# A Vision for SQL-Based Relational Deep Learning

Fahim Shahriar Khan
University of Texas at Arlington
fsk2739@mavs.uta.edu

Ashraf Aboulnaga
University of Texas at Arlington
ashraf.aboulnaga@uta.edu

## ABSTRACT

Much of the world's structured enterprise data resides in relational databases. Therefore, it is important to design machine learning models that are tailored to the specific characteristics of tabular data in relational databases. One such machine learning model that has recently been proposed is Relational Deep Learning (RDL). In RDL, a relational database is modeled as a heterogeneous graph. Each row in a table of the database represents a node in this graph, and each foreign-key link between two rows represents an edge. Graph Neural Network (GNN) techniques are used to train a deep learning model on this graph that can be used for various prediction tasks on the relational database. In this paper, we propose that much of the computation required for training and inference in RDL can be done in SQL, inside the database management system (DBMS). Using SQL is good for performance since it minimizes data movement from the DBMS to the deep learning software and can utilize the bulk query processing capabilities of the DBMS. It is also good for expressiveness, since the full power of SQL can be used in different stages of the training pipeline, such as sampling training data. We discuss different possibilities for integration between the SQL DBMS and the deep learning software in an RDL pipeline. We also present our preliminary efforts implementing these ideas, and show that using SQL enables much simpler and more powerful sampling of training data for the GNN compared to exporting the database as a graph and performing the sampling in PyTorch or a similar system. We also outline future research directions for this nascent and promising research area.

**VLDB Workshop Reference Format:**
Fahim Shahriar Khan and Ashraf Aboulnaga. A Vision for SQL-Based Relational Deep Learning. VLDB 2025 Workshop: Tabular Data Analysis (TaDA).

## 1 INTRODUCTION

Relational databases commonly store tabular data using well-defined schemas and foreign-key links that connect different tables. Traditionally, extracting predictive signals from such data for machine learning models required laborious manual feature engineering: domain experts join tables and craft summary features to flatten the data into a single table for modeling. This process is time-consuming, error-prone, and often suboptimal, as it can discard important relational structure [26]. Recent advances in deep learning offer a compelling alternative: Graph Neural Networks (GNNs) can directly learn from the inherent structure of relational data by representing the database as a graph [7]. In this paradigm, each row in each table of the database becomes a node in the graph and each foreign-key relationship becomes an edge, yielding a heterogeneous graph that represents the entire database. By training a GNN on this graph, one can automatically learn rich embeddings for each entity that integrate information from related records. This approach, recently termed *Relational Deep Learning (RDL)* [10], eliminates the need for manual feature engineering and results in powerful and accurate machine learning models for various prediction tasks.

However, applying GNNs to relational databases in the RDL paradigm is far from straightforward. The graph induced by a database schema is heterogeneous, containing multiple node types (one per table) and edge types (one per foreign-key relationship). Thus, starting from any node in the graph, one can define diverse multi-hop paths that vary in the types of nodes connected and edges followed. It is not obvious which of these multi-hop paths are most relevant for a given prediction task on a given database. Typical GNN approaches are *message passing* neural networks. They work by *collecting and aggregating messages from neighbors* of each node in the training data. To reduce the cost of training, message passing is sometimes done on a *sample* of the neighbors of a node. These GNN approaches do not distinguish between node types and edge types and treat all neighbors and paths the same [13], which can introduce noise and dilute important signals in an RDL setting. For example, if we are using RDL to predict customer churn in an e-commerce database, the graph may contain nodes representing rows from a customer table linked to nodes representing purchases, products, and reviews (from the respective tables). Without guidance, a GNN would give equal treatment during training to all types of nodes that neighbor a customer node, even though recent purchase nodes could be far more predictive of churn than older interactions through, say, review nodes. The key point is that in a large, multi-table graph with diverse link types, the GNN needs to focus on the proper subset of relationships for sampling and message passing during training. Researchers have begun to tackle this issue by identifying the most relevant multi-hop paths for a prediction task, often described as *meta-paths* in heterogeneous network analysis. In RDL, this can be done by leveraging semantic information from the relational database schema. We present more background about GNNs in Section 2.

Given the need for sophisticated GNN training, a typical RDL training pipeline proceeds as follows: (1) Convert the entire relational database into a heterogeneous graph and store it in a format understandable by PyTorch [22] or a similar deep learning system [1]. (2) Sample the graph to extract training data, possibly using meta-path rules to focus on certain multi-hop paths. (3) Train the GNN on the extracted data, possibly using a library such as PyTorch Geometric (PyG) [11]. (4) Repeat from Step 2 until convergence.

In this paper, we propose that ***much of the work required for RDL training (and inference) can be done in SQL, inside a relational database management system (DBMS)***. Using SQL has


Proceedings of the VLDB Endowment. ISSN 2150-8097.

several advantages. First, it can minimize data movement from the DBMS to the deep learning system since some of the data filtering and aggregation required for creating training data can be done inside the DBMS. Second, using SQL enables the full expressive power of SQL to be used in choosing and aggregating the nodes and edges to include when sampling the graph during training. Third, it may be possible to completely eliminate the need to convert the relational database into a graph. While modeling the database as a heterogeneous graph is useful at the conceptual level, this graph may remain a virtual construct that is never materialized. Fourth, by carefully dividing the work of training the GNN (and inference) between the DBMS and the deep learning system, it may be possible to use the full query processing power of the DBMS, for example, optimized query plans, joins on data bigger than memory, bulk operators, and pipelined query processing. We present more details of our vision for SQL-based RDL in Section 3, and we discuss different possibilities for integration between the SQL DBMS and the deep learning system.

We have taken preliminary steps in implementing a system that realizes this vision, which we call *SQL-GNN* (Section 4). The current focus of SQL-GNN is using SQL queries for schema-guided sampling of the database to create training data for the GNN. The core idea is to leverage SQL queries to define the neighborhood that each node in the training data will receive messages and aggregate information from. Instead of relying on uniform or ad-hoc neighbor selection, we allow the user to specify – in SQL – which connected records should be considered as a node's neighbors during GNN message passing. For example, a SQL query can select all products purchased by a customer in the last six months as that customer's neighborhood, reflecting a specific meta-path (Customer⟶Order⟶Product) with a temporal filter. Our SQL-defined neighborhood sampling provides several benefits: (1) It is inherently schema-aware – the query explicitly navigates the relational schema, so the GNN only traverses meaningful joins (e.g., customer⟶buys⟶product) as opposed to arbitrary links). (2) It is task-specific – one can tailor the SQL conditions to focus on relationships relevant to the prediction problem (e.g., restricting purchases by recency or price). (3) It can exploit the performance of database optimizations – by delegating neighbor selection to the DBMS, we use mature query optimizers and indexing to retrieve neighborhoods efficiently. By integrating declarative querying with GNN training, we obtain a flexible framework for relational deep learning that maintains the full expressive power of SQL while harnessing end-to-end graph learning.

We conclude the paper and outline a future research agenda for this nascent and important research area in Section 6.

