MetaSieve: Faster Relational Deep Learning through SQL-Based Metapath Selection

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

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

ABSTRACT

Relational Deep Learning (RDL) is an effective approach to machine learning over multi-table relational databases. In RDL, a database is modeled as a graph in which each row is a node and each foreign-key relation is an edge, and a graph neural network (GNN) is trained on this graph. Training a GNN requires sampling a subgraph around every seed node in the training set, and the cost of training is largely determined by the size of these subgraphs. This paper aims to reduce subgraph size by leveraging the join and aggregation capabilities of relational database systems. We observe that sampled subgraphs are obtained by following metapaths composed of foreign-key links, and that many of these metapaths can be pruned without loss of accuracy. We present MetaSieve, a metapath selection layer that determines which metapaths to retain and which to prune. For each candidate metapath extension, MetaSieve computes statistics via SQL join and aggregation queries and evaluates the extension based on a novel scoring function that prefers lightweight but informative candidates. Metapaths whose scores fall below a threshold are deemed uninformative and pruned. Metapath selection in MetaSieve is lightweight since it relies only on database statistics and task labels, and it is independent of GNN parameters, so it integrates with diverse GNN architectures for classification and regression. Our evaluation on the RelBench benchmark with multiple GNN backbones shows that MetaSieve consistently reduces per-epoch training time by large margins while maintaining and often improving accuracy.

PVLDB Reference Format:
Fahim Shahriar Khan and Ashraf Aboulnaga. MetaSieve: Faster Relational Deep Learning through SQL-Based Metapath Selection. PVLDB, 20(3): 636 - 648, 2026. doi:10.14778/3860016.3860029

PVLDB Artifact Availability:
Artifacts available at https://github.com/lids-lab/metasieve.

1 INTRODUCTION

Relational databases remain the default foundation for operational and analytical data, and much of the information used to train machine learning models is ultimately stored in relational database tables, connected by foreign-key relationships. Relational Deep Learning (RDL) has emerged as a practical way to learn directly from this multi-table structure by modeling the database as a heterogeneous graph and training a graph neural network (GNN) on this graph [9]. Each row in the database becomes a node in the graph whose type corresponds to its source table, and each foreign-key link becomes an edge whose type corresponds to its foreign-key type. By operating on the native relational structure, RDL sidesteps the need to flatten tables or hand-engineer features, and has shown strong results across multiple prediction tasks [32].

To train an RDL model for a specific task, the training data consists of rows from some table, termed *seed rows*, with a task-specific label associated with each row. For example, if the task is predicting customer churn, the seed rows are customers (identified by their customer id) and the labels indicate whether the customer churned. These rows become the *seed nodes* in the RDL graph, and standard GNN architectures are trained by *sampling a subgraph* from the multi-hop neighborhood of each seed node and using the sampled subgraphs in standard backpropagation neural network training [5, 15, 17]. In the churn example, the sampled subgraph for each customer summarizes their behavior (e.g., purchases, reviews) and is used to train the GNN churn model.

The cost of GNN training depends heavily on the size and complexity of these sampled subgraphs. These subgraphs are typically sampled by randomly selecting neighbors at each hop, which in the RDL context amounts to following random foreign-key links. One way to improve sampling is to consider the *metapaths* being sampled. A metapath is a sequence of foreign-key edge types traversed from a seed node, and it is well established in heterogeneous graph learning that not all metapaths are equally useful [7, 24, 44]. Some metapaths are irrelevant to the task, others occur too rarely to provide a useful training signal, and still others are very frequent but noisy, increasing training cost without adding label-relevant information. An effective way to reduce the cost of GNN training is therefore to identify useful metapaths and restrict subgraph sampling to these metapaths, a problem known as *metapath selection*.

Metapath selection is particularly important for RDL because relational database schemas can be complex, inducing many diverse metapaths of varying usefulness, while large database instances increase the cost of sampling [1, 8]. In this paper, we introduce *MetaSieve (Metapath Selection via SQL-Informed Evaluation of Valuable Extensions)*, a metapath selection technique for RDL that relies on two key ideas: (1) collecting metapath statistics using SQL queries, thereby leveraging a strength of the database system [18], and (2) identifying uninformative metapaths to prune ("sieve") using principled scoring based on the statistics and task labels.

Prior work has shown that counting metapath occurrences is important for metapath selection, since the task label is often determined by aggregate evidence obtained from multiple occurrences of a metapath [8]. MetaSieve counts metapath occurrences using SQL queries that are simple, efficient, and do not require materializing the graph. For example, consider the `rel-stack`

This work is licensed under the Creative Commons BY-NC-ND 4.0 International License. Visit https://creativecommons.org/licenses/by-nc-nd/4.0/ to view a copy of this license. For any use beyond those covered by this license, obtain permission by emailing info@vldb.org. Copyright is held by the owner/author(s). Publication rights licensed to the VLDB Endowment.
Proceedings of the VLDB Endowment, Vol. 20, No. 3 ISSN 2150-8097.
doi:10.14778/3860016.3860029

database from the RelBench benchmark [32], which represents the online activity of users on Stack Overflow, and consider the metapath `user→posts→comments`. We can compute the number of occurrences of this metapath for every seed user using a SQL aggregation query (e.g., `COUNT(comments)`). We can also compute more complex statistics such as the ratio of comments per post, which we term the *rate*, capturing "evidence per opportunity" (e.g., `COUNT(comments) / COUNT(posts)`). The queries used by MetaSieve are simple join-and-aggregation queries that can be executed efficiently, and we structure the computation so that each query is executed only once.

After collecting statistics for all candidate metapaths, MetaSieve treats metapath selection as a dependence-ranking problem and uses the mutual information (MI) between the statistics and the task label as a measure of metapath relevance [29], building on the standard view that MI captures general (including non-linear) dependence and is effective for lightweight feature scoring [2]. To penalize candidates that inflate the sampled neighborhood, MetaSieve makes its relevance measure cost-aware by favoring candidates that achieve high MI with low fanout. MetaSieve also penalizes metapaths infrequently observed in the data, and it combines these measures in a principled way to identify uninformative paths that can be avoided while sampling. Since MI applies to both discrete and continuous labels, the MetaSieve pipeline supports classification and regression tasks. Furthermore, the pipeline is agnostic to the GNN architecture, so it can be used with any GNN backbone.

We conduct experiments with multiple datasets from the RelBench benchmark [32], and we show that MetaSieve substantially outperforms the state-of-the-art MPS-GNN technique [8] in terms of preprocessing time, and **speeds up training by up to an order of magnitude compared to random sampling and MPS-GNN**. MetaSieve **often leads to an improvement in accuracy** in addition to reducing training time, especially with the state-of-the-art Relational Graph Transformer (RelGT) [5] GNN. These results hold for both classification and regression tasks.

In summary, our contributions are as follows:

- We introduce a task-specific metapath selection pipeline for RDL that computes metapath statistics efficiently using SQL queries, thereby avoiding graph materialization while directly leveraging relational structure.
- We propose a novel scoring function for pruning uninformative metapaths that uses mutual information as a lightweight proxy for relevance, penalizes large fanout to favor more compact subgraphs, and downweights infrequently observed metapaths.
- We evaluate the full pipeline on both node classification and node regression tasks from RelBench and show that the resulting metapath selection rules transfer across a wide range of GNN architectures, demonstrating the broad applicability of MetaSieve.

2 RELATED WORK

Metapath selection in heterogeneous graphs. Metapath selection methods for heterogeneous graphs broadly fall into two groups: methods that learn metapath importance jointly with representation learning inside the model, and methods that treat metapath selection as an external search or optimization problem. In the first group, HAN [40] assumes a candidate set of metapaths and learns their importance through hierarchical attention, GTN [42] removes the need for manually specified metapaths by learning soft edge-type compositions across hops, and GraphMSE [24] combines efficient metapath sampling with attention-based weighting and semantic feature-space alignment. In the second group, PM-HGNN [44] formulates metapath selection as a reinforcement-learning problem over node-specific metapaths, while LMSPS [23] treats metapath discovery as an explicit search problem with progressive sampling. MetaSieve is more like the second group since it treats metapath selection as a preprocessing step before GNN training.

Relational deep learning. Fey et al. [9] introduce RDL as an end-to-end learning framework for relational databases. 4DBInfer [39] similarly frames predictive modeling on relational databases as graph-centric multi-table learning and emphasizes the joint role of graph construction, subgraph extraction, and model design in a unified toolbox. RelBench [32] provides a public benchmark and the first broad empirical study of RDL. RelBench v2 [14] expands the benchmark by adding larger, more diverse databases, new prediction tasks, and support for pretraining and transfer. Within this emerging research area, several backbone architectures have been proposed. RelGNN [4] is an early RDL-specific model that addresses message passing challenges across multiple related tables through composite message passing. RelGT [5] is a recent model that uses a transformer-based design over sampled relational neighborhoods and shows remarkable accuracy. Relational Transformers [31] explore pretraining for transfer to unseen datasets and tasks, extending RDL toward relational foundation models [10, 41]. MetaSieve works with any GNN-based RDL model, and we evaluate it on the state-of-the-art RelGT, in addition to older GNN models.

