Incorporating Deep Learning Design in Database Queries

Yuval Lev Lubarsky*
Technion
Haifa, Israel
lubarsky@cs.technion.ac.il

Dean Light*
University of Washington
Seattle, USA
deanlcs@cs.washington.edu

Boaz Berger
Technion
Haifa, Israel
boazb@campus.technion.ac.il

Shunit Agmon
Technion
Haifa, Israel
shunita@campus.technion.ac.il

Benny Kimelfeld
Technion & RelationalAI
Haifa, Israel
bennyk@cs.technion.ac.il

ABSTRACT

Deep learning over relational databases is conventionally realized by translating data into graph representations and applying graph neural networks (GNNs) within external frameworks. This round-trip between the database and external machine learning (ML) systems introduces non-trivial engineering overhead. In effect, GNNs operate on tuple embeddings and manipulate them in ways that capture the interactions induced by relational joins. Given this correspondence, there is no fundamental reason why specifying a neural network over a database should be substantially harder than querying it. We propose an approach that naturally integrates deep learning with database queries. By associating each tuple with provenance, represented as a vector with learnable parameters, queries operate jointly on tuples and embeddings, producing new output relations with embedded tuples. This approach provides a declarative foundation for relational deep learning, facilitating integration with database systems, optimization, and adoption.

We present RelaNN, a proof-of-concept implementation of this approach built on top of PyTorch and cuDF. We illustrate the utility of RelaNN by implementing various graph-learning models, including graph convolutional networks, heterogeneous graph transformers, hypergraph neural networks and deep homomorphism networks. The simplicity of the programs and their competitive runtime performance demonstrate a concrete path toward making the implementation of state-of-the-art neural networks over databases as simple as writing a query.

VLDB Workshop Reference Format:
Yuval Lev Lubarsky, Dean Light, Boaz Berger, Shunit Agmon, and Benny Kimelfeld. Incorporating Deep Learning Design in Database Queries. VLDB 2026 Workshop: Tabular Data Analysis (TaDA).

1 INTRODUCTION

Integrating databases and machine learning (ML) is a central challenge in data science. Both paradigms are fundamental to data-driven workflows: databases provide powerful abstractions for modeling, querying, and managing data, while ML enables the extraction of predictive models from it. However, they traditionally rely on different modeling primitives and implementation strategies, making their integration nontrivial. As a result, bridging the gap between databases and ML has been a major challenge in database foundations over the past decades [1, 40]. Previous efforts to bridge this gap range from adding black-box ML functionalities to the database [22, 29], through integrating ML building blocks into DBMS kernels [26, 31], to creating unified languages for both paradigms [25, 30, 32, 44, 45]. Advances in deep neural networks, particularly neural networks over graphs [37, 43, 52], have fundamentally advanced relational deep learning by enabling prediction models to operate directly over structured data, avoiding ad-hoc feature engineering.

The common approach to applying deep learning over relational databases is to translate the data into a graph representation: records become nodes and foreign-key relationships become edges. A model (e.g., a GNN) is designed and trained over the resulting graph outside of the database context [5, 7, 10, 14, 33, 48]. To capture schema information of the database, architectures like GNNs and graph transformers have been extended to *heterogeneous* settings, where nodes and edges are associated with different types [50, 54]. Hence, designing a neural network over a relational database amounts to developing a predictive model over the induced graph using imperative ML graph libraries such as PyTorch Geometric [16], DGL [49], or Scikit-network [3], which act as abstractions over PyTorch, providing the graph operations that are cumbersome to write in raw tensor code. The role of the database query language is reduced to exporting the database into a tensor representation of a graph, and reintegrating the predictions back into the database for further processing.

We argue that this round-trip is unnecessary, and that it arises from an artificial separation of abstractions that forces developers to work across disparate paradigms. Instead, relational query languages are inherently well-suited for designing neural networks over relational databases, given an appropriate extension of the relational model. From this perspective, applying graph-learning architectures to a database amounts to manipulating *tuple embeddings*, where each tuple is associated with a (parametric) numeric vector, and the resulting computations reflect interactions between tuples through relational operations. In particular, similarly to the abstraction of database provenance as semirings [19, 20], joins should determine not only how tuples are combined, but also how their embeddings are composed, while projections (grouping) should

*Equal contribution.