# 2 BACKGROUND

## 2.1 Relational Deep Learning

A relational database can naturally be transformed into a graph structure for deep learning, where each tuple (row) becomes a node and each foreign-key relationship corresponds to an edge. The result is a heterogeneous graph where the type of a node is determined by its table name (entity type), and the type of an edge is determined by the foreign-key relationship it represents according to the schema (relationship type). This approach was formalized by recent work on Relational Deep Learning (RDL) [10], which demonstrates that converting a database into a graph and applying GNNs end-to-end can eliminate the need for manual joins and handcrafted features. By preserving the complete relational structure, the GNN has enough information and degrees of freedom to learn which attributes and connections are important, rather than relying on a data scientist's intuition. RDL is gaining prominence not only in academic research but also in industry. Companies are pioneering efforts to apply RDL on enterprise data, for example, by developing Relational Foundation Models [21].

Many real-world domains yield relational graphs. For example, an e-commerce database may have tables for customers, orders, and products; after conversion, we get a graph where a customer node connects to an order node (via a "customer-placed-order" edge), which connects to product nodes (via "order-contains-product" edges). A GNN operating on this graph can aggregate information from purchased products into the customer's representation, enabling predictions of customer behavior (e.g., churn or lifetime value) using signals that would be lost in a flattened table. Examples like this illustrate how multiple domains can benefit from representing the multi-table structure in a graph format, laying the groundwork for graph-based representation learning on relational data. The key to learning an effective representation is to use an appropriate GNN technique, and we turn to this topic next.

## 2.2 Message Passing Graph Neural Networks on Heterogeneous Graphs

*2.2.1 Message Passing GNNs and GraphSAGE.* Graph neural networks (GNNs) learn node embeddings through iterative message passing between neighbors [19]. At each layer of a GNN, every node collects and aggregates feature information from its adjacent nodes and updates its own representation; after $k$ layers, a node's embedding encodes information from all nodes within $k$ hops. This framework is powerful because it lets the model exploit complex relationships in the data, often outperforming traditional methods that rely on engineered features.

In practice, one must typically sample or restrict neighbors for aggregation to improve performance and control computational cost. For instance, GraphSAGE [13], which is a popular and flexible GNN approach, uses a strategy of sampling a fixed number of neighbors uniformly at random and learns weight matrices that transform and aggregate information from these sampled neighbors. We use a technique that builds on GraphSAGE in this paper.

*2.2.2 Adapting GraphSAGE to Heterogeneous Graphs.* GraphSAGE was developed for homogeneous graphs, and it makes no provision for nodes and edges of different types. Several frameworks extend GraphSAGE to handle heterogeneous graphs by grouping neighbors by type and applying type-specific transformations during aggregation [5]. Two examples are described next.

**HinSAGE [6]:** This is an extension of GraphSAGE for heterogeneous information networks that was introduced to support tasks like link prediction and node classification in multi-relational graphs. HinSAGE explicitly accounts for different neighbor types and relation types by maintaining separate weight matrices for each relation when aggregating neighbors. In practice, a HinSAGE

layer groups a node's neighbors by their edge type (or, equivalently, by the neighbor node's type) and computes an aggregated message from each group [5]. These type-specific neighbor representations are then combined (e.g., concatenated or summed) with the node's own features to produce the updated embedding. By assigning distinct transformation weights to each neighbor group, HinSAGE preserves schema information (what type the neighbor is and how it is connected) during message passing. The original implementations of HinSAGE have primarily used simple aggregators (such as mean pooling) per neighbor type. However, in principle, any GraphSAGE aggregator (max-pooling, LSTM, attention, etc.) can be applied within each type group.

**HetGNN [31]:** This is a hierarchical GNN designed to capture structural and attribute heterogeneity in graphs. HetGNN operates through a two-step aggregation process. First, it samples a fixed-size set of heterogeneous neighbors for each node using random walks with restart and groups them by type. Then, each group's neighbors are encoded separately using a type-specific bi-directional LSTM followed by mean pooling. Subsequently, HetGNN performs intra-type aggregation, combining embeddings of same-type neighbors into one vector per type using another Bi-LSTM and averaging. Finally, an attention mechanism aggregates these type-specific vectors, weighting their relative importance when updating the target node. This hierarchical approach (neighbors⟶type, then type⟶target node) enables HetGNN to effectively capture both within-type and cross-type interactions, albeit with increased model complexity and training time due to the multiple LSTMs and attention mechanisms.

Such schema-agnostic sampling treats all neighbor types equally, which means the GNN could end up focusing on many irrelevant or less informative neighbors. If a particular node is incident to an overwhelmingly large number of edges (e.g., a popular product linked to thousands of orders), it will dominate the messages a node receives, even if those messages are not the most useful for the task. Conversely, important but subtler patterns risk being drowned out by the sheer volume of other neighbor information.

*2.2.3 Schema-Aware Message Passing.* Recent research has proposed techniques to incorporate schema semantics into GNNs. One approach is to use predefined, expert-selected meta-paths (specific sequences of node/edge types) to guide the message passing. For example, metapath2vec generates embeddings by running random walks that follow a given meta-path in the heterogeneous graph, thereby capturing semantic connectivity along that path [8]. More recent models like Graph Transformer Networks (GTNs) can even learn which meta-paths are useful, dynamically generating composite relations between node types and building new connectivity accordingly [29]. By focusing on task-relevant connections, these methods have shown improved performance on heterogeneous graph benchmarks [9, 15, 16]. However, both approaches have limitations: metapath2vec and similar methods require significant domain knowledge to choose meta-paths, while GTNs add considerable complexity to the model and slow down training. There is still a need for a simpler, declarative approach to leverage relevant relational structure in GNN training.

Our approach in this paper addresses these issues by leveraging the relational schema in a more direct, declarative way through SQL-based neighbor sampling. By using SQL queries to specify which neighbors to retrieve for each node, we achieve schema-aware (and even task-specific) sampling that is both flexible and efficient. The SQL query can naturally encode meta-paths via join operations spanning multiple tables, and can apply filters relevant to the task (e.g., time windows or attribute conditions), all within the database engine. In essence, we shift the graph construction logic into the query layer, ensuring that the GNN only receives the most pertinent neighbor information for learning.

*2.2.4 HeteroGraphSAGE.* Our work uses a technique called `HeteroGraphSAGE`, introduced in [24] (implementation available at [12]), which is a lightweight schema-aware GNN model. In essence, `HeteroGraphSAGE` generalizes GraphSAGE to heterogeneous graphs by defining separate weights for each edge type and aggregating messages from each relationship independently, much like HinSAGE [6]. Formally, let $h_v^{(l)}$ be the embedding of node $v$ at layer $l$ of the GNN, and let $\mathcal{R}$ be the set of relationship types in the graph (each relationship $r \in \mathcal{R}$ connects source nodes of a certain type to target nodes of a certain type). We denote by $N_r(v)$ the set of neighbors of node $v$ via relationship $r$. A `HeteroGraphSAGE` layer updates node $v$ as follows. For each relationship $r$ incident on $v$, we aggregate the messages from $v$'s $r$-neighbors (e.g., by taking the mean of their embeddings from the previous GNN layer, i.e., layer $l-1$) and apply a relationship-specific linear transformation followed by a nonlinear activation function (e.g., ReLU).