Metapath selection for relational deep learning. Two closely related methods for metapath selection in the node classification setting in RDL have recently been proposed. MP-GNN [7] learns useful metapaths incrementally by scoring each extension of a partial metapath, assuming that node labels can be explained by whether a metapath exists in the data. MPS-GNN [8] shows that this assumption can be overly restrictive, since predictions may depend on aggregate evidence from multiple occurrences of the same metapath. MPS-GNN addresses this limitation by learning a scoring function that implicitly uses occurrence-based information rather than path existence alone. This allows the search to favor metapaths whose occurrence patterns, including higher-order statistical patterns like counts-of-counts, is predictive of the node label. However, the search is costly because assessing candidate extensions requires repeatedly training a GNN on the metapaths selected so far. In our experiments, we found that MPS-GNN does not scale because of the need to train a GNN at each step. We also found that it selects metapaths inferior to MetaSieve. Furthermore, it only works for classification, not regression.

3 PROBLEM DEFINITION

We consider predictive tasks defined by a *task table* over a relational database. Each row of the task table defines one training instance, or *seed node* (s, t, y), consisting of an identifier s, a timestamp t, and a task label y. The identifier s is a foreign key linking the task table to the database table that contains the seed nodes, the *seed node source table*. Sampling begins from this table and expands outward along the graph up to a maximum hop limit H. To prevent temporal

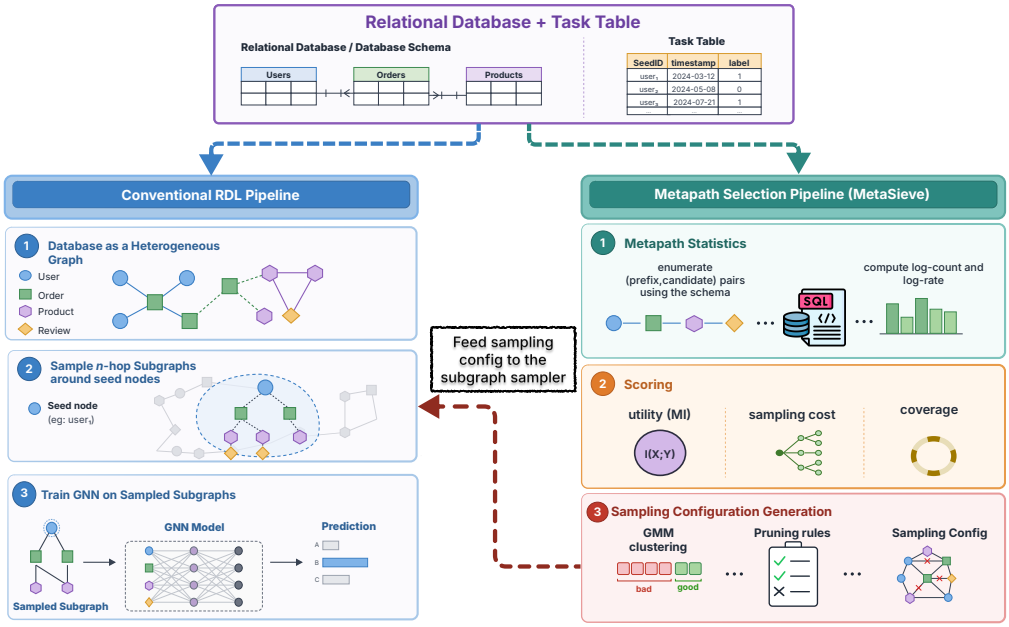


Figure 1: The MetaSieve pipeline. MetaSieve works as a metapath selection layer that plugs into the conventional RDL pipeline. MetaSieve provides sampling configurations to the subgraph sampler, which provides subgraphs for downstream GNN training.

leakage, only traversed rows with timestamp $t' < t$ are sampled. This temporal consistency condition is also used in RelBench. Tables without timestamps require no temporal condition, and all rows in such tables connecting to s are treated as available for sampling. MetaSieve assumes that the database contents, schema, and task definition remain fixed during preprocessing, training, and inference; any changes require rerunning the pipeline. This is a common assumption in RDL, and we leave dynamically handling database or schema updates as future work.

A *metapath* is a sequence of schema-valid foreign-key traversals starting from the seed node source table. We write a metapath of length h as $\pi_{1:h} = (r_1, \ldots, r_h)$, where $h \leq H$ and each r_i is one traversal in the relational graph. As an example, recall the metapath `user→posts→comments` from the introduction.

If table A has a foreign key referencing table B, the relational graph has a directed edge from the A node to the B node, which we call the *forward edge*. We add a *reverse edge* for every forward edge, connecting the primary-key node B to the foreign-key node A. We call a join that follows a forward edge a *forward join* and a join that follows a reverse edge a *reverse join*. Thus, every metapath $\pi_{1:h}$ can be viewed as a sequence of forward and reverse joins.

For a metapath of length h $\pi_{1:h} = (r_1, \ldots, r_h)$, let T_h denote the *terminal table* of this metapath, i.e., the final table in the join sequence induced by $\pi_{1:h}$. When extending this metapath to length $h+1$, we call $\pi_{1:h}$ the *prefix metapath*, and we define the set of valid next edge traversals from T_h as $C(\pi_{1:h})$. Each $e \in C(\pi_{1:h})$ is called a *candidate extension*. Thus, each pair $(\pi_{1:h}, e)$ represents one possible candidate extension of the prefix metapath $\pi_{1:h}$ that MetaSieve evaluates to decide whether it should be pruned or sampled.

Problem definition. Given a maximum hop count H, a database schema, and a set of labeled training seed nodes, the goal of MetaSieve is to produce a compact set of sampling rules $\mathcal{R}$. Each rule is defined for one candidate extension, specified by a hop $h \in \{0, \ldots, H-1\}$, a metapath prefix $\pi_{1:h}$, and the candidate extension itself $e \in C(\pi_{1:h})$. Each rule specifies whether that extension is expanded or pruned: $\mathcal{R}(\pi_{1:h}, e) \in \{\texttt{expand}, \texttt{prune}\}$.

3.1 Metapath Informativeness

Following a metapath from a particular seed node reaches a set of database rows. Although the traversal sequence is fixed by the schema, the number of distinct rows reached can vary substantially across seed nodes. This variation forms the metapath's occurrence pattern and can provide useful predictive evidence when it varies systematically across seed nodes with different task labels, e.g., the number of purchases may provide evidence for customer churn.

A schema-valid metapath may not provide predictive signal for a particular task. Schema validity only means that the traversal is permitted by the database schema, not that it helps predict the task label. For example, in a user-churn task, a metapath to an account-type table may reach exactly one row for every user, regardless of whether the user churns. Because its occurrence pattern is nearly identical across the two label groups, the metapath provides little discriminative signal. A schema-valid metapath may also occur for too few seed nodes to provide reliable evidence or may introduce many neighbors while contributing little label-relevant information.

MetaSieve therefore considers a metapath useful when its occurrence pattern is discriminative of the task label, with this dependence quantified using mutual information (Section 8.1), also when it is observed for enough seed nodes and provides sufficient predictive value to justify its additional sampling cost.

4 METHOD OVERVIEW

The MetaSieve pipeline (Figure 1) works as follows:

(1) **Schema processing.** Scan the schema and enumerate all valid metapaths starting from the seed node source table, up to a maximum of H hops. At every enumeration step, the metapath enumerated thus far is the prefix metapath, and the next step is the candidate extension (Section 5).
(2) **SQL.** Compute metapath statistics for each (prefix, candidate extension) pair over the training seed nodes using SQL queries and without materializing the full graph (Section 6). The seed nodes are divided into *batches* for this computation (Section 7).
(3) **Candidate extension scoring.** Combine these metapath statistics with the seed-node labels and metapath frequency to score each candidate extension. The score reflects the extension's task relevance, sampling cost, and support in the data. Extensions with low scores are pruned (Section 8).
(4) **Sampling rules.** Use the scores to derive pruning rules that specify which candidate extensions should not be sampled. These rules are provided to the sampler and result in a lightweight sampled subgraph for each seed node that still retains the task-relevant evidence needed for GNN training (Section 9).

5 SCHEMA-GUIDED METAPATH ENUMERATION

MetaSieve starts by enumerating all possible metapaths originating from the seed node source table by traversing all schema-valid foreign-key links, forward or reverse, up to a maximum hop count H. At hop h, for a metapath prefix $\pi_{1:h}$ representing a valid sequence of relation traversals from the seed node source table to terminal table T_h, the schema specifies all valid next relation traversals $C(\pi_{1:h})$. For every $e \in C(\pi_{1:h})$, MetaSieve evaluates the pair $(\pi_{1:h}, e)$.

During enumeration, we disallow immediate backtracking in which a reverse edge is followed by its corresponding forward edge, since this returns to the same node. Such a loop would create candidate extensions that add no new information. However, we do allow metapaths that return to the same node via longer cycles through multiple tables.

6 METAPATH STATISTICS USING SQL

MetaSieve collects statistics about each candidate extension $(\pi_{1:h}, e)$ from the database tables using SQL queries. The key idea is to materialize a *frontier table* for each metapath prefix $\pi_{1:h}$ representing all the rows reachable by this prefix, and then to reduce the frontier table to a small set of per-seed-node statistics, without materializing the relational graph. Each frontier table is computed once and reused for all candidate extensions that have $\pi_{1:h}$ as a prefix.

6.1 Computing the Frontier

For a labeled training seed node (s, t, y) and a metapath prefix $\pi_{1:h} = (r_1, \ldots, r_h)$ whose terminal table is T_h, we define the hop-h frontier $F_h(s, t, \pi_{1:h})$ as the set of distinct row identifiers in T_h that are reachable from s by following the join sequence induced by $\pi_{1:h}$ under the temporal constraint at time t. The frontier retains each reachable row identifier only once. As discussed below, duplicate elimination is required for forward joins but not for reverse joins.