Proceedings of the VLDB Endowment. ISSN 2150-8097.

determine how embeddings are aggregated. By extending the relational model with these embedding semantics, we can express deep-learning architectures directly over the database, without resorting to intermediate graph representations. This lets developers design neural models using familiar, declarative query constructs such as SQL and Datalog.

As an example, the following program represents query-key-value attention as used by heterogeneous graph transformers [23], over a dataset of patients and treatments.

```
Score(p,t; q*k) :- Treat(p,t), Queries(p; q), Keys(t; k) .
Attention(p; sum(a*v)) :- Score(p,t; a), Values(t; v) .
```

Each relation is written as $R(\vec{x}; z)$, pairing ordinary content attributes $\vec{x}$ with a tuple *embedding* z. The first rule derives `Score` by joining the three relations on their shared variables p and t: `Treat` links each patient p to the treatment t it received and carries no embedding. `Queries` and `Keys` attach a query vector q to each patient and a key vector k to each treatment. The join brings q and k together on every linked pair (p, t), and the head records their product $q * k$ as that pair's embedding. The second rule joins `Score` with the treatment values `Values` on the shared attribute t, weighting each value v by its score a. Its head, `Attention`, keeps only the patient p and drops t. This projection groups the treatments by patient and applies `sum`, aggregating the weighted values $a * v$ to a single embedding per patient. The program maintains an *assignment* of values to its parameters, and *training* adjusts them by minimizing a loss function.

Our approach adopts the idea of representing every database fact in two domains—the *logical domain* having the traditional database context, and the *numerical domain* that associates a meaning in the Euclidean space and allows for numerical reasoning, as studied in prior work on tuple embedding [4, 5, 11, 33, 36, 48]. This is contrasted with existing in-database ML frameworks that typically bridge the two domains by integrating tensor data types and linear algebra operators into the query engine [26, 30–32, 44, 45], as also endorsed by the recent Tensor Logic [12] that unifies neural and symbolic AI by treating relations as sparse Boolean tensors, interpreting joins as tensor products and projections as sums over dropped dimensions. These frameworks require the user to explicitly define and maintain the connection between the logical and the vector representations, as well as their evolution through the composition of deep models. Rather than collapsing both into one representation, as Tensor Logic does, our proposed framework holds the relational structure and the embedding tensors side by side, allowing the user to interact with both faces of the data.

The theoretical foundations of this approach have recently been studied in terms of *homomorphisms* of patterns (join queries) in the graph domain [2, 34]. However, a key question remains open: whether and how such an approach can be realized in practice. This paper addresses this challenge. We introduce *RelaNN*,[1] an open-source system that enables the design of neural models directly over relational data through a declarative query language as described above. As illustrated in Figure 1, programs are first translated into Neuro-Relational Algebra (NRA) to incorporate embedding manipulation, and subsequently compiled into a physical plan across

[1]Source code: https://github.com/yuvallu/relann.

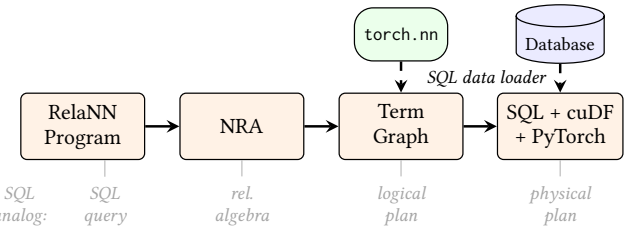

Figure 1: Compilation pipeline of RelaNN. A program is compiled into Neuro-Relational Algebra (NRA), then to a term graph and a physical plan that executes over SQL, cuDF, and PyTorch. Neural operators (e.g., `Linear` and `ReLU`) are resolved at compile time from the torch.nn library.

PyTorch, cuDF, and SQL to execute the model. In the current preliminary stage, we assume that datasets fit in GPU memory, and we are currently developing out-of-core or distributed data support (via designated mini-batching) as an ongoing effort.

NRA extends classic relational algebra by equipping every operator with two complementary semantics: the standard *content semantics*, determining how tuples are derived, and an *embedding semantics*, specifying how their embeddings are computed. In particular, joins combine tuples in the usual sense while concatenating their embeddings; projection and union are unified into a *projected union* operator that aggregates embeddings over multisets of tuples using aggregate functions such as sum, mean, or max; and transformation operators apply differentiable maps (e.g., feed-forward networks) to embeddings, introducing learnable parameters. To support learning, NRA is coupled with a *fit* operation that assigns values to these parameters by minimizing a loss function defined over the output of an NRA expression. Thus, NRA expressions define end-to-end differentiable computations over relational data, unifying relational query processing with neural model design and optimization within a single formal framework.