This `HeteroGraphSAGE` module is well-suited for relational data, especially when paired with our SQL-defined neighbor sampling. First, it is inherently schema-aware as it maintains separate parameters for each relationship, so the model can learn the relative importance and distinct patterns of each edge type (unlike a homogeneous GNN that would mix the parameters together). Second, it offers flexibility in leveraging domain knowledge – one can easily include or exclude certain relationships or apply custom filters simply by changing the SQL neighbor query, without altering the model architecture. In effect, `HeteroGraphSAGE` acts as a general inductive learning engine that can take any subset of the relational graph as input. Third, it remains efficient and relatively simple. Compared to advanced architectures like GTN or HetGNN (which add extra layers for learning attention or content encoders), `HeteroGraphSAGE` has a straightforward feed-forward update and fewer trainable components. This simplicity means faster training and easier optimization, while still capturing heterogeneity through its separate relationship-specific weights. Finally, `HeteroGraphSAGE` is built on the inductive GraphSAGE framework [5], so it can naturally handle new nodes or dynamically evolving graphs, a crucial property for many relational databases where new nodes in the graph (customers, orders, products, etc.) arrive over time.

In summary, by combining `HeteroGraphSAGE` with SQL-based neighbor sampling, our framework takes advantage of the relational schema to focus the GNN on the most relevant connections, without incurring the overhead of more complex techniques. This approach strikes a desirable balance as it uses schema information to guide message passing while avoiding manual meta-path engineering and keeping the model tractable. For these reasons, we adopt `HeteroGraphSAGE` as the backbone GNN in our proposed framework, enabling effective relational deep learning on heterogeneous graphs derived from standard relational databases.

## 2.3 The RelBench Relational Deep Learning Benchmark

The recently introduced RelBench benchmark [24] provides a suite of realistic multi-table databases and associated prediction tasks to be performed using RDL. RelBench includes an e-commerce relational dataset called `rel-amazon`, derived from Amazon's book review data. We use this dataset and the associated prediction tasks as examples and for the experiments in this paper. As shown in Figure 1, this database consists of three tables connected by foreign-key relationships (`customer, product, and review`) with over 24 million rows in total. The tables include rich information such as product attributes (`e.g., category and price`) and review details (`e.g., rating and timestamp`).

RelBench defines prediction tasks on `rel-amazon` at the customer and item level. For example, the user lifetime value prediction task (`user-ltv`) requires forecasting the future monetary value of a customer's purchases based on their reviewing activities before a certain timestamp. That is, the training data consists of all the reviews and the associated products and customers prior to the timestamp, and the prediction variable is the total price of all products reviewed by a given customer in the subsequent three months. The item lifetime value prediction task (`item-ltv`) is defined analogously: predict the total dollar value of purchases a given item receives in the next three months, using all available data before the timestamp. RelBench also includes `user-churn` and `item-churn` tasks, which involve predicting whether a customer or product will become inactive (i.e., receive no reviews) in the next three months.

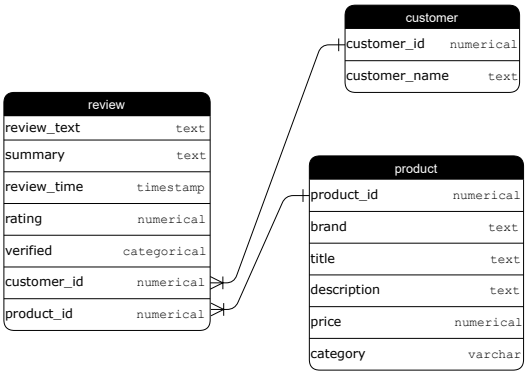


**Figure 1: Schema of the `rel-amazon` database**

# 3 SQL-BASED RELATIONAL DEEP LEARNING

The RDL approach of representing a relational database as a heterogeneous graph and using GNNs on this graph is undoubtedly powerful and promising. However, this conceptual approach does not necessitate exporting the relational database from the DBMS to a system like PyTorch as a graph and ignoring the power of SQL and relational database systems. In this paper, we propose ***doing much of the work required for RDL training and inference in SQL and using the full power of the DBMS***. We described the advantages of this approach in Section 1, and here we elaborate on specific ways that SQL and a DBMS can be used. In particular, we discuss (1) using SQL for sampling the training data, (2) doing RDL on an entirely virtual graph, and (3) moving part of the GNN training inside the DBMS. Among these, (1) is implemented in our system, as detailed in Section 4, while (2) and (3) are proposed as future research directions. We do not claim that this is a comprehensive list of possibilities, but rather specific suggestions for exploring this novel and promising research area.

## 3.1 SQL Sampling to Create Training Data

The RDL approach lets GNNs exploit the rich relational structure of multi-table data instead of flattening everything into one table. However, a major challenge in this setting is to generate the training data for the GNN. GNNs are trained on subgraphs sampled from the input graph. There is a node of interest, for example, the customer node in the `user-ltv` task, where the goal is to predict the lifetime value of a customer. This node is included in the training subgraph, as well as a sample of its one-hop neighbors, two-hop neighbors, and so on up to $k$-hop neighbors. The value of $k$ is typically small, for example, $k = 2$. One such subgraph is created for each node of interest in the training data, and these subgraphs are used in minibatches to train the GNN.

Traditional GNN training pipelines, for example, using PyTorch Geometric (PyG) [11] or DGL [27], perform neighborhood sampling in Python from a graph stored in memory or on disk. In the RDL setting, one must extract the entire graph (i.e., all relevant rows as nodes and key-foreign key links as edges) to files or memory, then use Python to create the training subgraphs by starting from the node of interest and randomly sampling neighbors [20]. For example, GraphSAGE-style sampling selects a few random neighbors per node at each GNN layer to limit the "neighborhood explosion" in large graphs. In this scenario, SQL is likely used to extract the graph from the DBMS, but sampling and training happen entirely outside the DBMS. While effective for scalability [3], this conventional approach is agnostic to the database schema and ignores any domain-specific information, treating the graph as a generic heterogeneous graph [20]. Sampling the subgraphs for training is done outside the DBMS, so it does not take advantage of the DBMS query processing capabilities. Furthermore, the sampling cannot be tailored to schema-derived constraints.

The most basic way to use SQL in RDL is to *sample the subgraphs for training the GNN using SQL*. The logic for creating each subgraph is expressed as a SQL query (or a sequence of queries) that runs efficiently in the DBMS. Edge traversals are easily expressed as foreign-key joins, and so are meta-paths that traverse multiple edges. Additional processing on the data when creating the subgraph, such as filtering or aggregation, can also be specified in the SQL query. Under this scheme, the tables in the database still have to be exported as a graph that is read by a system like PyTorch and used for GNN training. SQL is used for sampling the subgraphs used for training, but other aspects of GNN training, such as message passing between neighbors, loss minimization, and updating weights, all have to be done in PyTorch or a similar system. These activities require access to the graph.

Next, we elaborate on the benefits of using SQL for sampling.

**Schema-Aware Sampling:** The database schema encodes semantic information that can be very useful in improving the usefulness of the sampled subgraphs for GNN training. Because following foreign-keys is expressed declaratively as joins in a SQL query, one can naturally extend this SQL query to express domain-specific neighbor selections. For instance, one can sample not just any random neighbors but neighbors that follow a specific meta-path known to be relevant for the prediction task [18]. This is known to greatly improve accuracy in heterogeneous graphs. Prior work on heterogeneous GNNs often defines meta-paths or relation types to guide message passing (e.g., R-GCN for multi-relational graphs [7, 25, 30] or HAN which attends over meta-path neighbors [28]). With SQL, these meta-paths become simple join queries that retrieve the desired neighbor set declaratively, without writing custom code to traverse a graph. This declarative approach makes it easy to incorporate rich semantic context by simply adding filtering conditions (and potentially extra joins) to the WHERE clause of the SQL query. Sampled neighbors can be constrained to those matching specific patterns or attributes (e.g., "reviews of electronics products by users who also reviewed gardening products"). We show examples of these queries later in the paper.