The frontier $F_h(s, t, \pi_{1:h})$ is materialized as a frontier table. Every frontier table contains three *core* columns: `SeedId`, `timestamp`, and `last_id`. The pair `(SeedId, timestamp)` identifies the seed node, and `last_id` stores the identifier of a distinct row currently reached

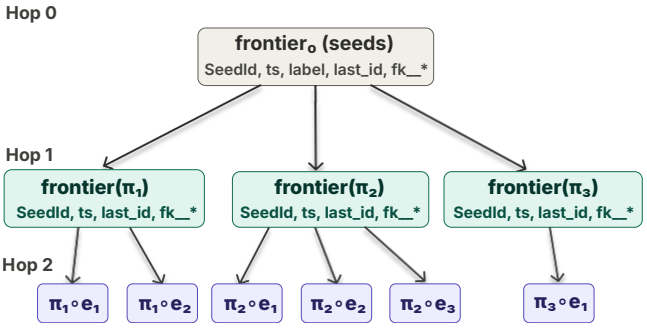


Figure 2: Generating frontier tables across hops. The hop-0 frontier is initialized from the training seed nodes, and each later frontier is generated from its parent by one additional join. Thus, each child hop-h frontier reuses the already-materialized parent hop-$(h-1)$ frontier. *ts* stands for timestamp.

in the terminal table T_h. Thus, for each seed node, the frontier table contains one row for each row of T_h it reaches, and `(SeedId, timestamp, last_id)` forms its primary key. Candidate extensions at the next hop may require following foreign-key links from this row of T_h, *so these links are also included in the frontier table.* That is, the frontier table includes optional columns `foreign_key_1`, `foreign_key_2`, …, `foreign_key_n`. These store the foreign-key values of the current row in the terminal table T_h, which will be needed for forward extension at the next hop.

This frontier table schema supports both forward and reverse joins. If the candidate extension is a reverse join, the next table is reached by matching one of its foreign keys to the current `last_id`, so no additional information is needed in the frontier table beyond the core columns. If the candidate extension is a forward join, the next table is reached by following a foreign-key value in the current row of T_h to the primary key of a row in the referenced table, and all the necessary foreign-key values are in the frontier table.

The two join directions also differ in whether duplicate elimination is required. In a reverse join, each matching row in the next table contributes its own unique row identifier, so the result is already duplicate-free. In a forward join, multiple rows in the parent frontier may contain the same foreign-key value and therefore reach the same row in the next table. MetaSieve consequently applies `DISTINCT` only to forward joins and not for reverse joins.

We initialize the frontier table using the training seed nodes at hop 0. The exact form of this initial frontier depends on the types of candidate extensions at hop 1. If the hop-1 candidate set contains forward joins, the initial frontier must store not only the seed identifier as `last_id`, but also the foreign-key values needed for these forward joins. In this case, we build the hop-0 frontier by a join that attaches each training seed node to its row in the seed node source table and copies the required foreign-key columns (Listing 1). (We only show the listings of three SQL queries here and provide the remaining listings in the full version of this paper [19].)

As described in Section 3, ⟨temporal_constraint⟩ in the WHERE clause is instantiated as $t' < t$ when the joined table has a timestamp t'; otherwise, it is omitted. The same rule is applied whenever a timestamped table is joined in frontier-construction queries.

```
CREATE TEMP TABLE frontier_0 AS
SELECT
    SeedId,
    timestamp,
    SeedId AS last_id,
    ⟨required_foreign_key_1⟩,
    ...,
    ⟨required_foreign_key_m⟩
FROM training_seed_nodes
JOIN seed_node_source_table
  ON seed_node_source_table.primary_key =
     training_seed_nodes.SeedId
WHERE ⟨temporal_constraint⟩;
```

Listing 1: Hop-0 frontier when hop-1 includes forward joins.

If the hop-1 candidate set contains only reverse joins, then no foreign-key columns need to be included in the frontier table. In this case, the initial frontier can store only the training seed node, its timestamp, and the seed identifier as `last_id` (Listing 2).

```
CREATE TEMP TABLE frontier_0 AS
SELECT
    SeedId,
    timestamp,
    SeedId AS last_id
FROM training_seed_nodes;
```

Listing 2: Hop-0 frontier when hop-1 has only reverse joins.

After the hop-0 frontier table is initialized, all later frontier tables are generated in the same way, illustrated in Figure 2. Each frontier is obtained from an already-materialized parent frontier via one additional join. Thus, for a candidate extension $(\pi_{1:h}, e)$, we start from the frontier of the prefix $\pi_{1:h}$ and apply exactly one join corresponding to e, so the prefix frontier serves as reusable state for all of its next-hop candidates. The result of this one-step extension is a new frontier table for prefix $\pi_{1:h+1} = \pi_{1:h} \circ e$. As before, we keep one row in the frontier table per distinct row reachable in the terminal table for each `(SeedId, timestamp)` pair. The new frontier table always updates `last_id` to the identifier of the newly reached terminal row. Per our design, the new frontier table also carries the foreign-key columns required for future joins. Listing 3 shows the query for expanding a frontier with a reverse join. Each matching row in the next table contributes a unique `last_id`, so the query does not require `DISTINCT`. A forward join uses a similar query with a different join condition, but requires `SELECT DISTINCT`.

```
CREATE TEMP TABLE frontier_⟨extended_prefix⟩ AS
SELECT
    SeedId,
    timestamp,
    next_table.primary_key AS last_id,
    ⟨required_foreign_key_1⟩,
    ...,
    ⟨required_foreign_key_m⟩
FROM frontier_⟨prefix⟩
JOIN next_table
  ON next_table.foreign_key = frontier_⟨prefix⟩.last_id
WHERE ⟨temporal_constraint⟩;
```

Listing 3: Extending a frontier by a reverse join.

6.2 Metapath Statistics: log-count and log-rate

A frontier table tells us which rows are reachable by a given prefix for a given seed node. As discussed in Section 3.1, variation in this occurrence pattern across seed nodes can provide predictive signal. To evaluate candidate extensions, MetaSieve summarizes the occurrence pattern using two complementary statistics: *log-count*, which captures the total amount of distinct evidence reached by a metapath, and *log-rate*, which captures the expansion introduced by a candidate extension relative to its parent prefix frontier.

These summaries are related to prior work on heterogeneous graphs and relational retrieval, where metapaths are treated as relation sequences with distinct semantics and path-based summaries are used to capture similarity or relevance [22, 34, 35]. MetaSieve follows the same intuition but uses lightweight SQL statistics over frontier tables instead of complex similarity scores or random walks.

The first statistic is *log-count*, which measures the total amount of distinct evidence reached by a metapath. Recall that, for a seed node (s, t, y) and a metapath $\pi_{1:h}$, $N_h(s, t, \pi_{1:h})$ denotes the number of distinct terminal rows in the hop-h frontier table. We define

$$\text{log-count}_h(s, t, \pi_{1:h}) = \ln\big(1 + N_h(s, t, \pi_{1:h})\big).$$

Log-count therefore captures the absolute volume of distinct reachable evidence. We apply $\ln(1 + \cdot)$ to flatten the potentially heavy-tailed distribution of $N_h(s, t, \pi_{1:h})$.

The second statistic is *log-rate*, which measures how strongly a candidate extension expands relative to the size of its parent frontier. For a candidate extension $(\pi_{1:h}, e)$, let the extended metapath be $\pi_{1:h+1} = \pi_{1:h} \circ e$. We define

$$\text{log-rate}_{h+1}(s, t, \pi_{1:h+1}) = \ln\left(1 + \frac{N_{h+1}(s, t, \pi_{1:h+1})}{N_h(s, t, \pi_{1:h})}\right).$$

Here, $N_{h+1}(s, t, \pi_{1:h+1})$ is the number of distinct rows reached after applying the candidate extension, and $N_h(s, t, \pi_{1:h})$ is the number of distinct rows already reached by its parent prefix. Log-rate therefore captures the extension's relative expansion by measuring how much evidence it reaches per row already present in the parent frontier.

Log-count and log-rate capture complementary aspects of an extension's occurrence pattern. For some extensions, the absolute number of distinct rows reached varies systematically with the task label. For others, the more informative pattern is how much the extension expands relative to its parent frontier, even when the absolute number of reached rows is modest. Log-count captures the first type of variation, while log-rate captures the second. Considering both allows MetaSieve to identify predictive occurrence patterns through either absolute reach or relative expansion.

Figure 3 illustrates how a parent and child frontier are reduced to per-seed metapath statistics. For each seed identified by `(SeedId, timestamp)`, we group the frontier rows using `GROUP BY` and apply `COUNT(*)` to obtain the frontier size. Applying this operation to the child frontier gives the count used for log-count, while applying it to both the parent and child gives the counts used for log-rate.

Because log-rate for a candidate extension depends on both the parent and child frontiers, it can only be computed starting at hop 2. Consequently, pruning begins with hop-1 prefixes and their candidate extensions at hop 2. Frontier tables are retained only while they are needed for expansion and statistics computation, and they are dropped when no longer needed to save storage.

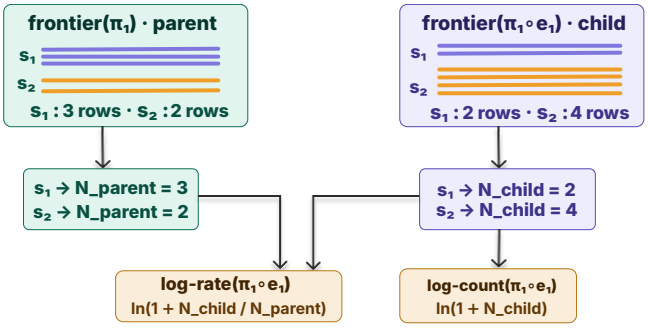


Figure 3: Reduction of frontier tables to metapath statistics. In this example, s_1 and s_2 denote two seeds, each identified by `(SeedId, timestamp)`. Each colored bar represents one distinct reachable row belonging to s_1 or s_2. By grouping the frontier rows by seed and applying `COUNT(*)`, we obtain the parent and child count used for log-count and log-rate.

