An Effective Framework for Enhancing Query Answering in a Heterogeneous Data Lake

Anonymous Author(s)

ABSTRACT
There has been a growing interest in cross-source searching to gain rich knowledge in recent years. A data lake collects massive raw and heterogeneous data with different data schemas and query interfaces. Many real-life applications require query answering over the heterogeneous data lake, such as e-commerce, bioinformatics and healthcare. In this paper, we propose LakeAns that semantically integrates heterogeneous data schemas of the lake to enhance the semantics of query answers. To this end, we propose a novel framework to efficiently and effectively perform the cross-source searching. The framework exploits a reinforcement learning method to semantically integrate the data schemas and further create a global relational schema for the heterogeneous data. It then performs a query answering algorithm based on the global schema to find answers across multiple data sources. We conduct extensive experimental evaluations using real-life data, and our approach outperforms existing solutions in terms of effectiveness and efficiency.

1 INTRODUCTION
The increasing diversity of information and products has led to the evolution of search and recommendations from primarily uni-directional and text-based to multi-model and heterogeneous data sources [29, 46]. Many industrial search systems have emerged interested in discovering a heterogeneous data lake to enhance the answer semantics and the search quality [14, 22]. A data lake includes multiple data models (e.g., structured, semi-structured, and unstructured) with different data schemas and query interfaces [24, 33]. For instance, an enterprise data lake as shown in Figure 1 naturally organizes transaction data in different data formats and models. Both user and product data are maintained in relational tables (see Figures 1(a)-(b)), which associates with the relation schema as shown in Figure 1(c). Orders and social networks are stored as a JSON document and a graph, respectively (see Figures 1(d)-(e)).

Exploring these data to obtain more enriching and complete knowledge is an important task for data management and applications, such as pharmacy, bioinformatics and healthcare [10, 18]. Consider an analysis task over the social e-commerce lake as shown in Figure 1, analysts would like to determine whether to recommend a product to a user Miller. This requires inspecting the holistic view of the customer information, i.e. (i) the personalized features and (ii) the social relationship of Miller. It is necessary to check the user and product information in Figures 1(a) and (b) for condition (i), and the social graph in Figure 1(e) for condition (ii).

For a holistic insight into the heterogeneous data lake, we focus on performing cross-source searching using a general query. Benefiting from the theories and techniques of relational database management systems (RDBMS), we utilize the SQL query to search for answers across multiple heterogeneous data sources. The following example motivates our work.
Clearly, $R(Q)$ spans three different data models and completely captures the relational information for customers, which is exactly what the user wants or could be interested in.

Cross-source query answering also occurs in medicine discovery scenarios where biologists would check the basic pharmacological features of drugs and the interactions among drugs and protein molecules in the human body. This paper aims to create a global relational schema for querying across heterogeneous data sources. We will support SQL queries across multiple data sources, and to the best of our knowledge, this is the first paper to consider a global relational schema for this purpose.

The query answering in a data lake faces many challenges presented as follows. Firstly (C1), how to create a global relational schema that bridges the schema associations between different sources is the first concern due to the heterogeneity and autonomy of the data lake. Some relational schema learning approaches have been proposed for a single data model such as for key-value stores [8, 12, 45], for uncertain data [30]. These cannot be adapted to the lake with multi-model data sources because they fail to extract the same or similar semantics from heterogeneous concepts or classes in different sources. A recent work [50] focuses on storing multi-model data in RDBMS, which differs from our goal of supporting SQL queries and exploring richer cross-source answers. Secondly (C2), how to efficiently perform query answering based on an integrated relational schema to enhance the answer semantics of a SQL query over the data lake. Most works have been studied to augment data lake tables [13, 27], such as Infogather [48] is designed to obtain more complete answers via entity augmentation and attribute discovery. These are restricted to relational data and do not enrich semantics with a global schema. Additionally, some engines have been proposed to process distributed SQL, such as MuSQLE [20], Presto [38], CockroachDB [43] and Polaris [5], but do not consider that in a data lake with heterogeneous data models and query engines.

**Main idea.** To tackle these challenges, we present a schema integration-based query answering framework. Every source $D_i$ maintains a local schema $C_i$, and a mediator $M$ manages a global relational schema $C_g$ that provides the schema associations between different sources, and the mappings from $C_g$ to $C_i$. A SQL query is formulated based on $C_g$ and submitted to the mediator $M$. The framework consists of schema integration and query answering, detailed in Section 3. We present classification-based method and an edge table-based approach to break semantic gap between different data sources, and then build the global relational schema to unify the semantics between the local schemas using a reinforcement learning approach (for C1). The global schema generation is considered a Markov decision process and is motivated by three types of actions and a novel reward function based on a Q-learning algorithm, introduced in Section 4. And then we present a query answering approach based on schema integration to enhance the answer semantics of SQL queries (for C2). Specifically, we enrich logical search spaces by bridging the schema gap between the query and the source schemas based on $F_q$. Semantic joining is proposed to establish semantic associations between data instances of different data sources to compute the final answers of the query, illustrated in Section 5. Finally, we experimentally verify that LakeAns can find the answers with greater coverage rate spending almost the same time with the methods performing local evaluation in a single data source, detailed in Section 6.