**Temporal and Constraint-Based Sampling:** Real-world relational data often requires sampling strategies that respect both temporal and constraint-based conditions to construct meaningful subgraphs GNN training. Temporal sampling ensures causality by preventing future data from leaking into training, for example, adding `WHERE review_date < prediction_date` in a SQL query can guarantee that only past interactions are considered when predicting future outcomes. However, constraint-based sampling provides a more general and often more impactful mechanism by allowing subgraph selection based on arbitrary attributes such as user locale, product category, or rating value. SQL excels at this by enabling expressive and efficient filters like `user_locale = 'en-US' or rating_value >= 4`, which would otherwise require verbose, hard-coded logic in Python-based samplers. These constraints can reflect domain knowledge, enforce fairness, or tailor samples to specific tasks, making the sampling process not only more flexible but also more interpretable. By combining temporal awareness with rich, declarative constraint handling, SQL-based sampling offers a much more powerful and scalable alternative to traditional GNN APIs.

**Leveraging Decades of Optimization:** Relational database systems implement optimizations that are backed by decades of research into storage, indexing, and query processing. By formulating subgraph sampling as SQL queries, we automatically tap into these optimizations. The DBMS query optimizer will choose efficient join orders, use indexes to speed up link traversal, and push down predicates (filters) to minimize I/O. In practice, even complex multi-hop graph traversals translate to multi-table joins that can be executed efficiently at scale by modern database systems. Pushing sampling into SQL also means minimal data movement. Instead of transferring entire adjacency lists to Python, the database can send back just the sampled subgraph (often orders of magnitude smaller). Recent work on graph databases and GNNs confirms that using the query engine for neighbor retrieval can significantly reduce memory overhead and speed up training [20].

In summary, SQL-based sampling for GNNs on relational data enables schema-aware, constraint-aware neighborhood selection and taps into the power of database engines for efficiency and scalability. This approach bridges the two worlds of database systems and machine learning, which has always been recognized as a promising direction [14]. Several works have begun to explore this synergy by integrating GNN training with graph databases and introducing new frameworks for relational GNNs that respect database structure [20]. Our approach contributes to this emerging area by proposing to use vanilla SQL (on systems like DuckDB or PostgreSQL) as the interface for graph sampling. In the next subsection, we illustrate concretely how one can express common graph neighborhood sampling patterns as SQL queries.

*3.1.1 SQL-Driven Neighborhood Sampling Examples.* Consider the `rel-amazon` database of the RelBench benchmark [24]. This database has three tables: `Customer`, `Review`, and `Product`. In the `user-ltv` task, which is a node regression task, we want to sample subgraphs around a particular customer since we are trying to learn customer node embeddings to predict the total value of a customer's future purchases. From the schema in Figure 1, we see that the one-hop neighbors of a customer are all the reviews by this customer. We can use the query shown in Listing 1 to sample a random subset of these neighbors. The query uses a parameterized neighbor count (`max_hop1`) and respects a timestamp cutoff (only reviews before a given prediction date). This SQL query finds up to `max_hop1` reviews written by a given customer (`seed_customer`) before a specified cutoff date. The `ORDER BY RANDOM() LIMIT :max_hop1` clause implements uniform random sampling of the customer's reviews. In a standard GNN library, this corresponds to selecting a random subset of the customer's one-hop neighbors in the review nodes. By expressing sampling in SQL, we can easily modify the WHERE clause to impose other constraints – for example, we can add `AND r.rating >= 4` if we only want to sample the customer's positive experiences.

```sql
SELECT r.review_id, r.product_id
FROM Review AS r
WHERE r.customer_id = :seed_customer_id
  AND r.review_timestamp < :cutoff_timestamp
ORDER BY RANDOM()
LIMIT :max_hop1;
```

**Listing 1: One-hop randomized neighborhood sampling for the `customer-ltv` task (customer⟶review)**

Next, we show how to extend the one-hop Customer-Review sampling query to sample an entire three-hop subgraph around the seed customer in a single SQL query. One way to capture a three-hop neighborhood around the seed customer in one SQL query is to leverage Common Table Expressions (CTEs). The SQL query for three-hop sampling is shown in Listing 2 and works as follows:

- **Hop 1 (Customer⟶Review)**: Select a random sample (up to `max_hop1` rows) of reviews written by the seed customer (`customer_id = :seed_customer_id`) before a given cutoff timestamp (`cutoff`). These are the seed customer's own reviews (one-hop neighbors of the seed customer).
- **Hop 2 (Review⟶Product):** From the Hop 1 results, collect the unique product IDs of those reviews. This yields the

set of products that the seed customer has reviewed. Given the schema of `rel-amazon`, each review always points to only one product, so `max_hop2 = 1` in this example.

- **Hop 3 (Product⟶Review)**: For each product from Hop 2, retrieve other customers' reviews of that product (excluding the seed customer's own review) from before the cutoff. We sample up to `max_hop3` of these peer reviews per product, capturing the experiences of other users on the same items.

```
WITH
hop1 AS (
  SELECT review_id, product_id
  FROM reviews AS r
  WHERE r.customer_id = :seed_customer_id
    AND r.review_timestamp < :cutoff_timestamp
  ORDER BY random()
  LIMIT :max_hop1
  ),
hop2 AS (
  SELECT DISTINCT product_id
  FROM hop1
  ),
hop3 AS (
  SELECT r2.review_id
  FROM reviews AS r2
  JOIN hop2 AS h2
    ON r2.product_id = h2.product_id
 WHERE r2.customer_id != :seed_customer_id
   AND r2.review_timestamp < :cutoff_timestamp
  ORDER BY random()
  LIMIT :max_hop3
 )
SELECT review_id, product_id FROM hop1
UNION
SELECT NULL, product_id FROM hop2
UNION
SELECT review_id, NULL FROM hop3;
```

**Listing 2: Three-hop neighbor sampling query using SQL CTEs (customer⟶review⟶product⟶review).**

In our SQL query, each CTE builds on the previous one: `hop1` yields the seed's review IDs and products; `hop2` uses those results to get distinct products; `hop3` then finds other reviews for those products. The final `SELECT` combines these results, as shown in Listing 2 (`NULL` is used to fill in missing values so that the outputs from hop1, hop2, and hop3 all have the same columns and can be combined using `UNION`). By chaining these hops, the query constructs a focused three-hop subgraph centered on the seed customer. Notably, the inclusion of the third hop (other reviews of the products) provides richer relational context beyond the seed customer's immediate actions. The seed customer becomes indirectly connected to peer customers who reviewed the same products via the peer customer's reviews of these shared products. In terms of GNN message passing, this means information can flow from those other customers' reviews through the product nodes back to the seed customer. Such connectivity greatly enriches the seed node's neighborhood with task-relevant signals, for example, capturing community opinions or product popularity. This can enhance the learned representation for predicting a customer's lifetime value. Figure 2 illustrates this three-hop structure, helping illustrate how the neighborhood is expanded across relational links. In summary, this single SQL query efficiently captures a three-hop neighborhood that encodes a meaningful relational structure. In a straightforward way, this query extends the one-hop sampling approach to a subgraph that supports more expressive GNN-based modeling.