Our experiments demonstrate several key properties of our approach. First, RelaNN can express a wide range of graph-learning architectures in concise, simple programs, including graph convolutional networks (GCN) [28], relational GCN (R-GCN) [43], heterogeneous graph transformers (HGT) [23], hypergraph neural networks (HyGNN) [42], and deep homomorphism networks (DHN) [34]. Second, RelaNN programs are significantly shorter than the reference implementations and closely mirror the mathematical formulations in the original publications. Third, RelaNN achieves comparable predictive accuracy to the corresponding PyTorch and PyG implementations, and remains within the same order of magnitude in runtime; in the case of DHNs, RelaNN achieves a considerable improvement of runtime, up to 14× faster. Finally, RelaNN enables rapid design and iteration of task-specific models where minor, localized programmatic changes lead to meaningful improvements in predictive performance, significantly reducing engineering overhead.

2 NEURO-RELATIONAL ALGEBRA (NRA)

This section presents NRA—the formal framework that serves as the basis for the RelaNN language. NRA extends standard relational algebra with numerical updates to tuple annotations, which are defined in Section 2.1. Section 2.2 defines the NRA operators. Finally,

Section 2.3 formalizes training as a loss-minimizing assignment of values to the parameters of an NRA expression.

2.1 Neuro-Relational Data Model (NRM)

Our database and schema models extend the classic relational model [9] with vector annotations of tuples, which we name *embeddings*. A schema $\mathcal{S}$ associates a *content arity* k and an *embedding dimension* d with each relation name R. We assume that relation names are unique within a schema.

A *database* D over the schema $\mathcal{S}$ contains, for each relation name R in $\mathcal{S}$ with content arity k and embedding dimension d, an *embedded relation* (r, η), where r is a relation with k attributes and η is an *annotation* that maps every tuple $t \in r$ to an *embedding* $\eta(t) \in \mathbb{R}^d$. We call r the *content* of the embedded relation.

2.2 Neuro-Relational Operators

Having fixed the data model in Section 2.1, we now define *Neuro-Relational Algebra* (NRA): a relational algebra whose objects are embedded relations. NRA operators take a finite number of embedded relations and return a single embedded relation. A finite composition of these operators is named an *NRA expression*. For each operator, we define its *content semantics* and its *embedding semantics*. The former treats the content of the derived embedded relation, while the latter treats its embeddings.

Join ($\bowtie$). This operator executes a natural join, pairing tuples from the given embedded relations (r_1, η_1) and (r_2, η_2) that agree on all shared attributes. Let $(r, \eta) := (r_1, \eta_1) \bowtie (r_2, \eta_2)$ be the join of the two embedded relations. The content semantics of join are defined as usual: $r = r_1 \bowtie r_2$. The embedding of each output tuple is formed by concatenating the contributing embeddings from the input relations. Hence, for $t \in r$:

$$\eta(t) := \eta_1(t[\mathbf{A_1}]) \oplus \eta_2(t[\mathbf{A_2}])$$

where $\oplus$ denotes concatenation and $\mathbf{A_1}$ and $\mathbf{A_2}$ are the attribute sequences of r_1 and r_2, respectively.

Projected union ($\cup_{\alpha,\mathbf{A}}$). This operator is a set-semantics k-ary union combined with a projection. It first projects the tuples of the k given relations onto a common attribute sequence, and then aggregates the embeddings of tuples with identical content. Let $\mathbf{A}$ denote the attribute sequence, and let α denote a multiset aggregator. Let $(r_1, \eta_1), \ldots, (r_k, \eta_k)$ be k embedded relations over the attributes $\mathbf{A}$, and denote $(r, \eta) := \cup_{\alpha,\mathbf{A}}\big((r_1, \eta_1), \ldots, (r_k, \eta_k)\big)$. The projections of the incoming tuples onto $\mathbf{A}$ are unified, that is, $r = \pi_{\mathbf{A}}(r_1) \cup \cdots \cup \pi_{\mathbf{A}}(r_k)$. Each tuple $t \in r$ is associated with a multiset B_t of input tuples whose projection over $\mathbf{A}$ yields t, namely $B_t := \{\!\{t' \in \cup_{i=1}^{k} r_i \mid t'|_{\mathbf{A}} = t\}\!\}$. The embedding of each $t \in r$ is then given by aggregating the embeddings of all the tuples from B_t:

$$\eta(t) := \alpha\big(\{\!\{\eta_i(t') \mid t' \in B_t \cap r_i,\ i \in \{1, \ldots, k\}\}\!\}\big)$$