## 2 PROBLEM DEFINITION

### 2.1 Preliminary

**Data lake.** A data lake denoted by $D = \bigcup_{i=1}^{n} D_i$, collects a set of heterogeneous data sources, such as JSON documents, graphs, relational data and multimedia data (e.g., videos, images). Here we mainly concern three general data models JSON, graphs, and relational databases presented as follows. Unstructured multimedia data can be converted into structured scene graphs [47, 51] by extracting all objects and their relations from images and videos, and will be concerned in our future works.

**JSON.** A JSON object is a collection of key-value pairs $\{k_1 : a_1, ..., k_l : a_l\}$ that maps the key $k_i$ to the value $a_i$. The JSON value can be a boolean value, a numeric value, a string value, an array and an object.

**Graph.** A graph is defined by $G = (V, E, L)$, where $V$ is a set of nodes, and $E \subseteq V \times V$ is a set of edges. $L(e)$ and $L(v)$ represent the labels of $e \in V$ and $v \in E$, respectively.

**Relational database.** A relation schema $R$ is associated with a set of attributes of the form $R(A_1, ..., A_k)$, where $A_i$ is an attribute with domain $\text{dom}(A_i)$, and the attribute $A_i$ of $R$ is written as $R[A_i]$, $i \in \{1, k\}$. A relation $R_g$ of schema $R$ is a set of tuples with attributes $A_i(i \in \{1, k\})$ of $R$. A database schema is a pair $(R, \Sigma)$, where $\Sigma = (R_1, ..., R_m)$ is a finite set of relation schemas, and $\Sigma$ is a set of constraints. A relational database $D$ is $(D_1, ..., D_m)$ that satisfies all constraints in $\Sigma$, where $D_i$ is a relation of schema $R_i$ for $i \in \{1, n\}$.

These data models have heterogeneous schema semantics and query interfaces. Below we discuss the key challenges of unifying the data schemas and performing a general SQL query.

### 2.2 Problem Statement

Schema integration is an effective approach to bridge heterogeneous schema semantics and provide a unified query interface for the data lake. For example, $C_1$, $C_2$ and $C_3$ shown in Figure 3 are the local schemas of the relational database $D_1$, the JSON document $D_2$ and the graph $D_3$ in Figure 1, respectively. We formally present a relational schema graph as follows.

**Relational schema graph.** A relational schema graph is modeled as a property graph $S = (V_s, E_s, L_s, F_s)$, where $V_s$ and $E_s$ are a set of nodes and edges, respectively. $L_s$ is a labeling function such that for each node $v \in V_s$, $L_s(v)$ is a node label. $F_s(v)$ is a set of attributes $\{A_1, ..., A_k\}$ for each node $v$.

In practice, a relational schema graph illustrates the relation schemas and their reference constraints in a database schema. The relation schema $R(A_1, ..., A_k)$ is specified by a node $v \in V_s$ whose label $L_s(v)$ is the relation name $R$, and $F_s(v)$ captures its attributes $A_i$ for $i \in \{1, k\}$. Each edge indicates the semantic relationships among nodes. In this paper, an edge from $u$ to $v$ indicates that there is a reference from $R_1$ to $R_2$, where $R_1$ and $R_2$ are relation schemas encoded by $u$ and $v$, respectively.

The local schema graph $C_1$ of $D_1$ shown in Figure 3 describes the relational database schema in Figure 1(c), which includes an edge connecting two relation schemas “User(name, location, buy)”
As for a data lake

Given a SQL query

which aims at exploring the complete answers for SQL queries across multiple heterogeneous data sources in the data lake.

and “Product(title, color)”. In Section 4.2, we will introduce how to create a local schema for different data sources.

To integrate heterogeneous data schemas and effectively support query answering across multiple data sources, we create a global relational schema graph, namely $C_g$ shown in Figure 3. $C_g$ is generalized from the local schemas of the heterogeneous data lake and stored in a mediator $M$, which will be detailed in Section 4.3. How to create an effective global schema $C_g$ for $D$ and efficiently support SQL queries based on it is our main concern in this paper.

To uniformly query the heterogeneous data lake, we study the query answering of SQL queries with SELECT-FROM-WHERE clauses. A SQL query $Q$ is formulated based on the global relational schema $C_g$, and its answer after performing $Q$ over the data lake is formally defined as follows.

**Query answer.** Given a SQL query $Q$ over the data lake $D$, an answer for $Q$ is a set of tuples with attributes $(A_1, ..., A_k)$, denoted by $T(Q)$, where (1) each $A_i (1 \leq i \leq k)$ corresponds to an attribute in the SELECT clause of $Q$; and (2) each attribute values is an instance value of $D$.

Note that (i) $A_j$ may be an attribute of the relational database, a key of the JSON object and an edge label of the graphs; and (ii) the answer of $Q$ follows a relational schema $R(A_1, ..., A_k)$, which may span multiple data sources of $D$ and can be deduced from $C_g$.

**Example 2.** Figure 2 shows the answer of $Q$ over the data lake shown in Figure 1. The answer $T(Q)$ spans three data sources, i.e., $D_1$, $D_2$ and $D_3$, and follows a schema $R$ with attributes $A = \{ \text{name}, \text{knows}, \text{title}, \text{brand} \}$ specified by the SELECT clause of $Q$. The schema $R$ is deduced by joining the relation schemas “Customer” and “Product” based on $C_g$ and projecting into the attribute set $A$.