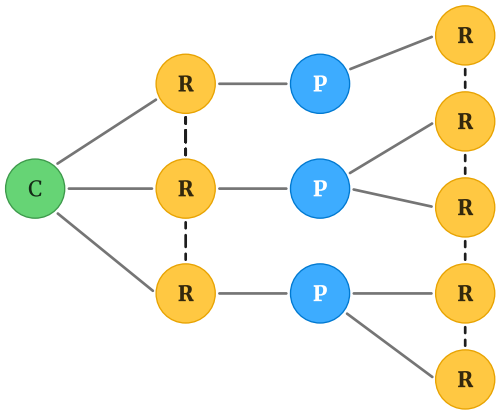


**Figure 2: Three-hop neighborhood of a seed customer node. C represents the customer node, R represents review nodes, and P represents product nodes.**

## 3.2 Representing the Database as a Virtual Graph

Using SQL for subgraph sampling helps improve GNN training in RDL, but does not eliminate the need for materializing the graph representing the relational database. The next step in SQL-based and DBMS-powered RDL is to eliminate this graph. Conceptually, RDL would still think of the database in terms of a graph, but this graph can be a purely virtual construct.

The GNN training would still happen in a machine learning system like PyTorch, but PyTorch would work on data structures that are created and managed by the DBMS, leveraging as much as possible the bulk processing capabilities of the DBMS. This approach would reduce data movement between the DBMS and Python, since the graph representing the database would be virtual. Making the graph virtual requires careful design choices and careful engineering. The information needed in GNN training, such as node ids, graph connectivity, and neural network weights and biases, must be carefully split between the DBMS and Python and orchestrated to ensure correctness and efficiency. Since the graph is virtual, it may be necessary to implement some of the GNN training operations without using a graph-aware library such as PyG, but such a decision is part of the research and engineering required to achieve the goal of GNN training on a virtual graph.

## 3.3 GNN Training in the DBMS

Beyond SQL-based sampling and GNN training on a virtual graph, it should be possible to improve RDL by pulling part of the GNN

training inside the DBMS. Training ML models inside the DBMS has always been a topic of interest for the database community [14], and RDL offers an opportunity to re-explore this possibility.

For example, the GNN message-passing operation is a neural realization of a SQL join and group-by aggregation query [10]. It is possible that implementing message passing as a SQL query in a DBMS would benefit from the bulk processing capability it offers. It is also possible to incorporate other ideas developed specifically for machine learning systems [2]. It is also worth noting that operations such as tensor operations on GPUs will likely always be best performed by a system like PyTorch. Exploring this design space with rich tradeoffs offers many research challenges, but can lead to substantial gains in expressiveness, scalability, and efficiency.

# 4 THE SQL-GNN SYSTEM

In order to realize the vision presented in this paper, we are implementing a system that we call SQL-GNN. This section describes our current effort in implementing this system.

## 4.1 Training Pipeline Implementation

Our training pipeline is based on the RelBench benchmark implementation [12], which defines a suite of relational learning tasks along with their corresponding training, validation, and test splits. It includes implementations of heterogeneous GNN models such as `HeteroGraphSAGE`, and is tightly integrated with PyG, enabling the use of high-performance data loaders for GNN training. Importantly, RelBench allows for customizable neighborhood sampling strategies along with providing easy integrations for native PyG samplers. We extend this flexibility by incorporating DuckDB—an in-process, high-performance analytical SQL engine [23]—to express neighbor sampling logic declaratively in SQL. This combination of RelBench, PyG, and DuckDB allows us to express and execute relational graph sampling logic in SQL while maintaining compatibility with the PyG training workflow. Our current pipeline (illustrated in Figure 3) focuses on integrating an SQL-based subgraph sampler directly into the GNN training loop. This design enables declarative, schema-aware neighbor sampling via SQL, in place of the typical hand-coded Python sampling routines that come with the PyG library. The pipeline consists of the following steps:

(1) **Relational Data Storage:** We begin with normalized relational tables (e.g., `Customer, Review, Product` in the `rel-amazon` dataset) where each table represents a distinct entity type, and foreign-key columns define relationships between these entities in the database schema.
(2) **Global Graph Construction**: Using the RelBench implementation of Global Graph Construction, we transform relational tables and their schema into a heterogeneous graph in memory. Each row in each table becomes a node of the corresponding type, and edges are created for every foreign key reference, linking a row with its referenced row. This yields a graph structure that mirrors the relational links. For each node, a feature vector is derived from its row attributes, combining numeric features, learned embeddings for categorical fields, and vector representations of any textual fields using PyTorch Frame [17]. This global graph (all nodes, features, and edges) is maintained entirely in-memory for fast access during training. While this materialization facilitates on-the-fly retrieval of subgraph node features during training, it is not conceptually necessary; the global graph is materialized solely to simplify implementation.
(3) **SQL-Based Neighbor Sampling:** During training, instead of relying on a Python-based neighbor sampler (e.g., GraphSAGE style sampling), we leverage the embedded DuckDB SQL engine to perform on-the-fly subgraph sampling. At each iteration, the PyG training loop provides a mini-batch of seed node IDs (for instance, a set of Customer node IDs for a prediction task). These seed IDs are passed as parameters to a pre-defined SQL query executed by DuckDB running in-process, in-memory. The SQL query encodes the neighbor sampling logic, a multi-hop expansion of the seed nodes to their neighbors via joins along foreign-key relationships. For example, the query can join a batch of `Customers` to their related `Reviews`, then join to `Products`, thereby retrieving all neighbor nodes and edges within a certain hop radius. We can easily incorporate sampling constraints or filters in SQL (e.g., limiting the number of neighbors per node or applying `WHERE` clauses to enforce application-specific criteria), and we show in the next section that this improves training speed and accuracy.
(4) **Subgraph Retrieval:** The SQL query returns the identifiers of the nodes and edges that form the sampled subgraph for the current batch. With these results, the system gathers the required node features and connectivity information from the global in-memory graph. Because the full graph's adjacency lists and feature vectors reside in memory, this lookup is fast and avoids any disk I/O. The query output identifies the subgraph's nodes and edges, while the in-memory graph provides their features and linkage.
(5) **HeteroData Subgraph Assembly:** Next, we package the sampled mini-batch subgraph into a `HeteroData` object from PyG, which is PyG's native data container for heterogeneous graphs. It organizes the subgraph's nodes by type (each with their feature vector) and edges by relation type (with corresponding edge index lists). The `HeteroData` representation is directly compatible with heterogeneous GNN models in PyG and mostly importantly compatible with training the `HeteroGraphSage` GNN which is the backbone model we train.
(6) **GNN Forward Pass and Training Loop:** The assembled subgraph is then fed into our GNN model, which in our implementation is an $n$-layer `HeteroGraphSAGE`. The model performs a forward pass on this mini-batch, aggregating messages across the sampled neighborhood hops, and computes the training loss. We then apply back-propagation to compute gradients and update the model parameters. This entire process (Steps 3–6) repeats for each batch of seed nodes, cycling through the training data until convergence.

By integrating the SQL engine directly into the training pipeline, SQL-GNN replaces traditional in-memory neighbor sampling with declarative SQL queries executed within DuckDB, which improves the flexibility, accuracy, and running time of RDL.