7 BATCHWISE EVALUATION OF STATISTICS

The previous section treats all seed nodes in the training set $\mathcal{S}_{\text{train}}$ as one batch for statistics computation. MetaSieve actually evaluates candidate extensions on multiple disjoint subsets of training seed nodes (i.e., multiple batches of seed nodes). The log-count and log-rate statistics are computed separately for each batch, and the resulting distribution of statistics lets us identify stable, repeatable signals for metapath pruning. Batching also enables us to address label imbalance while forming the batches.

7.1 Seed Node Batching and Label-aware Subsampling

We score candidate extensions on a set of labeled training seed nodes $\mathcal{S}_{\text{eval}} \subseteq \mathcal{S}_{\text{train}}$. In many cases, $\mathcal{S}_{\text{eval}} = \mathcal{S}_{\text{train}}$, but when the training set is very large or highly imbalanced, we form $\mathcal{S}_{\text{eval}}$ by subsampling a smaller label-aware subset. This reduces the cost of SQL-based scoring while preventing the evaluation from being dominated by a majority label, which could obscure rarer but important signals [16, 20]. We implement this by capping the number of seed nodes per label. Although related in spirit to class-aware resampling [3, 16], the goal here is not to rebalance for model training, but to construct an informative subset for candidate scoring.

After this optional subsampling step, we partition $\mathcal{S}_{\text{eval}}$ into B approximately equal-sized batches, $\mathcal{S}_{\text{eval}} = \bigcup_{b=1}^{B} \mathcal{S}_b$. Each batch $\mathcal{S}_b$ contains labeled training seed nodes and serves as the input for frontier materialization and candidate scoring (discussed in section 8). We implement deterministic batching using the SQL window function `NTILE`, which first orders the seed nodes by identifier and then assigns them to B buckets of approximately equal size [19].

7.2 Benefits of Batchwise Scoring

We compute the metapath statistics log-count and log-rate for all $(\pi_{1:h}, e)$ pairs separately on each batch, and we use the batchwise statistics to score the metapaths. This has two benefits:

- **Stability.** Candidate extensions that score well across many batches are more likely to reflect a repeatable relationship between the statistical evidence and the label. Candidates that score well in only a few batches are treated as less reliable, and the spread of scores across batches reveals whether a candidate extension is informative for a broad subset of training seed nodes [26].
- **Scalability.** Because the batches are disjoint, their statistics can be evaluated concurrently and independently by multiple worker threads. Each worker thread can process one batch end-to-end, including frontier materialization, metapath statistics generation, and scoring, without any cross-worker synchronization. This parallel execution can be implemented on any DBMS and results in throughput that scales linearly with the number of workers due to the coordination-free nature of the parallelism.

8 SCORING CANDIDATE EXTENSIONS

After the SQL-based computation of the log-count and log-rate statistics for the different batches and metapaths, what remains is scoring the candidate extensions based on these statistics, which happens outside the DBMS. Our scoring depends on the utility of an extension to learning the training labels, its cost, and the frequency of the extension in the database. We discuss the details next.

8.1 Mutual Information as a Proxy for Utility

Each training seed node has an associated task-specific label, and our goal is to retain candidate extensions whose statistics are predictive of that label while pruning extensions that are statistically uninformative. To evaluate this, we treat the log-count and log-rate values for an extension as candidate features X, the task label as the target Y, and measure their dependence using mutual information (MI) $I(X;Y)$. This provides a concrete measure of the predictive signal in Section 3.1, where an extension is informative when its log-count or log-rate values vary systematically with the task label. This use of MI is consistent with prior work showing that path-based relational summaries can be informative for prediction [22, 34, 35].

MI is commonly used in supervised feature selection because it quantifies feature-target dependence without requiring a specific predictive model [2, 29]. In our setting, this makes MI a natural task-specific way to assess whether a log-count or log-rate statistic carries useful signal for the corresponding prediction task. MI is also model-agnostic, captures potentially non-linear associations, and is zero if and only if X and Y are independent.

We use MI as a cheap proxy for the utility of an extension because directly estimating utility by training a predictive model and monitoring validation error would be prohibitively expensive. For a candidate extension $(\pi_{1:h}, e)$, we would need to train a model using the per-seed-node statistics available up to $\pi_{1:h}$ and obtain its validation error $E_{\pi_{1:h}}$. We would also train a model on the extended metapath $\pi_{1:h+1} = \pi_{1:h} \circ e$ using the union of the prefix statistics and the new ones introduced by the extension. If the validation error of that model is $E_{\pi_{1:h+1}}$, the utility of candidate extension $(\pi_{1:h}, e)$ can be computed as $E_{\pi_{1:h}} - E_{\pi_{1:h+1}}$. This requires training many models and is not practical for large databases with complex schemas.

8.2 Estimating Mutual Information

In our setting, MI is not available in closed form since we only observe seed nodes, task labels, and metapath statistics for each batch. We therefore estimate MI from data using standard nonparametric estimators based on k-nearest-neighbor distances.

We estimate MI separately for log-count and log-rate within each batch of seed nodes. For batch b, candidate extension $(\pi_{1:h}, e)$,

and statistic $g \in \{\text{log-count}, \text{log-rate}\}$, let $X_b(\pi_{1:h}, e, g)$ denote the vector of values of g for the seed nodes in b. Let Y_b denote the corresponding vector of task labels. We estimate a batchwise mutual information between X_b and Y_b, denoted by $MI_b(\pi_{1:h}, e, g)$. For regression tasks, both X_b and Y_b are continuous, so we use the Kraskov-Stögbauer-Grassberger (KSG) estimator [21]. For classification tasks, X_b is continuous while Y_b is discrete, so we use Ross's mixed discrete-continuous estimator [33]. These standard estimators are readily available in libraries such as scikit-learn [28].

8.3 Entropy-normalized Mutual Information

The batchwise MI estimates introduced above are not directly comparable across batches for a given extension because the uncertainty of the task labels can vary from one batch to another. For example, in a classification task, one batch may have a more balanced label distribution than another, while in a regression task, one batch may span a wider range of target values than another. As a result, the same raw MI value can represent different levels of relevance depending on how much label uncertainty is present in that batch. To make MI values comparable across batches, we normalize each batchwise MI by an entropy term, so that the resulting score measures the fraction of label uncertainty explained by the statistic rather than its absolute MI magnitude. This follows the standard information-theoretic view that normalization is needed when MI values are compared across settings with different entropies [38].

We use the empirical entropy of the labels in each batch b, computed as $H(Y_b) = -\sum_y p_b(y) \ln p_b(y)$. The summation is taken differently for classification and regression tasks. For classification tasks, we sum over the different label values y in batch b. Thus, $p_b(y)$ is the frequency of label value y in batch b normalized by the size of b. For regression tasks, the task label is continuous, so we first discretize it into bins. We choose the number of bins and their ranges once from the training data and then use the same bin count for all batches. We use quantile-based binning so that the bins remain populated even for skewed or heavy-tailed targets [6]. After discretization, entropy is computed by taking the summation over the bins. That is, y ranges over the bins and $p_b(y)$ is the number of times bin y appears in the batch divided by the batch size.

We define the entropy-normalized MI score for batch b as

$$\text{NMI}_b(\pi_{1:h}, e, g) = \frac{MI_b(\pi_{1:h}, e, g)}{H(Y_b)}.$$

It is possible for $H(Y_b)$ to be zero. This happens when the batch contains no variation in the task label, for example, a batch in a classification task where all seed nodes belong to the same class. In this case, we set NMI_b for this batch to zero.

8.4 Cost-aware Scoring

For a given batch, entropy-normalized MI measures how strongly a candidate extension is associated with the task label, but it does not account for how expensive that extension is to sample. In GNN training, computation and memory grow with the size of the sampled neighborhood, and high-fanout extensions can quickly lead to neighborhood explosion [1, 15, 43]. We therefore score each candidate extension in a batch using not only its NMI_b, but also the *cost* of sampling it, as measured by the number of paths it introduces.

We derive the cost of a candidate extension from its log-count values. For a given candidate extension $(\pi_{1:h}, e)$ and a given batch b, the cost is defined as

$$\text{cost}_b(\pi_{1:h}, e) = \text{avg}_{s_i \in b} \left(e^{\text{log-count}_h(s_i, t_i, \pi_{1:h})} - 1 \right)$$

where s_i is a seed node in batch b and t_i is the corresponding timestamp. The expression within parentheses recovers, for the seed node s_i, the actual count of paths for the extension by inverting the equation of log-count. The costs of the extension is then computed as the average of all counts in the batch.

We combine relevance, as measured by NMI_b, and cost into a single batchwise score. For candidate extension $(\pi_{1:h}, e)$, statistic $g \in \{\text{log-count}, \text{log-rate}\}$, and batch b, we define

$$\text{score}_b(\pi_{1:h}, e, g) = \frac{\text{NMI}_b(\pi_{1:h}, e, g)}{\text{cost}_b(\pi_{1:h}, e)}.$$

Thus, an extension receives a high score only when it is both label-informative and cheap to sample within the batch, and MetaSieve prunes metapaths with low score values. This follows the same principle as cost-sensitive feature selection, where a feature is preferred when it is informative yet inexpensive to acquire [13, 27, 36, 37].