3 QUERy ANSWERING FRAMEWORK

Figure 4 illustrates our query answering framework (namely LakeAns), which aims at exploring the complete answers for SQL queries across multiple heterogeneous data sources in the data lake.

The LakeAns consists of two components: offline schema integration and online query answering.

**Schema integration.** As for a data lake $D$, we formulate a schema integration paradigm $S$, which consists of a global schema $C_g$, a set of local schemas $C_l$ extracted from $D$, and a mapping functor $F_g$. $C_l$ is built to bridge the schema gap between heterogeneous data sources. $C_g$ provides global relational schema and uniformly general query interface for data scientists, which is created by integrating the local schemas based on a reinforcement learning approach. The $F_g$ maps the concepts from $C_g$ to $C_l$ and benefits to query rewriting and across-sources access over the data lake. Details will be presented in Section 4.

**Query answering.** Given a SQL query $Q$ formulated based on $C_g$, it is answered based on the integrated schema to enhance the query semantics, as introduced in Section 5. We enrich logical plans of $Q$ based on the local relational schema $C_l$ and the mapping functor $F_g$. Thereafter, the local results computed from different data sources are required to perform semantic joining to deduce a relational schema based on the local schema $C_l$ to obtain the final answers for $Q$.

4 SCHEMA INTEGRATION

In this section, a schema integration approach is developed to create a global relational schema graph to establish semantic associations across different data sources in the data lake.

4.1 Method Overview

Given a data lake $D = \{D_1, ..., D_m\}$, we define a schema integration paradigm as $S = \{C_l, C_g, F_g\}$. As shown in Figure 3, we have $C_l = \bigcup_{i=1}^{m} C_l$ and $F_g = \bigcup_{i=1}^{m} F_i$.

As for $C_l$, a classification-based method and an edge table-based approach are introduced to create the local relational schema for the JSON documents and the graphs, respectively. Details will be described in Section 4.2.

To unify the schema semantics among the local schemas, a global relational schema $C_g$ is required. We consider the generation of $C_g$ as a Markov decision process that aims to find the best relational schema (i.e., state) for $D$. This is done by performing a sequence of transformations (i.e., actions) on the local schemas to efficiently support cross-source answering of SQL queries. Starting from the local schemas $C_l$, $C_g$ is created utilizing a reinforcement learning approach motivated by three types of actions and a new reward function. Details will be introduced in Section 4.3.

In the integration paradigm $S$, a query can be uniformly formulated by using $C_g$ and completely answered based on $F_g$ and $C_l$ over the heterogeneous data sources $D$.

4.2 Local Schema Construction

For each heterogeneous data of $D$, local relational schema is constructed by doing the following two steps: (1) extracting a data schema, and (2) transforming it into relational database schema.
Given a graph database, we summarize a graph schema by merging vertices with the same local schema. In this section, we introduce a reinforcement learning method (namely LakeGrsg) to generate a global relation schema, which requires to augment the semantic associations across multiple data sources and to further efficiently support relational queries in the data lake.

4.3 Global Schema Generation

In this section, we introduce a reinforcement learning method (namely LakeGrsg) to generate a global relation schema, which requires to augment the semantic associations across multiple data sources and to further efficiently support relational queries in the data lake.

4.3.1 Idea of LakeGrsg. Given a set of SQL queries and the local schema $C_l$ of $D_l$, LakeGrsg aims to create a global relational schema $C_g$ based on a Q-learning algorithm, such that these queries can obtain the most richest answer semantics with the least cumulative query time.

We consider the generation of $C_g$ as a Markov decision process defined by a quadruple $(S, A, T, R)$, where $S$ is a state space, $A$ is a set of actions, $T$ is a transition function linking state-action pairs to new states, and $R$ is a reward function that reports a reward value for a state-action pair. In this paper, we design three types of actions, i.e., linking, merging and splitting. The state is a relational schema graph generated by taking the action. This process generates $C_g$ by interacting an agent with an environment. Figure 6 illustrates the overview of LakeGrsg where the environment is a RDBMS. A Q-learning algorithm is used to take actions on the environment to maximize the expected rewards, which is recorded in a Q-table for action-state pairs. In each step, the agent greedily select the action with maximum expected rewards according to the Q-table and change the state of the environment. The state and reward are fed back into the agent to perform the next action.

Specifically, taking $C_l$ as an initial state, the agent works in the following steps: for each iteration, (1) it selects an action $a_t$ and creates a new state $s_t$; (2) the given SQL queries are performed based on the new state; and (3) it computes a reward $r$ and updates the value of Q-table based on the generated state. This process is repeated until the maximum number of iterations is reached or a finite number of actions are tried, and we can finally obtain a global relational schema $C_g$ for $D$.

The multi-model data is stored as relational tables based on $C_l$ to support the execution of the given SQL queries. Furthermore, the functor $F_l$ that records the mappings of concepts (i.e., relation names and attributes) between $C_g$ and $C_l$ would be updated when an action is taken in each iteration.

To enable to calculate the semantics of relations and instances, we project the string values into an embedding space based on the fastText database [26], which can handle out-of-vocabulary words by learning character embeddings. As for a relation schema $R(A_1, ..., A_k)$, $\hat{R}$ represents the embedding vector projected by the value of $R$. We utilize $\Pi_g$ to represent the feature space associated with the instance values of $R$. Let $dom(A_i)$ be the instances of $A_i$ in $R$, and $\Pi_g(A_i)$ be the embedding set of $dom(A_i)$. A pivot vector is used to represent the sample mean of $\Pi_g(A_i)$, which is calculated by the embedding mean of the values in $dom(A_i)$, denoted by $\mu(A_i)$.