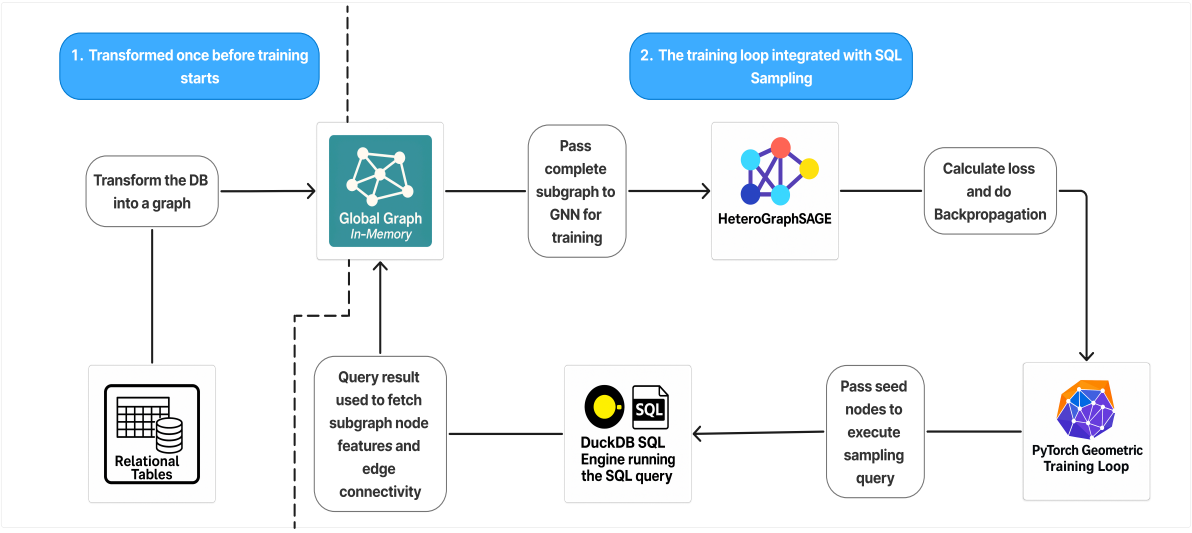


**Figure 3: Overview of our training pipeline.**

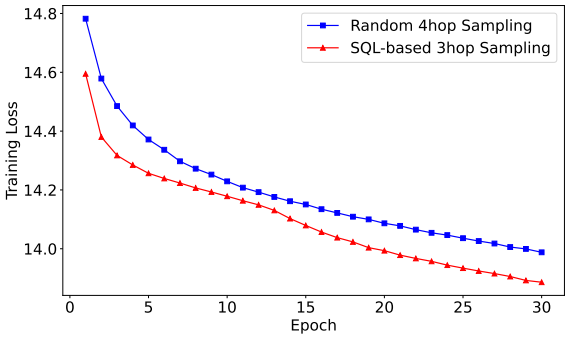


**Figure 4: Training loss for `user-ltv`**

# 5 EXPERIMENTS

## 5.1 Experimental Setup

All experiments were conducted on a high-performance Linux server running Ubuntu 22.04. The machine is powered by an AMD EPYC 9554 processor with 64 physical cores and 128 threads and has 768GB of RAM. The server is equipped with four NVIDIA L40S GPUs, each providing 46 GB of VRAM. This setup offers substantial computational capacity to support efficient training of large-scale graph neural networks. We adopt a 4-layer `HeteroGraphSAGE` architecture (Section 2) using sum aggregation at each layer. All models are trained for 30 epochs with a batch size of 128 and a learning rate of 0.005. These hyperparameters remain constant across all experiments to ensure consistency and enable a fair comparison between different sampling strategies and model configurations.

The experiments in this section are meant to provide examples that demonstrate the benefits of SQL-based sampling. For all the experiments, we use the node regression and node classification tasks from the `rel-amazon` dataset of the RelBench benchmark [24].

## 5.2 Task Specific Subgraph Formation

Using the `user-ltv` task (recall that the goal of this task is to predict the value of a customer's future purchases) we demonstrate one of the key benefits of SQL-based sampling which is its simplicity in defining task-specific subgraphs around seed nodes. We evaluate two subgraph sampling strategies: (1) a standard GraphSAGE-based random sampling approach, and (2) a SQL-based neighborhood extraction (Listing 2).

For the first approach, we employ a uniform neighbor sampling strategy to construct mini-batch subgraphs. In particular, we sample 4-hop neighborhoods around each seed customer node, with a fanout of 64 neighbors at each hop. To maintain temporal consistency, we impose a uniform temporal sampling constraint: at each hop, neighbors are chosen uniformly at random from those whose timestamps precede the seed customer's own interactions. The resulting sampled subgraph extends the structure illustrated in Figure 2 by one additional layer: every review node at the third hop is connected to the customer who authored it, introducing a fourth-hop customer node. We hypothesize that these distant fourth-hop customer nodes provide a signal for the LTV prediction task of only limited usefulness. In the `rel-amazon` dataset, each customer node has only one attribute, the customer's name. Since the name of one customer is not informative for predicting the LTV of another customer, connecting a seed customer to other customers four hops away via shared products and reviews may introduce more noise than information, once closer three-hop neighbors are considered.

To capture more task-relevant context, we formulated an alternative subgraph sampling method using an SQL-based three-hop query (Listing 2). This declarative approach explicitly follows the relational schema and focuses on semantically meaningful connections: starting from a seed customer, it retrieves that customer's own reviews (hop1) and the products of those reviews (hop2), then finds other reviews of those same products (hop3). Notably, this query intentionally skips the extra fourth-hop customer nodes,

thereby limiting the neighborhood to information more directly tied to the seed customer's behavior. The SQL sampling configuration bounds the neighborhood size, with up to 128 direct reviews of the seed customer (max_hop1 = 128) and up to 256 other product reviews (max_hop3 = 256); the hop2 fanout is inherently 1, since each review is associated with exactly one product in the schema.

Empirically, the model trained on subgraphs extracted by the SQL query converged faster and achieved better predictive performance than the model using standard GraphSAGE neighbor sampling. As shown in Figure 4, the loss curve for the SQL-based model decreases more rapidly, indicating faster convergence during training. This suggests that the three-hop neighborhood constructed using SQL-based sampling offers a more focused and useful relational signal, enabling the model to learn useful representations more efficiently. The validation and test Mean Absolute Error (MAE) are shown in Table 1, and they demonstrate that the superiority of SQL-based sampling is evident not only in the training loss, but also in the overall validation and test accuracy measures.

We also evaluate the SQL-based subgraph approach on the `user-churn` prediction task, which is a node classification problem. We use the same three-hop SQL query from Listing 2 to define subgraphs around each customer node, as both the user-ltv and user-churn tasks depend on learning high-quality embeddings for customer nodes. This method achieves a validation AUCROC of 70.777 and a test AUCROC of 70.938, which is similar to recent state-of-the-art work [4]. This experiment shows that the SQL-based sampling query has strong predictive performance for tasks involving customers and can be reused for multiple tasks.

Note that it is possible to implement the same sampling logic used for the SQL-based sampling in a Python-based sampler. However, the benefits of SQL-based sampling are simplicity in expressing complex sampling logic and efficiency in query execution. We explore more complex sampling logic next.

| Subgraph Formed Using | Validation MAE | Test MAE |
|---|---|---|
| Listing 2 (three-hop) | **11.797** | **13.884** |
| Random four-hop sampling | 11.963 | 14.130 |

**Table 1: Accuracy for `user-ltv`.**

## 5.3 Filtering Irrelevant Nodes

We consider the `item-ltv` and `item-churn` prediction tasks of the `rel-amazon` dataset, which are a node regression and a node classification task, respectively. Learning high-quality item embeddings is critical for these tasks.