8.5 Coverage as a Measure of Support

The scores computed in the previous section tell us whether an extension is informative for the task label and whether it is cheap to sample, but they do not tell us how broadly that extension is observed across the training seed nodes. That is, an extension may receive a high score because it behaves well on a small subset of training seed nodes, yet still be observed too infrequently to support a reliable pruning decision. We therefore introduce a third statistic for evaluating extensions, which we call *coverage*. Unlike log-count and log-rate, coverage is label-agnostic. Its role is to measure how frequently an extension is present in the training data. This is analogous to the notion of *support* in frequent itemset data mining.

For a candidate extension $(\pi_{1:h}, e)$ and a batch b, coverage is the fraction of training seed nodes for which the extended frontier is non-empty, and it is computed by recovering N_{h+1} from log-count:

$$\text{coverage}_b(\pi_{1:h}, e) = \frac{1}{|b|} \sum_{s_i \in b} \mathbb{I}\big(N_{h+1}(s_i, t_i, \pi_{1:h} \circ e) > 0\big).$$

where s_i is a seed node in b, N_{h+1} is the count of the candidate extension, $\mathbb{I}(\cdot)$ is the indicator function which returns 1 if its argument is true and 0 otherwise. A zero N_{h+1} value indicates that no row was reached for this extension and this seed node, and too many zero N_{h+1} values indicate a low-support metapath.

8.6 Combining the Statistics into a Quality Score

For every candidate extension $(\pi_{1:h}, e)$, we compute for every batch $b = 1 \ldots B$ the score for log-count, the score for log-rate, and the coverage. Across the B batches, these values form three *empirical distributions*, which must be combined into a single quality score to determine which metapaths to sample and which to prune.

Our approach combines these scores in two steps. First, we summarize each of the three distributions as a single number. Next, we combine the three summary numbers into one overall quality score per candidate extension.

Summarizing Each Distribution. A natural way to summarize the distribution of a statistic is to take the mean. We instead adopt a more conservative summary: a one-sided *lower confidence bound*

(LCB) on the mean. The LCB discounts extensions whose performance appears good only because of a few favorable batches. In this respect, it plays a role analogous to stability-oriented selection criteria, which favor signals that remain strong under repeated subsampling over signals that are only occasionally strong [26].

For any batchwise statistic $z_1, \ldots, z_B$, under a Gaussian i.i.d. assumption, the LCB is defined as

$$\mathrm{LCB}_\delta\Big(\{z_i\}_{i=1}^B\Big) = \hat{\mu} - t_{1-\delta,\,B-1}\,\frac{\hat{\sigma}}{\sqrt{B}},$$

where $\hat{\mu}$ and $\hat{\sigma}$ are the sample mean and standard deviation across batches, and $t_{1-\delta,\,B-1}$ is the $(1-\delta)$ quantile of the Student-t distribution with $B-1$ degrees of freedom. We use the same significance level δ for all three statistics (log-count, log-rate, and coverage). The parameter δ controls how conservative the resulting estimate is, and we tune it for each specific prediction task.

For every candidate extension $(\pi_{1:h}, e)$, we compute an LCB for log-count, log-rate, and coverage, e.g., the formula for log-count is $\mathrm{LCB}_\delta\Big(\{\mathrm{score}_b(\pi_{1:h}, e, \text{log-count})\}_{b=1}^B\Big)$. We remove extensions for which the LCB of coverage is zero, since this occurs only when the extension does not lead to any actual paths in the data.

The remaining candidate extensions are grouped by hop, and percentile ranks are computed within each hop. Hop-wise grouping matters because the numerical range of the LCB values shifts with hop depth: paths at deeper hops typically exhibit different fanout behavior, span different attainable count scales, and induce different relevance distributions than shallower ones, making raw LCB values incomparable across hops. To eliminate this scale mismatch while preserving the relative ordering of candidates within each hop, we sort the metapaths at each hop separately by each of the three LCB values and compute the percentile rank of each candidate within each sorted list. Thus, each candidate extension $(\pi_{1:h}, e)$ now has three scores on the common scale of $[0, 1]$ summarizing the distribution of each of its three statistics as a single number. The first score is $p_{\log-\mathrm{count}}(\pi_{1:h}, e) = \mathrm{rank}_{\%}\Big(\mathrm{LCB}_\delta\Big(\{\mathrm{score}_b(\pi_{1:h}, e, \text{log-count})\}_{b=1}^B\Big)\Big)$. The other two scores, $p_{\log-\mathrm{rate}}(\pi_{1:h}, e)$ and $p_{\mathrm{coverage}}(\pi_{1:h}, e)$, are computed analogously.

Computing One Overall Quality Score. As discussed in Section 6.2, the usefulness of a candidate extension can be represented in either the log-count or the log-rate score. We therefore represent the relevance of a candidate extension by the stronger of log-count and log-rate. That is, we compute

$$p_{\max}(\pi_{1:h}, e) = \max\big(p_{\text{log-count}}(\pi_{1:h}, e),\ p_{\text{log-rate}}(\pi_{1:h}, e)\big)\ .$$

This avoids penalizing an extension simply because one way of looking at relevance is less appropriate for its semantics.

We also include coverage in the final score of the candidate extension because even a label-informative extension should not be preferred if it is only present in a small fraction of training seed nodes. Our final quality score for a candidate extension is therefore

$$Q(\pi_{1:h}, e) = p_{\max}(\pi_{1:h}, e) + p_{\mathrm{coverage}}(\pi_{1:h}, e).$$

8.7 Separating Promising and Poor Extensions

MetaSieve should keep candidate extensions with a high quality score $Q(\pi_{1:h}, e)$ and prune ones with a low quality score. The question is where to draw the line separating good and bad extensions.

We develop a data-driven process to make this decision. We standardize the Q values by subtracting the mean and dividing by the standard deviation. We then use expectation-maximization [25] to fit a two-component Gaussian mixture model to the resulting Q values. The component with the larger fitted mean is labeled *good*, and the component with the smaller fitted mean is labeled *bad*.

Because Q is based on $p_{\max}$ and p_{coverage}, which are hop-wise percentile ranks, the Q values are much better aligned across hops than the raw LCB values. This makes it possible to apply a single global mixture model to the standardized Q scores while still respecting the different raw scales that arise at different hop depths.

9 GENERATING SAMPLING RULES

After clustering the candidate extensions into good and bad clusters, the remaining task is to convert these cluster assignments into actionable pruning decisions for neighborhood sampling.

We generate a *pruning rule* for each candidate extension that indicates whether the extension should be kept and expanded or pruned: $\mathcal{R}(\pi_{1:h}, e) \in \{\texttt{expand}, \texttt{prune}\}$. For a prefix $\pi_{1:h}$ and candidate extension e, let $\pi_{1:h} \circ e$ denote the corresponding extended metapath. We mark the extension as `expand` if the extended metapath is itself assigned to the good cluster, or if it has at least one descendant metapath assigned to the good cluster. Intuitively, even when the immediate extension is not highly ranked, it should be retained if it serves as a gateway to a strong downstream extension. Conversely, we mark the extension as `prune` only when the extended metapath and *all* of its descendants lie in the bad cluster.

The pruning rules take the form: *"after following prefix $\pi_{1:h}$, do not expand candidate e at hop $h+1$."* Because a hop h node in the terminal table T_h may be reachable through multiple schema-valid metapaths from a seed node, implementing these rules require the sampler to track not only the candidate being expanded at hop h but also the prefix metapath that reached it. Consider, for example, a candidate extension e reachable through two prefix metapaths $\pi_{1:h}^A$ and $\pi_{1:h}^B$, with $\mathcal{R}(\pi_{1:h}^A, e) = \texttt{expand}$ and $\mathcal{R}(\pi_{1:h}^B, e) = \texttt{prune}$. The decision whether to expand (i.e., sample neighbors) depends on which of the two prefix metapaths was traversed to reach e.

If the neighborhood sampler used in the GNN training framework tracks at runtime the prefix used to reach a node while sampling, the pruning rules can be implemented directly in the sampler. However, standard frameworks do not track prefixes while sampling. For example, MetaSieve uses the popular PyG library [11, 12], whose sampler allows the user to specify the extensions to sample and the number of samples per extension type at each hop, but does not track the prefix used to reach each hop. As a result, it cannot distinguish between the two cases described above.

To bridge this gap, we approximate the prefix-aware pruning rules with prefix-agnostic rules of the form: *"if a node at hop h belongs to the terminal table T_h of the prefix metapath $\pi_{1:h}$ of a rule, do not expand candidate e at hop $h+1$."* That is, ignore the specific prefix used to reach T_h. To specify that a candidate not be expanded, we instruct PyG to sample zero neighbors for that candidate at hop h, leaving the default sampling unchanged for other extensions.

Under this approximation, if the terminal table of prefix $\pi_{1:h}$ is reachable through multiple prefixes and even one of them carries a rule $\mathcal{R}(\pi_{1:h}, e) = \texttt{prune}$, the extension e is pruned. Our experiments in the full version of the paper show that this works well [19].

10 EXPERIMENTS

10.1 Experimental Setup

10.1.1 Datasets and Tasks. We evaluate MetaSieve on the RelBench benchmark, which provides realistic databases, temporally scoped predictive tasks, and standardized training/validation/test splits with task-specific metrics [32]. Our evaluation focuses on **node-level classification and regression tasks**, which are the two settings supported by our metapath pruning pipeline. We use five RelBench databases with complex schemas and large instances: `rel-ratebeer`, `rel-trial`, `rel-f1`, `rel-stack`, and `rel-avito`.