We next introduce the action identification and a Q-learning based state update.

4.3.2 Actions. Action space is defined by a set of transformations (i.e., actions) about the relational schemas motivated by the unique attributes in $C_l$. This means that when an attribute is selected, the

---

1https://json-schema.org
agent will determine how to transform the relational schema associated with it. Based on the transformation, we further define three types of actions: linking, merging and splitting to integrate the initial relation schema.

Linking actions are introduced to complement implicit references across multiple local schemas and unify the schema semantics of different data sources. Splitting and merging actions are developed to normalize schemas and make them to effectively support relational query processing. Before detailing these actions, we first describe how to identify whether an attribute \(A_i\) of relation schema \(R_1\) can join with the attribute \(B_j\) in the relation schema \(R_2\).

Given two relation schemas \(R_1(A_1, ..., A_k)\) and \(R_2(B_1, ..., B_l)\), \(R_1\) and \(R_2\) are semantically joinable on attributes \(A_i\) and \(B_j\), if (a) \(A_i\) and \(B_j\) refer to the same or similar concept; and (b) \(A_i\) and \(B_j\) have similar value domains. Specifically, we first check how close are \(A_i\) and \(B_j\) in the embedding space. If the similarity is greater than threshold \(\alpha\), we measure the distance of \(\mu(A_i)\) and \(\mu(B_j)\) using the cosine similarity. We say that \(R_1\) and \(R_2\) are semantically joinable on \(A_i\) and \(B_j\) if \(\cos(\mu(A_i), \mu(B_j)) \geq \alpha\) and \(\cos(\mu(A_i), \mu(B_j)) \geq \theta\) where \(\alpha\) and \(\theta\) are the thresholds for concept similarity and pivot vector distance, respectively.

**Linking action.** There is a linking action on two joinable attributes \(R_1[A_i]\) and \(R_2[B_j]\), which inserts an edge connecting \(A_i\) of \(R_1\) and \(B_j\) of \(R_2\). Essentially, the linking action corresponds to complementing a foreign key reference between \(R_1\) and \(R_2\). Accordingly, \(F_g\) stores a pair of attributes \(R_1[A_i]\) and \(R_2[B_j]\), which are semantically equivalent concepts.

**Merging action.** To reduce schema redundancy and avoid frequent expensive join operations, we introduce merging action to combine the relation schemas. As for \(R_1(A_1, ..., A_k)\) and \(R_2(B_1, ..., B_l)\), \(R_1\) is contained in \(R_2\), denoted by \(R_1 \subseteq R_2\), if the following conditions are satisfied: (a) \(R_1\) and \(R_2\) have the same or similar relation name; and (b) for each attribute \(A_i\) of \(R_1\), there is an attribute \(B_j\) in \(R_2\) that is similar to its concept and domain. That is, we have \(R_1 \subseteq R_2\) if \(\cos(\mu(R_1), \mu(R_2)) \geq \lambda\), \(\cos(\mu(A_i), \mu(B_j)) \geq \alpha\) and \(\cos(\mu(A_i), \mu(B_j)) \geq \theta\), where \(\lambda\) is a distance threshold for the relation name.

In this case, a merging action is required to combine \(R_1\) and \(R_2\), which inserts all of the foreign key references of \(R_1\) into the corresponding attributes of \(R_2\). Therefore, \(F_g\) updates the mappings associated with \(R_1\) to those of \(R_2\).

**Splitting action.** To effectively support query processing and improve schema readability, we perform a splitting action when there are too many references included in a relation schema. Given a schema \(R\), splitting action corresponds to a projection of \(R\), which divides \(R\) as a schema set \(P = \{P_1, ..., P_s\}\) such that (1) \(A = \bigcup_{1 \leq i \leq s} P_i\), where \(A_i\) is the attribute set of \(P_i\); (2) \(I_{P_i} = \bigcup_{1 \leq i \leq s} I_{P_i}\) for \(i \in [1, s]\), and \(I_{P_i} \supseteq ... \supseteq I_{P_s} = I_R\) where \(\supseteq\) is the join operations of \(I_{P_i}, ..., I_{P_s}\) in the embedding space \(I_R\); (3) There is a foreign key reference from \(P_i[A_{P_i}]\) to \(P_j[A_{P_j}]\), \(1 \leq i, j \leq s\), where \(A_P\) and \(A_R\) are the joinable attributes for \(P_i\) and \(P_j\).

The first and third conditions aim at dependency preservation and lossless joining for relation schemas, respectively. The second condition ensures that the join operation for each relation \(P_i\) of \(P\) does not lose the instance information in \(R\). Accordingly, \(F_g\) updates the mappings associated with \(R\) to those of \(P_i\), \(1 \leq i \leq s\).

### 4.3.3 State Update

At the \(t\)-th step, the agent starts with the state \(s_t\) and greedily selects the action \(a_t\) with the maximum expected reward to generate a new state \(s_{t+1}\).

