeria De Antonellis. 1993. *Relational database theory*. Benjamin-Cummings Publishing Co., Inc.

[10] Iovka Boneva, Benoit Groz, Jan Hidders, Filip Murlak, and Slawek Staworko. 2023. Static analysis of graph database transformations. In *Proceedings of the 42nd ACM SIGMOD-SIGACT-SIGAI Symposium on Principles of Database Systems*. 251–261.

[11] Angela Bonifati, Filip Murlak, and Yann Ramusat. 2024. Transforming Property Graphs. *Proc. VLDB Endow.* 17, 11 (Aug. 2024), 2906–2918. https://doi.org/10.14778/3681954.3681972

[12] Nathan Bronson, Zach Amsden, George Cabrera, Prasad Chakka, Peter Dimov, Hui Ding, Jack Ferris, Anthony Giardullo, Sachin Kulkarni, Harry Li, Mark Marchukov, Dmitri Petrov, Lovro Puzar, Yee Jiun Song, and Venkat Venkataramani. 2013. TAO: Facebook's distributed data store for the social graph. In *Proceedings of the 2013 USENIX Conference on Annual Technical Conference* (San Jose, CA) *(USENIX ATC'13)*. USENIX Association, USA, 49–60.

[13] Patrícia Cavoto and André Santanchè. 2015. ReGraph: bridging relational and graph databases. In *Proceedings of the 30th Brazilian Symposium on Databases*.

[14] Edgar F Codd. 2007. Relational database: A practical foundation for productivity. In *ACM Turing award lectures*. 1981.

[15] Alin Deutsch, Yu Xu, Mingxi Wu, and Victor Lee. 2019. Tigergraph: A native MPP graph database. *arXiv preprint arXiv:1901.08248* (2019).

[16] DMDave, Todd B., and Will Cukierski. 2014. Acquire Valued Shoppers Challenge. https://kaggle.com/competitions/acquire-valued-shoppers-challenge.

[17] Orri Erling, Alex Averbuch, Josep Larriba-Pey, Hassan Chafi, Andrey Gubichev, Arnau Prat, Minh-Duc Pham, and Peter Boncz. 2015. The LDBC social network benchmark: Interactive workload. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*. 619–630.

[18] Hui Feng and Meigen Huang. 2022. An approach to converting relational database to graph database: From MySQL to Neo4j. In *2022 IEEE 2nd International Conference on Power, Electronics and Computer Applications (ICPECA)*. IEEE, 674–680.

[19] Daya Guo, Dejian Yang, Haowei Zhang, Junxiao Song, Ruoyu Zhang, Runxin Xu, Qihao Zhu, Shirong Ma, Peiyi Wang, Xiao Bi, et al. 2025. Deepseek-r1: Incentivizing reasoning capability in llms via reinforcement learning. *arXiv preprint arXiv:2501.12948* (2025).

[20] Jan L Harrington. 2016. *Relational database design and implementation*. Morgan Kaufmann.

[21] Richard Henderson. 2020. Using graph databases to detect financial fraud. *Computer Fraud & Security* 2020, 7 (2020), 6–10.

[22] Aaron Hurst, Adam Lerer, Adam P Goucher, Adam Perelman, Aditya Ramesh, Aidan Clark, AJ Ostrow, Akila Welihinda, Alan Hayes, Alec Radford, et al. 2024. Gpt-4o system card. *arXiv preprint arXiv:2410.21276* (2024).

[23] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying recommendations using distantly-labeled reviews and fine-grained aspects. In *Proceedings of the 2019 conference on empirical methods in natural language processing and the 9th international joint conference on natural language processing (EMNLP-IJCNLP)*. 188–197.

[24] Anil Pacaci, Alice Zhou, Jimmy Lin, and M Tamer Özsu. 2017. Do we need specialized graph databases? Benchmarking real-time social networking applications. In *Proceedings of the Fifth International Workshop on Graph Data-management Experiences & Systems*. 1–7.

[25] Debachudamani Prusti, Daisy Das, and Santanu Kumar Rath. 2021. Credit card fraud detection technique by applying graph database model. *Arabian Journal for Science and Engineering* 46, 9 (2021), 1–20.

[26] Chen Qian, Xin Cong, Cheng Yang, Weize Chen, Yusheng Su, Juyuan Xu, Zhiyuan Liu, and Maosong Sun. 2023. Communicative agents for software development. *arXiv preprint arXiv:2307.07924* 6, 3 (2023).

[27] Kashif Rabbani, Matteo Lissandrini, Angela Bonifati, and Katja Hose. 2025. Transforming RDF Graphs to Property Graphs using Standardized Schemas. (2025).

[28] Ian Robinson, Jim Webber, and Emil Eifrem. 2015. *Graph databases: new opportunities for connected data*. O'Reilly Media, Inc.

[29] Joshua Robinson, Rishabh Ranjan, Weihua Hu, Kexin Huang, Jiaqi Han, Alejandro Dobles, Matthias Fey, Jan Eric Lenssen, Yiwen Yuan, Zecheng Zhang, et al. 2024. Relbench: A benchmark for deep learning on relational databases. *Advances in Neural Information Processing Systems* 37 (2024), 21330–21341.