We select these databases because their schemas admit many valid metapaths and contain highly connected tables, making metapath selection both useful and challenging. Simpler schemas, such as `rel-amazon`, provide few to no meaningful pruning choices. Further details on the selected databases and additional ablation experiments can be found in the full version of the paper [19].

10.1.2 Hardware and Software Environment. We use DuckDB [30] v1.2.0 as the DBMS. All MetaSieve workers run as threads within the same Python process, and each worker uses its own private, embedded DuckDB instance, with each instance having multiple query execution threads (8 threads per worker by default). The RelBench database is attached to each DuckDB instance in read-only mode. Each of the W workers gets a memory limit of $M_{\text{worker}} = 0.65 M_{\text{available}}/W$, where $M_{\text{available}}$ is the available system memory.

We use Pytorch with the Pytorch Geometric (PyG) library v2.6.1 as the sampling and GNN framework. All experiments were conducted on a GPU cluster with H100 NVIDIA GPUs with 80 GB of memory each and AMD CPUs. Each experiment uses 1 GPU and 2 TB of RAM. The NVIDIA driver is v570.172.08 and CUDA is v12.8.

10.1.3 Sampling Strategies. We compare MetaSieve to two other strategies for constructing the training subgraph for each seed node.

Random m-hop sampling. Starting from the seed node, expand the frontier for m successive hops. At each hop, every frontier node draws a uniform sample of its neighbors with a fixed fanout applied identically across all edge types and hops [15]. This is the default sampling strategy for GNN training, and it is metapath-agnostic: it makes no distinction among the relational paths it traverses.

MPS-GNN. MPS-GNN [8] is the state-of-the-art in metapath selection for RDL. We use the implementation provided by the authors, adapting it to the much larger RelBench databases. MPS-GNN is designed for binary node classification and was originally evaluated on simpler databases without a temporal dimension.

To scale MPS-GNN to the more complex and temporal RelBench databases, we first draw a small label-balanced subset of training seed nodes, together with a small validation subset, and then construct a single temporally consistent H-hop sampled subgraph around those seeds. We pass this sampled subgraph, together with the classification labels of the seeds, to the MPS-GNN search routine. The search is constrained by two parameters: k, which keeps only the top-k candidate extensions at each step, and `beam_width`, which caps the number of partial metapaths retained after each expansion round. These constraints are essential, since increasing either parameter quickly expands the number of explored metapaths, leading to a sharp increase in search time even for small databases. We use $k = 3$, `beam_width`=6 by default. The metapaths evaluated by this search are ranked, and our default is to retain the top 20 and prune the rest, ensuring sufficient coverage. If MPS-GNN evaluates fewer than 20 metapaths, we retain them all.

10.1.4 GNN Backbones. The sampling strategies can be applied to any GNN backbone. We evaluate them with the following three backbone models, which represent message-passing and transformer-based architectures: HeteroGraphSAGE [15], Heterogeneous Graph Transformer (HGT) [17], and the state-of-the-art Relational Graph Transformer (RelGT) [5]. These are also the backbones used in [5], and they are described in detail in the full version of the paper [19].

10.1.5 Hyperparameters. We train the GNNs for 10, 20, or 30 epochs, depending on the dataset size and the GNN complexity. For a given dataset, each GNN backbone is trained for the same number of epochs across all sampling strategies, ensuring a fair comparison. When training for more than 10 epochs, we apply early stopping based on improvement in the validation metric. The neighborhood sampling fanout is set to 64 at every hop in all experiments, except for RelGT on large tasks such as `beer-churn` and `post-votes`, where we use a fanout of 16 to control training cost. The number of GNN layers is always matched to the number of sampled hops for message passing models. The δ value has to be below 0.5, so for each experiment we choose the best δ from $\{0.1, 0.2, 0.3, 0.4\}$ based on the best validation accuracy, using time per epoch as the secondary criterion when the validation accuracies are close [19].

10.2 Improvement in GNN Training

Tables 1 and 2 present the main experimental results of this paper. Table 1 reports test accuracy and per-epoch training time for the RelBench node classification tasks across the five databases in our evaluation, covering all three GNN backbones paired with the three sampling strategies. Table 2 reports the same metrics for node regression tasks, but includes only random sampling and MetaSieve, **since MPS-GNN does not support node regression, a significant limitation relative to MetaSieve**.

Training time. Since the primary goal of MetaSieve is to reduce training time, we first examine per-epoch time. Both MPS-GNN and MetaSieve reduce training time compared to random neighborhood sampling, since they sample smaller subgraphs. However, MetaSieve delivers substantially larger gains on all tasks, with the exception of a few cases on `rel-stack` where the metapaths selected by MPS-GNN results in smaller subgraphs. Thus, we conclude that **MetaSieve achieves the fastest per-epoch time, yielding substantial savings over both MPS-GNN and random neighborhood sampling.** The speedup from MetaSieve can reduce training time from days to hours, for example, on the `beer-churn` task with the powerful RelGT backbone.

Test accuracy. The training-time gains from MetaSieve and MPS-GNN do not come at the cost of accuracy. In fact, these sampling strategies often *improve* accuracy, because the pruned metapaths are typically uninformative, so removing them from the training data yields a more accurate GNN. This effect is more pronounced for RelGT than for HeteroGraphSAGE and HGT. HeteroGraphSAGE and HGT are weaker GNN architectures and cannot fully exploit the improved training data produced by metapath selection. RelGT, in contrast, is a powerful GNN architecture designed specifically for RDL, and is therefore better positioned to leverage higher-quality

Dataset	Task Name	GNN Architecture	Random		MPS-GNN		MetaSieve (ours)	
			Test ↑	Time/epoch ↓	Test ↑	Time/epoch ↓	Test ↑	Time/epoch ↓
rel-ratebeer	user-churn	HeteroGraphSAGE (3 hop)	0.923	1,397	0.949	332	**0.951**	160
		HGT (3 hop)	0.875	3,781	**0.894**	1,010	0.875	721
		RelGT (3 hop, k=50)	0.933	9,742	0.942	2,443	**0.953**	1,253
rel-ratebeer	beer-churn	HeteroGraphSAGE (3 hop)	0.784	3,103	**0.814**	1,554	0.785	518
		HGT (3 hop)	**0.740**	11,181	0.719	5,823	0.725	1,299
		RelGT (3 hop, k=50)	0.777	17,644	0.780	11,102	**0.799**	1,494
rel-trial	study-outcome	HeteroGraphSAGE (3 hop)	0.648	16	**0.668**	8	0.660	5
		HGT (3 hop)	0.552	22	**0.597**	16	0.569	11
		RelGT (3 hop, k=100)	0.691	146	0.676	119	**0.702**	97
rel-f1	driver-dnf	HeteroGraphSAGE (3 hop)	0.712	27	0.676	17	**0.723**	6
		HGT (3 hop)	0.661	41	0.508	26	**0.672**	11
		RelGT (3 hop, k=100)	0.675	624	0.681	331	**0.736**	115
rel-f1	driver-top3	HeteroGraphSAGE (3 hop)	**0.797**	6	0.774	5	0.779	2
		HGT (3 hop)	0.666	9	**0.695**	8	0.673	4
		RelGT (3 hop, k=100)	0.681	141	0.750	123	**0.766**	29
rel-stack	user-badge	HeteroGraphSAGE (4 hop)	**0.891**	699	0.885	314	0.889	185
		HGT (4 hop)	**0.854**	818	0.854	425	0.852	234
		RelGT (3 hop, k=50)	**0.878**	3,808	0.877	2,720	0.877	3,063
rel-stack	user-engagement	HeteroGraphSAGE (4 hop)	**0.907**	1,434	0.904	182	0.906	209
		HGT (4 hop)	0.851	1,846	0.845	221	**0.876**	248
		RelGT (3 hop, k=50)	0.904	4,636	0.901	3,368	**0.905**	3,697
rel-avito	user-visits	HeteroGraphSAGE (3 hop)	0.660	52	**0.662**	35	0.661	18
		HGT (3 hop)	0.626	70	0.632	45	**0.635**	25
		RelGT (3 hop, k=100)	0.654	924	0.653	663	**0.661**	427
rel-avito	user-clicks	HeteroGraphSAGE (3 hop)	**0.678**	68	0.669	30	0.669	16
		HGT (3 hop)	0.648	55	0.633	33	**0.660**	17
		RelGT (3 hop, k=50)	0.649	742	0.640	565	**0.670**	325

Table 1: Node classification performance. Evaluation metric is AUCROC (higher is better). Epoch time is in seconds (lower is better). The fastest time/epoch is highlighted in green, and the best test metrics are in bold.

training data. A clear takeaway from Tables 1 and 2 is that **the powerful RelGT backbone combined with MetaSieve metapath selection consistently achieves the highest accuracy.**

10.3 Preprocessing Time for Metapath Selection

Table 3 reports the preprocessing time of MetaSieve. For each task, the table lists the number of seed nodes, the number of batches into which they are divided, and the number of worker threads in the MetaSieve process. All tasks use 3 hops. The next two columns give the time spent in SQL processing and total preprocessing. SQL processing dominates, and the remaining steps (scoring metapaths, GMM clustering, and generating rules) take only 5–36 seconds. To place these numbers in context, the final column shows the preprocessing time divided by the average epoch time for training RelGT with random neighborhood sampling on the same task. We use RelGT because it is the strongest backbone in our experiments, and random neighborhood sampling because it is the baseline that MetaSieve is designed to improve upon. The results in this last column are striking: **the substantial gains in training time and test accuracy delivered by MetaSieve are obtained at a preprocessing cost of only a fraction of one epoch.**

Table 4 reports MPS-GNN preprocessing time using the default configuration ($k = 3$, `beam_width`=6, retain 20 metapaths) on classification tasks, as it does not support regression. We also considered an expanded search configuration ($k = 6$, `beam_width`=15, retain 30 metapaths), but it resulted in longer search times with limited accuracy gains [19]. The first two columns in Table 4 report the training and validation sample sizes, which we keep small because MPS-GNN trains a separate GNN at every decision step, making larger sample sizes expensive. The last two columns report the total preprocessing time, both in seconds and as a multiple of the average RelGT epoch time on the same task. Despite using far fewer samples than MetaSieve, MPS-GNN is substantially slower, taking up to 45 epochs. We train the GNN on the selected metapaths for at most 30 epochs, so a metapath selection procedure that can itself exceed this budget offers little practical value. In short, **MPS-GNN is too slow to be a viable choice in our setting**.