A Q-table is used to record the expected reward for each action-state pair, which is a \((N \times N)\)-dimensional table whose rows are states represented by all attributes associated with the relation schemas and columns are actions defined by \(A\), where \(N\) is the size of \(A\). At each step \(t\), the agent selects an action \(a_t\) and update Q-value based on the reward \(r_t\) for \(a_t\) and \(s_t\).

As for an action \(a_t\) and a state \(s_t\), the reward function is designed to motivate the agent not only to explore the optimal global relational schema, but also to weight the semantic diversity of answers and the query complexity for the given query set. Consider the answers of query set returned by the environment, we define the reward function to be negatively correlated with the query time and positively correlated with the number of crossing answers. The reward \(R(a_t, s_t)\) for \(a_t\) and \(s_t\) is computed as follows:

\[
R(a_t, s_t) = \eta \frac{d_t}{d_{t-1}} - 1 + (1 - \eta)(w_t - 1),
\]

where (i) \(d_t\) (resp. \(d_{t-1}\)) is the number of crossing answers at the \(t\)-th (resp. \((t - 1)\)-th) iteration; (ii) \(w_t\) is the query time performing the given query set in the RDBMS at the \(t\)-th iteration; and (iii) \(\eta\) is a weight parameter, \(0 \leq \eta \leq 1\).

The update of Q-value is based on the Bellman optimality equation, which is a weighted average of the current value and the new information, as introduced below:

\[
Q_{t+1}(s_{t+1}, a_t) = (1 - \beta) \cdot Q_t(s_{t+1}, a_t) + \beta \cdot \max_{a \in A} \left[ R(s_t, a_t) + \gamma \max_{a \in A} Q(s_{t+1}, a) \right],
\]

where (i) \(\beta \in [0, 1]\) is the learning rate that determines the ratio of accepting newly learned information; (ii) \(R(s_t, a_t)\) is the reward from \(s_t\) to \(s_{t+1}\) by taking the action \(a_t\); (iii) \(\gamma \in [0, 1]\) is a discount factor that determines the influence of future reward, where the smaller \(\gamma\) the more short-term benefits are considered; and (iv) \(\max_{a \in A} Q(s_{t+1}, a)\) is the maximum reward that can be obtained from state \(s_{t+1}\), which is weighted by the learning rate \(\beta\) and the discount factor \(\gamma\).

Algorithm 1 illustrates the process of state update and the details is described by the following example.

---

**Algorithm 1: LakeGrsg**

**Input:** a set of queries \(P\), an action space \(A\), the Q-table \(Q_t\), and an initial state \(s_t\);

**Output:** a new state \(s_{t+1}\).

1. \(m \leftarrow 0;/** an episode **/\)
2. Select an action \(a_t\);
3. while \(A\) is not empty do
4. Compute \(R(s_t, a_t)\) based on \(P\);
5. Obtain the next state \(s_{t+1}\) based on \(s_t\) and \(a_t\);
6. Compute \(Q_{t+1}(s_{t+1}, a_t)\) and update \(TQ[t, i]\);
7. if \(m < Q_{t+1}(s_{t+1}, a_{t-1})\) then
8. \(a_t \leftarrow a_t;\)
9. \(m \leftarrow Q_{t+1}(s_{t+1}, a_{t-1});\)
10. \(A \leftarrow A \setminus \{a_t\};\)
11. Update \(s_{t+1}\) based on \(a_t\) and \(s_t\);
12. return \(s_{t+1}\).
5 QUERY ANSWERING WITH INTEGRATION PARADIGM

In this section, we first present an overview of query answering pipeline, and then introduce how to attack the heterogeneity of the data lake to enhance the query semantics of \( Q \) based on the schema integration paradigm \( S \).

5.1 Query Answering Pipeline

Generally, distributed query optimizers\cite{1, 21, 39} handle a SQL query in the following steps: (1) parsing the input query string and creates an abstract syntax tree; (2) compiling the created syntax tree to generate the logical search space where contains all logical equivalent alternative plans to execute the query; and (3) enumerating all physical distributed execution plans of these logical plans and choosing one with the least estimated cost.

To perform query answering over \( D \), there are two problems should to be solved, as described below: (1) heterogeneity of data schemas, which may lead to incomplete logical plans and further produce incorrect answers; and (2) semantic heterogeneity of data instances, which will lead to incomplete query results computed by the physical plans.

Therefore, we present two key modifications to break the heterogeneity of schemas and instances in the data lake. Firstly, logical plan enrichment enriches the logical search space by bridging the schema gap between the query and the source schemas. Secondly, semantic joining builds semantic associations among data instances of different data sources.

5.2 Semantic Enhancement

To enhance the query semantics of \( Q \), we present our logical plan enrichment and semantic joining based on the schema integration paradigm \( S \) as follows.

**Logical plan enrichment.** We bridge the semantic gap between the query and the local relational schema based on the mapping function \( F_2 \). This is done by replacing the concept in the logical tree of \( Q \) as that of the local schemas.

For example, the class “Customer” in the query \( Q \) shown in Figure 2 should be projected into “User” in \( C_1 \) and “Customer” of \( C_3 \), as illustrated in Figure 3.

**Semantic joining.** After performing the distributed physical query plan over the naive database engines of \( D \), a semantic join operation is presented to progressively join these local results. We need to address the heterogeneous matches that refer to the same entity, and further deduce a relation schema for them in terms of the local schema \( C_3 \).