Building on the results in the previous section, we compose a SQL query to perform three-hop neighborhood expansion, starting from a seed product (i.e., item) node. Listing 3 shows this query, which uses a CTE for each hop. Starting from a product ID, the query selects a random sample of reviews for that item (hop1); it then finds the customers who authored those reviews (hop2); finally, it retrieves other reviews written by those customers (hop3). We retrieve a random sample of up to 256 reviews associated with the seed product, along with the customer IDs of the reviewers (i.e., max_hop1 = 256). Given the relational schema, each review is written by exactly one customer, so hop2 will contain at most 256

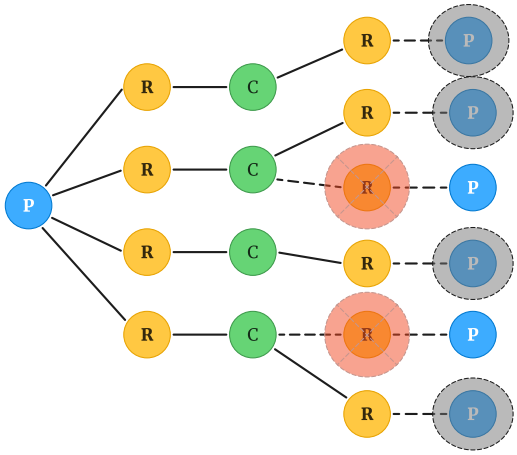


**Figure 5: Filtering the third hop of a product node subgraph. P represents product nodes, R represents review nodes, and C represents customer nodes.**

distinct customers, one per review. In hop3, we join those customers with the reviews table again to fetch up to 128 additional reviews authored by the same customers (i.e., max_hop3 = 128), completing the three-hop traversal.

```
WITH
hop1 AS (
  SELECT review_id, customer_id
  FROM reviews AS r
  WHERE r.product_id = :seed_product_id
    AND r.review_timestamp < :cutoff_timestamp
  ORDER BY random()
  LIMIT :max_hop1
),
hop2 AS (
  SELECT DISTINCT customer_id
  FROM hop1
),
hop3 AS (
  SELECT r2.review_id
  FROM reviews AS r2
  JOIN hop2 AS h2
    ON r2.customer_id = h2.customer_id
 WHERE r2.product_id != :seed_product_id
   AND r2.review_timestamp < :cutoff_timestamp
  ORDER BY random()
  LIMIT :max_hop3
)
SELECT review_id, customer_id FROM hop1
UNION
SELECT NULL, customer_id FROM hop2
UNION
SELECT review_id, NULL FROM hop3;
```

**Listing 3: Three-hop neighbor sampling SQL query for product⟶review⟶customer⟶review.**

We use the subgraphs sampled by this query in both the `item-ltv` and `item-churn` tasks. The results are shown in Table 2, in the first

| Task | Query | Validation Metric | Test Metric | Average Reviews in third hop (training) | Average Reviews in third hop (validation) | Average Epoch Time (training + validation) |
|---|---|---|---|---|---|---|
| item-ltv | Listing 3 | 37.331 | 43.244 | 81238.72 | 145937.84 | 1.90 hours |
| item-ltv | Listing 4 | 38.006 | 44.167 | 19967.56 | 39256.44 | 1.00 hour |
| item-churn | Listing 3 | 82.571 | 83.085 | 81238.72 | 145937.84 | 2.97 hours |
| item-churn | Listing 4 | 82.516 | 82.9767 | 19967.56 | 39256.44 | 1.14 hours |

**Table 2: Results with and without filtering reviews in the third hop.**

and third row, respectively. The third and fourth columns of the table show the validation and test accuracy. The accuracy metric for `item-ltv` is MAE and the accuracy metric for `item-churn` is AUCROC. The accuracy results in these rows are comparable to (slightly better than) recent work [4] showing that our SQL-based sampling is effective, achieving state-of-the-art performance, and can be reused for multiple tasks on the same data.

```
hop3 AS (
  SELECT r2.review_id
  FROM reviews r2
  JOIN hop2 h2 ON r2.customer_id = h2.customer_id
  JOIN products p2 ON p2.product_id = r2.product_id
  WHERE r2.product_id <> :seed_product_id
    AND r2.review_timestamp < :cutoff_timestamp
    AND p2.price BETWEEN
    0.9 * (SELECT price FROM products
    WHERE product_id = :seed_product_id)
    AND 1.1 * (SELECT price FROM products
    WHERE product_id = :seed_product_id)
  ORDER BY random()
  LIMIT :max_hop3
)
```

**Listing 4: Third-hop sampling with price-range filter.**

However, the main point we want to illustrate in this experiment is that it is possible to optimize the training pipeline by adjusting the subgraph sampling using domain-specific knowledge. In particular, we can use domain-specific knowledge to remove less relevant nodes from the sampled subgraphs. For this experiment, we posit that focusing on reviews of similar products to the seed product will improve training speed without affecting accuracy. We define a *similar product* as one whose price is between 90% and 110% of the seed product's price. In the third hop, when we sample additional reviews of customers who reviewed the seed product, we filter out reviews of products that are not similar to the seed product according to our definition, thereby applying domain-specific constraints to remove less relevant nodes from the sampled subgraph.

Figure 5 shows how we perform neighborhood filtering at the third hop: The seed product (P, blue) connects to first-hop review nodes (R, yellow), which in turn link to second-hop customer nodes (C, green). Third-hop review nodes (R, yellow) represent reviews that those customers wrote for other products (P, blue). Our filtering strategy prunes away any third-hop review whose associated product (P, gray) falls outside a ±10% price range of the seed product. In the figure, the crossed-out red circles highlight pruned third-hop review nodes.

This filtering constraint is very simple to implement, demonstrating the power of SQL-based sampling. We extend the hop3 part of the SQL query to join each third-hop review with its product information and apply a price filter in the WHERE clause, as shown in Listing 4 (only the hop3 part of the query is shown for brevity, and the added parts are highlighted). This concise modification prunes a substantial portion of the third-hop neighborhood.

Table 2 shows the results of using the query in Listing 4 that filters irrelevant nodes in the second and fourth rows. The third and fourth columns show that filtering irrelevant nodes has a minor effect on accuracy; the accuracy metrics on the test data degrade by a very small amount. The fifth, sixth, and seventh columns of the table show the substantial savings in training data volume and training time due to filtering irrelevant nodes. The fifth and sixth columns show that the average number of third-hop reviews in the training and validation splits of `rel-amazon` drops by 75% and 73%, respectively, meaning the third-hop neighborhood is only about one-quarter its original size. This aggressive pruning translates to significantly faster epoch runtimes, as shown in the final column of the table. Adding the price-range filter yields approximately 2x speedup without harming model accuracy.

## 6 CONCLUSION AND FUTURE RESEARCH

This paper introduces a vision for Relational Deep Learning in which a portion of the work to train the GNNs used in RDL is done in SQL and inside the DBMS. This has advantages in terms of expressiveness, flexibility, and speed. We presented initial results from the SQL-GNN system, showing that using SQL queries to sample the subgraphs for GNN training provides a simple, powerful, and flexible way to improve model accuracy and training speed.