Dataset	Task Name	GNN Architecture	Random		MetaSieve (ours)	
			Test ↓	Time/epoch ↓	Test ↓	Time/epoch ↓
rel-ratebeer	user-count	HeteroGraphSAGE (3 hop)	7.991	890	**7.651**	220
		HGT(3 hop)	**12.431**	3,740	14.447	736
		RelGT (3 hop, k=50)	8.387	2,086	**7.960**	900
rel-trial	study-adverse	HeteroGraphSAGE (3 hop)	**44.420**	29	44.727	15
		HGT (3 hop)	57.379	80	**57.118**	45
		RelGT (3 hop, k=100)	46.012	388	**44.984**	225
rel-trial	site-success	HeteroGraphSAGE (3 hop)	**0.391**	105	0.437	49
		HGT (3 hop)	**0.462**	410	**0.462**	124
		RelGT (3 hop, k=100)	0.394	1,720	**0.369**	744
rel-f1	driver-position	HeteroGraphSAGE (3 hop)	4.973	14	**4.317**	4
		HGT (3 hop)	4.641	65	**4.103**	4
		RelGT (3 hop, k=100)	4.238	410	**4.110**	84
rel-stack	post-votes	HeteroGraphSAGE (3 hop)	**0.065**	1,084	**0.065**	442
		HGT (3 hop)	**0.068**	1,205	**0.068**	506
		RelGT (3 hop, k=50)	**0.068**	9,233	**0.068**	4,824
rel-avito	ad-ctr	HeteroGraphSAGE (3 hop)	**0.042**	8	**0.042**	3
		HGT (3 hop)	**0.043**	8	0.044	4
		RelGT (3 hop, k=100)	0.039	152	**0.038**	104

Table 2: Node regression performance. Evaluation metric is MAE (lower is better). Epoch time is in seconds (lower is better). The fastest time/epoch is highlighted in green, and the best test metrics are in bold.

Dataset	Task Name	Train Samples	Batch Count	SQL Workers	SQL Time	Total Time	
						Seconds	Epochs
rel-ratebeer	user-churn	300,000	32	4	638	649	0.07
	beer-churn	300,000	32	4	2318	2340	0.13
	user-count	300,000	32	4	652	669	0.32
rel-trial	study-outcome	11,994	8	8	5	10	0.07
	study-adverse	43,335	8	8	11	17	0.04
	site-success	151,407	8	8	10	16	0.01
rel-f1	driver-dnf	11,411	8	8	6	11	0.02
	driver-top3	1,353	8	8	3	8	0.06
	driver-position	7,453	8	8	5	11	0.03
rel-stack	user-badge	363,048	32	8	31	55	0.02
	user-engagement	168,020	32	8	24	35	0.01
	post-votes	407,416	32	8	310	345	0.04
rel-avito	user-visits	86,619	16	4	304	312	0.34
	user-clicks	59,454	16	4	267	274	0.37
	ad-ctr	5,100	8	8	54	59	0.39

Table 3: Time for MetaSieve to complete preprocessing and generate the required sampling configuration.

Dataset	Task	Train Samples	Val Samples	Total Time	
				Seconds	Epochs
rel-ratebeer	user-churn	500	120	3,532	0.27
	beer-churn	500	120	7,922	0.45
rel-trial	study-outcome	720	192	6,573	45.02
rel-f1	driver-dnf	720	192	9,268	14.85
	driver-top3	500	130	6,405	45.43
rel-stack	user-badge	720	192	6,884	1.81
	user-engagement	720	192	10,092	2.18
rel-avito	user-visits	700	180	2,488	2.69
	user-clicks	720	192	3,516	4.74

Table 4: MPS-GNN metapath search duration through hop 3 using the default configuration ($k = 3$, beam_width=6, retain 20 metapaths). Total Time is reported in seconds and as a multiple of the RelGT epoch time with random sampling.

10.4 Scalability of SQL Operations

MetaSieve performs two SQL operations for each candidate extension: frontier materialization followed by aggregation to compute per-seed log-count and log-rate statistics. In this section, we study the scalability of these operations to ensure that MetaSieve is applicable even to large databases. We focus on frontier materialization because extensions entering dense or high-fanout regions can introduce many new rows, expanding small parent frontiers into much larger child frontiers and increasing data volume. On the other hand, aggregation to compute statistics summarizes an existing frontier, so it introduces no rows and reduces data volume.

As the database size grows, the time and memory required for frontier materialization both grow. Table 3 shows that the time required is small, even for large databases. We therefore focus on the memory pressure of frontier materialization, specifically for the three largest databases in Table 3: `rel-ratebeer`, `rel-stack`, and `rel-avito`. Smaller databases do not create memory pressure.

Dataset/Task Name	Workers	Batches	# Cand.	Task Frontier Rows	Peak Child Rows	Concurrent Peak Child Rows	Global Peak RSS (GB)	Wall-Clock Time (s)
rel-ratebeer/beer-churn	4	32	65	124.13B	5.28B	16.73B	1,040.22	1,436
rel-ratebeer/user-churn	4	32	51	51.51B	3.59B	13.25B	630.45	471
rel-avito/user-visits	4	16	23	25.67B	878.28M	3.34B	280.34	155
rel-avito/user-clicks	4	16	23	21.85B	758.26M	2.92B	233.91	135
rel-stack/post-votes	8	32	277	13.02B	153.23M	581.94M	147.65	168
rel-stack/user-badge	8	32	90	252.45M	7.38M	9.14M	12.77	31
rel-stack/user-engagement	8	32	90	217.78M	7.31M	7.42M	11.22	30

Table 5: Frontier materialization and memory pressure. Task Frontier Rows is the total materialized volume, Peak Child Rows is the largest single frontier, and Concurrent Peak Child Rows is the largest combined frontier size of concurrently active queries. Global Peak RSS is the maximum sampled process RSS, and Wall-Clock Time is the end-to-end runtime. M and B denote million and billion rows.

10.4.1 Quantifying Memory Pressure. MetaSieve processes multiple batches concurrently, as seen in Table 3, using multiple workers and multiple execution threads per worker. Consequently, several candidate-extension queries may materialize child frontiers at the same time. Table 5 reports three measures of frontier materialization. *Task Frontier Rows* is the total number of child-frontier rows materialized across the task, reflecting the overall SQL processing workload and explaining execution time. *Peak Child Rows* is the largest child frontier produced by a single candidate-extension query and identifies the largest individual materialization bottleneck. Neither metric captures the additional pressure from concurrent queries. We measure *Concurrent Peak Child Rows* by summing the child-frontier sizes of queries running simultaneously and taking the maximum sum over the entire run. This serves as a proxy for memory pressure from concurrent frontier materialization. The table also reports *Global Peak RSS*, the largest resident set size of the MetaSieve process sampled every 0.5 seconds, which captures the combined memory footprint of all workers, and the end-to-end *Wall-Clock Time*.

The results show that larger Concurrent Peak Child Rows correspond to higher Global Peak RSS, indicating that memory pressure is strongly associated with the combined size of concurrently materialized frontiers. For example, `beer-churn` reaches 16.73 billion concurrent rows and 1,040.22 GB peak RSS. In contrast, `user-badge` and `user-engagement` have concurrent peaks below 10 million rows and peak RSS remain near 12 GB. We next turn our attention to controlling memory pressure using standard techniques.

10.4.2 Regulating Memory Pressure. Memory pressure can be reduced by reducing the number of concurrent workers or increasing the number of batches to make each batch smaller. The effect of these approaches is reported in [19]. Here, we focus on a more direct approach: limiting each DuckDB worker's memory and execution threads. We evaluate this mechanism on `rel-ratebeer`'s two largest tasks, `beer-churn` and `user-churn`, using 4 and 6 workers, respectively, and 32 batches each. Table 6 shows the effect of reducing the per-worker memory limit, along with the number of threads to ensure sufficient memory per thread. Lower memory limits force DuckDB to spill intermediate data to disk rather than retaining it in memory. The table reports *Peak Spill*, the largest total spill-file size across all workers sampled every 0.5 seconds, along with *Peak RSS* and end-to-end *Wall-Clock Time*.

Task	Mem Limit (GB/Worker)	Threads/ Worker	Peak Spill (GB)	Peak RSS (GB)	Wall-Clock Time (s)
beer-churn (4)	default	8	10.64	1,040.22	1,436
	128	8	375.25	520.33	1,335
	64	4	489.83	272.80	1,608
	32	2	504.13	142.15	2,253
	16	2	544.30	76.81	2,150
user-churn (6)	default	8	0.00	974.21	390
	96	8	301.00	577.74	312
	48	4	463.67	304.18	390
	24	2	534.75	160.40	565
	12	2	574.84	84.90	547

Table 6: Effect of per-worker memory and thread limits on memory consumption and runtime. Worker counts are shown in parentheses. All tasks use 32 batches.