Given a set of tuples \( T \) with relation schema \( R(b_1, \ldots, b_m) \) in \( D_k \) and a subgraph \( G \) in \( D_l \), we identify a match \((u, t)\) for a tuple \( t \in T \) and vertex \( u \in G \) by checking whether they have the same or similar concepts and instance values. Recall that there is an edge from \( R_1[A_1] \) to \( R_2[B_1] \) in \( C_3 \) if there is a reference from the foreign key \( A_1 \) of \( R_1 \) to the primary key \( B_1 \) of \( R_2 \). Let \( F_2(u) \) represent the global concept of \( u \) in \( C_3 \). Therefore, \( u \) conceptually matches \( t \) if \( F_2(u) \) and an attribute \( b \) of \( t \) in \( C_3 \) are reachable. We then use the embeddings \( L(u) \) and \( L(b) \) to compute their instance similarity for the condition (2). If the similarity is larger than a threshold, we call that \( t \) and \( u \) can be semantically joined via the joining key \( t.b \).

As for the match \((u, t)\), we deduce a relation schema \( R_G \) with attributes \( (b_1, \ldots, b_m, b_{m+1}, \ldots, b_n) \), such that \( t \) can be enriched with additional attributes \( A' \) of \( u \) extracted from the subgraph \( G \), where \( A' = \{b_i: 1 \leq i \leq n\} \). Specifically, we first extract the relation schema \( R' \) with attributes \( A' \) for \( G \) based on \( C_1 \), and then semantically join \( R \) with \( R' \) via the joining key \( b \) to obtain \( R_G \), i.e., \( R_G = R \bowtie R' \). Thereafter, we populate \( R_G \) with the corresponding instance values to compute \( T \bowtie G \).

6 EVALUATION

Using real-life datasets, we experimentally evaluate LakeAns for its efficiency and scalability. Our highlights are described as follows:

(1) Our schema integration approach LakeGrsg outperforms its competitors in terms of query time, the number of crossing answers and convergence speed.

(2) LakeAns can find the answers cross multiple sources spending almost the same time with the methods performing local evaluation in a single data source.

(3) LakeAns can efficiently perform cross-source queries based on the integrated relational schema in LakeGrsg, and has better scalability than its competitors in terms of data size and number of sources.

6.1 Experimental Setting

**Datasets.** We use four real-life datasets shown in Table 1: (1) Unibench, a multi-model e-commerce benchmark\cite{52}, in which the customer and product information are modeled as two relational tables including 150,000 tuples. Order information and transaction relationships are represented by a JSON document and a graph, respectively. The JSON contains 142,257 objects, and graph has 9,949 vertices and 375,620 edges. (2) IMDB: movie dataset in\cite{10}. The relational data includes performing and rating information. The JSON stores film information and the graph is about cooperative information that indicates two actors have ever worked for the same movie and they are linked. (3) DrugBank, is extracted from DrugBank Central\cite{3} and ChEMBL database\cite{2}, which includes 8 tables to present drug, target, ChEMBL information, etc. The interaction data of drugs and targets is modeled as a graph. (4) DBLP: publication data consisting of bibliography records in computer science. The publication records are presented in relational data. A subset of the paper information is represented in JSON. We also construct a co-authorship graph where two authors are connected if they publish at least one paper together.

Every dataset derive a local schema for each data source and generates a global relational schema that is maintained in the mediator using LakeGrsg. SQL queries are formulated based on the global schema and submitted at the mediator.

**Query workload.** We first generate a set of SQL queries for each dataset by randomly selecting classes from the global relational schema. The queries have SELECT and FROM clauses. We then extend these queries by adding references or filter conditions into the WHERE clause that relate to the classes of the SELECT clause.
we denote the query set as \( Q_1, Q_2, Q_3, Q_4, Q_5 \) and \( Q_6 \), corresponding to 1J, 2J, 3J, 3J1S, 3J2S and 4J, respectively, where the query \( i/J \) has \( i \) join conditions and \( j \) filter operations in the WHERE clause. The default one is \( Q_5 \).

**Implementation.** We develop a prototype system of LakeAns where the mediator employs a SQL server to maintain the global relational schema, and MongoDB, Neo4j and SQL server as the local database for the JSON, graph and relational data, respectively. As we discuss in Section 3, our algorithm mainly performs on the mediator and benefits from the technology of these local databases. We conduct our experiments on CentOS Linux 5.4 equipped with 3 Intel(R) Xeon(R) Silver 4110 CPUs, 30 GB memory.

**Baselines.** Most relational query baselines are limited in their ability to perform cross-source querying because they are restricted to a single data model, such as structured relational data. To ensure fair comparisons, we modify several existing methods to accommodate cross-source querying over the data lake as described below. (1) Naive performs the input SQL query on each local relational schema of the data source and then returns the final answer by directly matching the returned local results with the same attributes. Naive is barely able to find those results across heterogeneous data sources and is mainly used to measure the semantic fusion capability of other methods. (2) InfoGather [48] considers indirect and direct matching, and exploits a holistic matching framework based on entity augmentation and attribute discovery to find the answer across multiple web tables. InfoGather cannot support the query answering over the data lake including semi-structured and unstructured data. To adapt it to heterogeneous data lake, we load the dataset into relational tables complying with the local relational schemas created in Section 4.2. (3) Snowflake [9] is a shared-data distributed architecture that provides SQL extensions to traverse semi-structured and unstructured data. Snowflake cannot support global schema-based computation and it bridge heterogeneous semantics between different data sources using schema discovery. Similar to InfoGather, Snowflake focuses only on the enhancements of query semantics but neglects the optimal joining order of intermediate results, which is constrained by the SQL extensions.