[30] Arnab Sinha, Zhihong Shen, Yang Song, Hao Ma, Darrin Eide, Bo-June Hsu, and Kuansan Wang. 2015. An overview of microsoft academic service (mas) and applications. In *Proceedings of the 24th international conference on world wide web*. 243–246.

[31] StackExchange. n.d.. StackExchange Data Explorer. https://data.stackexchange.com/.

[32] Fabian M Suchanek, Gjergji Kasneci, and Gerhard Weikum. 2007. Yago: a core of semantic knowledge. In *Proceedings of the 16th international conference on World Wide Web*. 697–706.

[33] Jianheng Tang, Jiajin Li, Ziqi Gao, and Jia Li. 2022. Rethinking graph neural networks for anomaly detection. In *International Conference on Machine Learning*. PMLR, 21076–21089.

[34] Bing Tong, Yan Zhou, Chen Zhang, Jianheng Tang, Jing Tang, Leihong Yang, Qiye Li, Manwu Lin, Zhongxin Bao, Jia Li, and Lei Chen. 2024. Galaxybase: A High Performance Native Distributed Graph Database for HTAP. *Proc. VLDB Endow.* 17, 12 (Nov. 2024), 3893–3905. https://doi.org/10.14778/3685800.3685814

[35] Yelda Unal and Halit Oguztuzun. 2018. Migration of data from relational database to graph database. In *Proceedings of the 8th International Conference on Information Systems and Technologies*. 1–5.

[36] A Vaswani. 2017. Attention is all you need. *Advances in Neural Information Processing Systems* (2017).

[37] Harsha R Vyawahare, Pravin P Karde, and Vilas M Thakare. 2019. An efficient graph database model. *Int. J. Innov. Technol. Explor. Eng* 88, 10 (2019), 1292–1295.

[38] Minjie Wang, Quan Gan, David Wipf, Zheng Zhang, Christos Faloutsos, Weinan Zhang, Muhan Zhang, Zhenkun Cai, Jiahang Li, Zunyao Mao, et al. [n.d.]. 4DBInfer: A 4D Benchmarking Toolbox for Graph-Centric Predictive Modeling on RDBs. In *The Thirty-eight Conference on Neural Information Processing Systems Datasets and Benchmarks Track*.

[39] Jason Wei, Maarten Bosma, Vincent Y Zhao, Kelvin Guu, Adams Wei Yu, Brian Lester, Nan Du, Andrew M Dai, and Quoc V Le. 2021. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652* (2021).

[40] Qingyun Wu, Gagan Bansal, Jieyu Zhang, Yiran Wu, Beibin Li, Erkang Zhu, Li Jiang, Xiaoyun Zhang, Shaokun Zhang, Jiale Liu, et al. 2023. Autogen: Enabling next-gen llm applications via multi-agent conversation. *arXiv preprint arXiv:2308.08155* (2023).

[41] Tianyu Wu, Shizhu He, Jingping Liu, Siqi Sun, Kang Liu, Qing-Long Han, and Yang Tang. 2023. A brief overview of ChatGPT: The history, status quo and potential future development. *IEEE/CAA Journal of Automatica Sinica* 10, 5 (2023), 1122–1136.

[42] Jihong Yan, Chengyu Wang, Wenliang Cheng, Ming Gao, and Aoying Zhou. 2018. A retrospective of knowledge graphs. *Frontiers of Computer Science* 12 (2018), 55–74.

[43] Chen Zhang, Jing Wu, and Yan Zhou. 2024. *Graph Databases: Theory and Practice*. Electronics Industry Press.

[44] TV Zhidchenko, MN Seredina, NM Udintsova, and NA Kopteva. 2021. Design of energy-loaded systems using the Neo4j graph database. In *IOP Conference Series: Earth and Environmental Science*, Vol. 659. IOP Publishing, 012108.

[45] Wangchunshu Zhou, Yuchen Eleanor Jiang, Long Li, Jialong Wu, Tiannan Wang, Shi Qiu, Jintian Zhang, Jing Chen, Ruipu Wu, Shuai Wang, et al. 2023. Agents: An open-source framework for autonomous language agents. *arXiv preprint arXiv:2309.07870* (2023).

[46] Xuanhe Zhou, Guoliang Li, Zhaoyan Sun, Zhiyuan Liu, Weize Chen, Jianming Wu, Jiesi Liu, Ruohang Feng, and Guoyang Zeng. 2023. D-bot: Database diagnosis system using large language models. *arXiv preprint arXiv:2312.01454* (2023).

[47] Jun-Peng Zhu, Peng Cai, Kai Xu, Li Li, Yishen Sun, Shuai Zhou, Haihuang Su, Liu Tang, and Qi Liu. 2024. AutoTQA: Towards Autonomous Tabular Question Answering through Multi-Agent Large Language Models. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3920–3933.