The results show that limiting worker memory substantially reduces memory consumption. For `beer-churn`, Peak RSS decreases from 1,040.22 GB to 76.81 GB, while for `user-churn`, it decreases from 974.21 GB to 84.90 GB. Lower memory limits increase disk spilling and execution time due to additional disk I/O. However the increase in execution time remains manageable. Thus, DuckDB enables MetaSieve to process these large tasks with less than 100 GB of peak RSS.

In summary, the time and memory required for frontier materialization in MetaSieve grows as the database size grows. The time remains small relative to the per-epoch training time, and memory pressure can be controlled using standard database techniques.

11 CONCLUSION

We presented MetaSieve, a metapath selection layer for RDL that leverages SQL to identify and prune uninformative metapaths before GNN training. MetaSieve computes lightweight statistics and combines mutual information, sampling cost, and coverage into a scoring function that determines the metapaths to prune. Because the pipeline relies only on database statistics and task labels, it is GNN-agnostic and supports both classification and regression tasks. Experiments on RelBench show that MetaSieve improves both training time and accuracy, especially on RelGT, demonstrating that metapath selection complements strong GNN architectures.

REFERENCES

[1] Muhammed Fatih Balin and Ümit Çatalyürek. 2023. Layer-Neighbor Sampling—Defusing Neighborhood Explosion in GNNs. *Advances in Neural Information Processing Systems* (2023).

[2] Roberto Battiti. 1994. Using Mutual Information for Selecting Features in Supervised Neural Net Learning. *IEEE Transactions on Neural Networks* (1994).

[3] Nitesh V Chawla, Kevin W Bowyer, Lawrence O Hall, and W Philip Kegelmeyer. 2002. SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research* (2002).

[4] Tianlang Chen, Charilaos Kanatsoulis, and Jure Leskovec. 2025. RelGNN: Composite Message Passing for Relational Deep Learning. In *Forty-second International Conference on Machine Learning*.

[5] Vijay Prakash Dwivedi, Sri Jaladi, Yangyi Shen, Federico Lopez, Charilaos I. Kanatsoulis, Rishi Puri, Matthias Fey, and Jure Leskovec. 2026. Relational Graph Transformer. In *The Fourteenth International Conference on Learning Representations*.

[6] Usama M Fayyad and Keki B Irani. 1993. Multi-interval discretization of continuous-valued attributes for classification learning. In *IJCAI*.

[7] Francesco Ferrini, Antonio Longa, Andrea Passerini, and Manfred Jaeger. 2024. Meta-path Learning for Multi-Relational Graph Neural Networks. In *Learning on Graphs Conference*.

[8] Francesco Ferrini, Antonio Longa, Andrea Passerini, and Manfred Jaeger. 2025. A Self-Explainable Heterogeneous GNN for Relational Deep Learning. *Transactions on Machine Learning Research* (2025).

[9] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. 2024. Position: Relational Deep Learning - Graph Representation Learning on Relational Databases. In *Proc. International Conference on Machine Learning*.

[10] Matthias Fey, Vid Kocijan, Federico Lopez, J Lenssen, and Jure Leskovec. 2025. Kumorfm: A foundation model for in-context learning on relational data.

[11] Matthias Fey and Jan Eric Lenssen. 2019. Fast Graph Representation Learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[12] Matthias Fey, Jinu Sunil, Akihiro Nitta, Rishi Puri, Manan Shah, Blaž Stojanovič, Ramona Bendias, Alexandria Barghi, Vid Kocijan, Zecheng Zhang, Xinwei He, Jan Eric Lenssen, and Jure Leskovec. 2025. PyG 2.0: Scalable Learning on Real World Graphs. In *Temporal Graph Learning Workshop @ KDD*.

[13] Russell Greiner, Adam J Grove, and Dan Roth. 2002. Learning cost-sensitive active classifiers. *Artificial Intelligence* (2002).

[14] Justin Gu, Rishabh Ranjan, Charilaos I. Kanatsoulis, Haiming Tang, Martin Jurkovič, Valter Hudovernik, Mark Znidar, Pranshu Chaturvedi, Parth Shroff, Fengyu Li, and Jure Leskovec. 2026. RelBench v2: A Large-Scale Benchmark and Relational Data Repository. In *ICLR 2026 Workshop on Data Problems for Foundation Models (DATA-FM)*. https://openreview.net/forum?id=kplDfjCbC7

[15] Will Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive Representation Learning on Large Graphs. *Advances in Neural Information Processing Systems* (2017).

[16] Haibo He and Edwardo A Garcia. 2009. Learning from imbalanced data. *IEEE Transactions on knowledge and data engineering* (2009).

[17] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous graph transformer. In *Proceedings of the web conference 2020*.

[18] Fahim Shahriar Khan and Ashraf Aboulnaga. 2025. A Vision for SQL-Based Relational Deep Learning. In *Workshop on Tabular Data Analysis at VLDB (TaDA)*.

[19] Fahim Shahriar Khan and Ashraf Aboulnaga. 2026. MetaSieve: Faster Relational Deep Learning through SQL-Based Metapath Selection. *arXiv preprint arXiv:2608.25903* (2026).

[20] Gary King and Langche Zeng. 2001. Logistic regression in rare events data. *Political analysis* (2001).

[21] Alexander Kraskov, Harald Stögbauer, and Peter Grassberger. 2004. Estimating mutual information. *Physical Review E—Statistical, Nonlinear, and Soft Matter Physics* (2004).

[22] Ni Lao and William W Cohen. 2010. Relational retrieval using a combination of path-constrained random walks. *Machine learning* (2010).

[23] Chao Li, Zijie Guo, Qiuting He, and Kun He. 2024. Long-range Meta-path Search on Large-scale Heterogeneous Graphs. *Advances in Neural Information Processing Systems* (2024).

[24] Yi Li, Yilun Jin, Guojie Song, Zihao Zhu, Chuan Shi, and Yiming Wang. 2021. Graphmse: Efficient Meta-path Selection in Semantically Aligned Feature Space for Graph Neural Networks. In *Proceedings of the AAAI conference on Artificial Intelligence*.

[25] Geoffrey J McLachlan and David Peel. 2000. *Finite mixture models*. John Wiley & Sons.

[26] Nicolai Meinshausen and Peter Bühlmann. 2010. Stability selection. *Journal of the Royal Statistical Society Series B: Statistical Methodology* (2010).

[27] Feng Nan, Joseph Wang, and Venkatesh Saligrama. 2015. Feature-budgeted random forest. In *International Conference on Machine Learning*.

[28] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research* (2011).

[29] Hanchuan Peng, Fuhui Long, and Chris Ding. 2005. Feature Selection Based on Mutual Information: Criteria of Max-Dependency, Max-Relevance, and Min-Redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence* (2005).

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

[31] Rishabh Ranjan, Valter Hudovernik, Mark Znidar, Charilaos I. Kanatsoulis, Roshan Reddy Upendra, Mahmoud Mohammadi, Joe Meyer, Tom Palczewski, Carlos Guestrin, and Jure Leskovec. 2026. Relational Transformer: Toward Zero-Shot Foundation Models for Relational Data. In *The Fourteenth International Conference on Learning Representations*.

[32] 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* (2024).

[33] Brian C Ross. 2014. Mutual information between discrete and continuous data sets. *PloS one* (2014).

[34] Chuan Shi, Xiangnan Kong, Yue Huang, Philip S Yu, and Bin Wu. 2014. Hetesim: A general framework for relevance measure in heterogeneous networks. *IEEE Transactions on Knowledge and Data Engineering* (2014).

[35] Yizhou Sun, Jiawei Han, Xifeng Yan, Philip S Yu, and Tianyi Wu. 2011. Pathsim: Meta path-based top-k similarity search in heterogeneous information networks. *Proceedings of the VLDB Endowment* (2011).

[36] Paweł Teisseyre and Tomasz Klonecki. 2021. Controlling costs in feature selection: information theoretic approach. In *International Conference on Computational Science*.

[37] Peter D Turney. 1994. Cost-sensitive classification: Empirical evaluation of a hybrid genetic decision tree induction algorithm. *Journal of artificial intelligence research* (1994).

[38] Nguyen Xuan Vinh, Julien Epps, and James Bailey. 2010. Information Theoretic Measures for Clusterings Comparison: Variants, Properties, Normalization and Correction for Chance. *Journal of Machine Learning Research* (2010).

[39] Minjie Wang, Quan Gan, David Wipf, Zhenkun Cai, Ning Li, Jianheng Tang, Yanlin Zhang, Zizhao Zhang, Zunyao Mao, Yakun Song, et al. 2024. 4DBInfer: A 4d Benchmarking Toolbox for Graph-Centric Predictive Modeling on RDBs. *Advances in Neural Information Processing Systems* (2024).

[40] 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*.

[41] Yanbo Wang, Xiyuan Wang, Quan Gan, Minjie Wang, Qibin Yang, David Wipf, and Muhan Zhang. 2025. Griffin: Towards a Graph-Centric Relational Database Foundation Model. In *Proceedings of the 42nd International Conference on Machine Learning*.

[42] Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. 2019. Graph Transformer Networks. *Advances in neural information processing systems* (2019).

[43] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning Method. In *International Conference on Learning Representations*.

[44] Zhiqiang Zhong, Cheng-Te Li, and Jun Pang. 2022. Personalised meta-path generation for heterogeneous graph neural networks. *Data Mining and Knowledge Discovery* (2022).