These baselines differ from our approach in that LakeAns requires to enhance query semantics over the data lake with multiple data models and heterogeneous query engines. To this end, LakeAns considers the global relational schema to bridge the schema heterogeneity of multiple data sources and the optimal join order of intermediate results to improve the query efficiency.

There is no baseline considers schema integration of heterogeneous data to efficiently support SQL queries across sources. We design the following baselines. (1) MultiStore [50] learns a relational schema to store multi-model data into RDBMS using reinforcement learning. It denotes all data as a set of two-column tables and aims to find the optimal join sequences to generate relational schema. MultiStore expects to obtain the schema with minimal query time but ignores the query semantics over the generated schema. (2) Two variants of LakeGrs are designed by modifying the agent of LakeGrs. LakeRGr is the agent removing merging and splitting actions. LakeRGr is the agent removing the splitting action.

**Metrics.** To evaluate the performances of schema integration quantitatively, we use the following aspects: (1) the query time (2) the relative coverage rate of crossing answers in the final results, and (3) the expected reward of the given SQL queries. The relative coverage rate is computed as \( Rate = |T_i - T_0|/T_0 \), where \( T_i \) and \( T_0 \) are the answer set at the \( i \)-th iteration and the initial state, respectively.

The evaluated metrics for query answering contain the query time and the relative coverage rate of crossing answer in terms of the answers computed by Naive.

**Parameter selection.** As for the LakeGrs algorithm, we search hyper-parameters in the following values: the learning rate \( \beta \) in \{0.0001, 0.0005, 0.001, 0.005, 0.01\}, discount factor \( \gamma \) in \{0, 0.2, 0.4, 0.6, 0.8, 1\}, reward weight parameters \( \eta \) in \{0, 0.2, 0.4, 0.6, 0.8, 1\}. The similarity thresholds in actions are set \( \lambda \) in \{0.2, 0.6, 0.8, 1\}, \( \alpha \) in \{0.5, 0.6, 0.7, 0.8, 0.9\} and \( \theta \) in \{0.5, 0.6, 0.7, 0.8, 0.9\} for relation names (i.e., classes), attribute names and pivot vectors of attribute domains, respectively. The selected setting is that \( \beta = 0.001, \gamma = 0.8, \eta = 0.6, \lambda = 0.6, \alpha = 0.8, \theta = 0.7\).

### 6.2 Experimental Results

We next report our findings.

**Exp-I: performance of schema integration.** To evaluate the performance of schema integration algorithm, we concern two aspects: (1) the changes in query time and coverage rate as the increasing of episodes; and (2) the expected reward with respect to the discount factor \( \gamma \) and weight parameter \( \eta \).

As reported in Figure 7(a), the query time of LakeGrs is always smaller than that of MultiStore especially when the episode number is large. The reason is that (i) MultiStore creates a relational schema by finding a sequence of joins, which results in some relation schema including much foreign keys to increase the query cost. (2) Considering the foreign key of schemas, LakeGrs defines merging and
To evaluate the impact on query time and coverage rate, we vary the source number from 3 to 7, which is done by partitioning the DBLP dataset into several fragments. As illustrated in Figure 12, LakeAns outperforms other methods in terms of scalability, yielding much smaller execution time than InfoGather and Snowflake, and greater coverage rate than Naive and InfoGather. With the increasing of source number, all execution times decrease and coverage rate improve. As the number of sources increases, all execution times decrease and the coverage rate slowly increases. Furthermore, LakeAns can obtain twice the coverage rate of the answers obtained by Naive and InfoGather, as reported in Figure 12(b). This is because that LakeAns finds crossing answers a schema-based approach that answers queries based on a global relational schema and the mappings from the global to the local. (ii) Snowflake utilizes SQL extensions to answer cross-source queries, which progressively traverses the bindings of attributes and entities. LakeAns benefits from an integrated schema and can effectively bridge the semantic associations between different data sources to perform cross-source querying.

Figure 9(b) illustrates the following two aspects. On the one hand, the coverage rate of LakeAns and Snowflake are larger than that of Naive and InfoGather. In most cases, the coverage rate of LakeAns is greater than or equal to that of the other methods, as we can see from the IMDB and DrugBank datasets shown in Figures 10(b) and 11(b). On the other hand, the coverage rate of LakeAns, InfoGather and Snowflake all increase with the increasing of join condition number in the query, while LakeAns significantly has a much larger upward trend than that of Naive and InfoGather in different datasets.

Totally, LakeAns spends almost the same time with Naive that performs local evaluation in a single data source to find the answers with greater coverage rate than other methods.

Exp-3: impact of source numbers. To evaluate the scalability, we vary the source number from 3 to 7, which is done by partitioning the DBLP dataset into several fragments.