Future work includes optimizing these SQL queries to ensure that they are expressed in the simplest possible form, combining the queries to use the power of DBMS query processing for bulk sampling, and adding awareness of these queries to the query optimizer to improve the execution plans it selects for them. In this paper, we decided the optimizations to include in the SQL queries used for sampling manually, using our domain knowledge. An important direction for future research is automatically choosing the best subgraphs for a given database and task, leveraging the full power of SQL, including grouping and aggregation. Beyond SQL for subgraph sampling, an open question is whether the graph representing the relational database can be made entirely virtual. This is doable in principle, but it remains to be seen if the required research and engineering efforts justify the expected savings in running time and data movement. Ultimately, it would be interesting to investigate how much of the GNN training can be done inside the DBMS, using SQL, existing query processing operations, and possibly new specialized query processing operations.

# REFERENCES


[1] Martín Abadi, Paul Barham, Jianmin Chen, Zhifeng Chen, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Geoffrey Irving, Michael Isard, et al. 2016. {TensorFlow}: a system for {Large-Scale} machine learning. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*. 265–283.

[2] Matthias Boehm, Iulian Antonov, Sebastian Baunsgaard, Mark Dokter, Robert Ginthör, Kevin Innerebner, Florijan Klezin, Stefanie N. Lindstaedt, Arnab Phani, Benjamin Rath, Berthold Reinwald, Shafaq Siddiqui, and Sebastian Benjamin Wrede. 2020. SystemDS: A Declarative Machine Learning System for the End-to-End Data Science Lifecycle. In *Proc. Conf. on Innovative Data Systems Research (CIDR)*.

[3] Aydin Buluç, Jeremy T Fineman, Matteo Frigo, John R Gilbert, and Charles E Leiserson. 2009. Parallel sparse matrix-vector and matrix-transpose-vector multiplication using compressed sparse blocks. In *Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*. 233–244.

[4] Tianlang Chen, Charilaos Kanatsoulis, and Jure Leskovec. 2025. RelGNN: Composite Message Passing for Relational Deep Learning. *arXiv preprint arXiv:2502.06784* (2025).

[5] Ha Na Cho, Imjin Ahn, Hansle Gwon, Hee Jun Kang, Yunha Kim, Hyeram Seo, Heejung Choi, Minkyoung Kim, Jiye Han, Gaeun Kee, et al. 2022. Heterogeneous graph construction and HinSAGE learning from electronic medical records. *Scientific Reports* 12, 1 (2022), 21152.

[6] CSIRO Data61. 2020. *Heterogeneous GraphSAGE (HinSAGE) — StellarGraph v1.2.1 documentation*. https://stellargraph.readthedocs.io/en/stable/hinsage.html

[7] Milan Cvitkovic. 2020. Supervised learning on relational databases with graph neural networks. *arXiv preprint arXiv:2002.02046* (2020).

[8] Yuxiao Dong, Nitesh V Chawla, and Ananthram Swami. 2017. metapath2vec: Scalable representation learning for heterogeneous networks. In *Proceedings of the 23rd ACM SIGKDD international conference on knowledge discovery and data mining*. 135–144.

[9] Maurizio Ferrari Dacrema, Paolo Cremonesi, and Dietmar Jannach. 2019. Are we really making much progress? A worrying analysis of recent neural recommendation approaches. In *Proceedings of the 13th ACM conference on recommender systems*. 101–109.

[10] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. 2023. Relational deep learning: Graph representation learning on relational databases. *arXiv preprint arXiv:2312.04615* (2023).

[11] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. *arXiv preprint arXiv:1903.02428* (2019).

[12] Stanford SNAP Group. 2023. relbench. https://github.com/snap-stanford/relbench. Accessed: 2025-05-25.

[13] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. *Advances in neural information processing systems* 30 (2017).

[14] Joseph M. Hellerstein, Christopher Ré, Florian Schoppmann, Daisy Zhe Wang, Eugene Fratkin, Aleksander Gorajek, Kee Siong Ng, Caleb Welton, Xixuan Feng, Kun Li, and Arun Kumar. 2012. The MADlib Analytics Library or MAD Skills, the SQL. *Proc. VLDB Endow.* 5, 12 (2012), 1700–1711.

[15] Weihua Hu, Matthias Fey, Hongyu Ren, Maho Nakata, Yuxiao Dong, and Jure Leskovec. 2021. Ogb-lsc: A large-scale challenge for machine learning on graphs. *arXiv preprint arXiv:2103.09430* (2021).

[16] Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. 2020. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems* 33 (2020), 22118–22133.

[17] Weihua Hu, Yiwen Yuan, Zecheng Zhang, Akihiro Nitta, Kaidi Cao, Vid Kocijan, Jinu Sunil, Jure Leskovec, and Matthias Fey. 2024. Pytorch frame: A modular framework for multi-modal tabular learning. *arXiv preprint arXiv:2404.00776* (2024).

[18] Ao Liu, Jing Chen, Ruiying Du, Cong Wu, Yebo Feng, Teng Li, and Jianfeng Ma. 2024. HETEROSAMPLE: Meta-path Guided Sampling for Heterogeneous Graph Representation Learning. *IEEE Internet of Things Journal* (2024).

[19] Xiao Liu, Lijun Zhang, and Hui Guan. 2022. Uplifting message passing neural network with graph original information. *arXiv preprint arXiv:2210.05382* (2022).

[20] Dmytro Lopushanskyy and Borun Shi. 2024. Graph Neural Networks on Graph Databases. *arXiv preprint arXiv:2411.11375* (2024).

[21] Federico Lopez Jure Leskovec Matthias Fey, Vid Kocijan. 2025. Introducing KumoRFM: A Foundation Model for In-Context Learning on Relational Data. https://kumo.ai/company/news/kumo-relational-foundation-model/.

[22] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. PyTorch: An Imperative Style, High-Performance Deep Learning Library. *arXiv e-prints* (2019), arXiv–1912.

[23] Mark Raasveldt and Hannes Mühleisen. 2019. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 international conference on management of data*. 1981–1984.

[24] 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.

[25] Michael Schlichtkrull, Thomas N Kipf, Peter Bloem, Rianne Van Den Berg, Ivan Titov, and Max Welling. 2018. Modeling relational data with graph convolutional networks. In *The semantic web: 15th international conference, ESWC 2018, Heraklion, Crete, Greece, June 3–7, 2018, proceedings 15*. Springer, 593–607.

[26] Benjamin Taskar, Eran Segal, and Daphne Koller. 2001. Probabilistic classification and clustering in relational data. In *International joint conference on artificial intelligence*, Vol. 17. Citeseer, 870–878.

[27] Minjie Yu Wang. 2019. Deep graph library: Towards efficient and scalable deep learning on graphs. In *ICLR workshop on representation learning on graphs and manifolds*.

[28] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S Yu. 2019. Heterogeneous graph attention network. In *The world wide web conference*. 2022–2032.

[29] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. 2019. Graph transformer networks. *Advances in neural information processing systems* 32 (2019).

[30] Lukáš Zahradník, Jan Neumann, and Gustav Šír. 2023. A deep learning blueprint for relational databases. In *NeurIPS 2023 Second Table Representation Learning Workshop*.

[31] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 793–803.