As illustrated in Figure 12, LakeAns outperforms other methods in terms of scalability, yielding much smaller execution time than InfoGather and Snowflake, and greater coverage rate than Naive and InfoGather. With the increasing of source number, all execution times decrease and coverage rate improve. As the number of sources increases, all execution times decrease and the coverage rate slowly increases. Furthermore, LakeAns can obtain twice the coverage rate of the answers obtained by Naive and InfoGather, as reported in Figure 12(b). This is because that LakeAns finds crossing answers...
in the data lake as much as possible by leveraging an integrated relational schema.

All the above experimental results justify that our LakeAns framework is efficient and effective. It scales well on all metrics and has high coverage rate of answers across multiple data sources. Moreover, the integrated relational schema learned by LakeGrsg can effectively and efficiently support cross-source querying and enhance the answer semantics for a heterogeneous data lake.

7 RELATED WORK

We categorize the related work as follows.

Schema Matching. Some problems in data integration, such as entity resolution [16, 31], ontology alignment [28, 35], schema matching [17, 40] and dataset linking [19] have attracted significant interests in recent years.

The most related work to our schema integration is schema matching. Given a set of datasets and their schemas, schema matching is the problem of discovering potential correspondences between concepts of different datasets and it is one of the most important prerequisite steps for analyzing heterogeneous data collections [40]. State-of-the-art schema matching algorithms use simple schema- or instance-based similarity measures to struggle with finding matches beyond the trivial cases. Semantics-based algorithms require the use of domain-specific knowledge encoded in a knowledge graph or an ontology. For example, Data Tamer [41] match attributes using a variety of similarity measures and algorithms called experts. The Data Civilizer system [10] uses a linkage graph to support data discovery, while Aurum [18] builds knowledge graphs where different datasets are correlated with respect to their content or schema. All of these methods rely on similarity computation, e.g. Jaccard Similarity, value distribution. [53] and [19] build relationships between concepts of different databases using cluster-based matching algorithm and a given ontology. As a result, schema matching depends on external knowledge [19] such as domain-specific ontologies, and still remains a largely manual process.

These efforts differ from ours in that we generate a global relational schema for heterogeneous data sources by using a reinforcement learning method. This does not require external knowledge, but rather interaction with a relational database to find the schema with maximum expected reward.

Relational Schema Mapping. There has been a growing interest in mapping various data types into relational database to be able to reuse mature robust relational database technology. The choice of relational schema design plays a crucial role for efficiency. Different designs not only imply different kinds of physical data partitions, but also different translations to SQL operations. Our global relational schema generation relates closely to store the multi-model data to relational databases [50], but it is not designed for supporting the unified SQL query. Additionally, relational mapping has been studied in semi-structured data [11, 12, 44], RDF [4, 37] and uncertain data [15, 30].

The most straightforward of semi-structured to relational mapping earlier efforts provide generic mapping rules that do not necessitate upfront analysis of the input dataset [11]. Some other works present a mapping strategy based on structural analysis of the input dataset [11, 12, 49]. They design schemas for XML datasets by analyzing a graph of the DTD elements present in the input data and a set of heuristics is used to dictate whether an element should be materialized as its own table or linked within a parent element’s table. Investigations of various schema designs form RDF data to relational have been started by [4, 36]. Recently, for a single-node system, the analysis provided in [37] gives arguments for using an emergent schema. [6] obtains the relational schemata using Apache Hadoop as the distributed processing platform and mapping SPARQL queries into Spark SQL. There are two major and complementary approaches to dealing with uncertainty in data [15]. In general, probability theory is quantitative with more precise outcomes, but these come at the price of acquiring actual probabilities and high computational complexities in managing them [42]. However, if data uncertainty does come with meaningful probability values, a probabilistic model is more appropriate if it can be managed with feasible resources. In recent work, a survey of practical methods for constructing possibility distributions was given [15]. Research on probabilistic databases has focused on queries [42]. Typical is the desire to extend trusted relational technology to handle uncertainty.

Data Lake Discovery. [33] identifies the challenges and opportunities in a data lake. Aurum [18] discovers syntactic relationships between datasets in a graph data structure and supports keyword search and similar content search. [34] defines the table union search problem on open data, which proposes value set, class semantic and embedding similarity to determine the attribute union-ability. Skluma [7] extracts diverse embedded metadata from files based on a probabilistic pipeline and allows the topic-based discovery. Constance [23] exploits the semantic annotations of data sources to enriches metadata, which can be accessed by a template-based query. Some approaches also navigate dataset based on the linkage graphs [10] or version graphs [25]. Recently, a new organization structure based on the Markov probabilistic model is proposed such that users can navigate a data lake more effectively [32]. Josie [55] finds some datasets in the data lake that can be joined with a given table, which is transformed into an overlap set similarity problem. Juneau [54] finds additional tabular or nested data for training or validation from computational notebooks (e.g., Jupyter), workflows and cells. More related works for knowledge discovery in a data lake can be found in surveys [24, 33].

These works have different goals with our work, and we aim at exploring complete answers for SQL queries across multiple heterogeneous data sources.

8 CONCLUSION

In this paper, we study query answering over a heterogeneous data lake, which aims to enhance answer semantics of cross-source relational querying. To this end, a novel schema-based framework LakeAns is proposed. LakeAns integrates local schemas to create a global relational schema based on a reinforcement learning approach to effectively support the query answering and to find answers across multiple data sources. Our experimental evaluation verify that this framework is promising for finding more answers cross multiple data sources with minimum execution time. In future works, we will extend our work to handle unstructured multimedia data in a heterogeneous data lake.