Note that this operation generalizes union and projection operators. For $k = 1$ we get the standard projection operator. For $k = 2$ and identical relational schemas $\mathbf{A}$ of r_1 and r_2, we get the union operator. The reason for this unification of operators is rooted in the embedding calculation. Any of the sub-operations along the way (projection or union of a subset of the k given relations) may trigger the need for aggregation, due to duplicate tuples. Since we do not assume decomposability of the aggregation (e.g., mean), it must be applied only once, at the end of the full operation.

Transformation (T_τ). A transformation operator alters the embeddings of a single embedded relation. Content is unchanged; a differentiable transformation τ, composed of learnable layers and activations, is applied independently to each tuple embedding. This is the only operator that may introduce new parameters, as part of the learnable layers. We denote by P_τ the set of parameters associated with a transformation operator T_τ. Formally, given an embedded relation (r_0, η_0) and a transformation τ over its embeddings, the outcome $(r, \eta) := \mathsf{T}_\tau(r_0, \eta_0)$ is given by $r := r_0$ and $\forall t \in r : \eta(t) := \tau(\eta_0(t))$.

Other operators. NRA also includes selection (σ), set difference ($\setminus$), and renaming (ρ), which behave as in classical relational algebra and apply no transformation to embeddings: each output tuple keeps its input embedding.

2.3 Training (fit)

To evaluate an NRA expression, the parameters introduced by its transformation operators must be assigned values. Given an NRA expression ϕ, we denote by P_ϕ the union of the parameter sets P_τ for each transformation τ in the composition of ϕ. We now define a training operation that produces the best such values.

A *fit operation* accepts an NRA expression ϕ, its parameter set P_ϕ, and a database D. It returns an *assignment* $\gamma : P_\phi \to \mathbb{R}$ to the parameters, ideally one that minimizes a predefined optimization objective. In Section 3.2 we lift this operation into a language-level statement that maintains γ across calls.

3 THE RELANN LANGUAGE

We next describe the concrete syntax of the RelaNN language, extending the Neuro-Relational Algebra introduced in Section 2.

A *neuro-relational program (NRP)* is a finite sequence of *statements*. It takes as input a database D_{in} over a schema $\mathcal{S}$ and produces an output database D_{out}. The core building blocks are *rule statements*, each adding a new embedded relation to D_{out} (Section 3.1). As in Section 2, rules may introduce parameters; the full set P, with an assignment $\gamma : P \to \mathbb{R}$, is held statefully alongside D_{out}. *Fit statements* (Section 3.2) update the parameter assignment to fit data. *Predict statements* (Section 3.3) evaluate the program at the current assignment without updating it.

On top of these core statements, RelaNN provides two organizational primitives: *aliases* (Section 3.4) for parameter sharing across rules, and *function definitions* (Section 3.5) for code reuse.

3.1 Rule Statements

A RelaNN *rule statement* (*rule* for short) is the basic component of an NRP. It takes as input a set of embedded relations, either from D_{in} or from previous rules, and specifies how to translate them into a new embedded relation that is introduced to the database. Each rule is directly compiled into an NRA expression, composed of neuro-relational operators (Section 2.2). The embedded relations in a rule take the form $R(\vec{x}; z)$, where R is the relation name, and $\vec{x}$ and z tuple-wise bind the relation's content variables and embeddings, respectively. RelaNN supports two types of rules: *join rules* and

union rules. Together with filter clauses in their body, they expose the join, union, projection, transformation, and selection operators of NRA (Section 2.2).

A *join rule* is an expression of the form

$$R(\vec{x};\, \alpha(\tau)) \;:-\; R_1(\vec{x}_1; z_1), R_2(\vec{x}_2; z_2)\,, \cdots,\; R_k(\vec{x}_k; z_k),\;\; \varphi\;.$$

It compiles into the NRA $\cup_{\alpha,\vec{x}}(\mathsf{T}_\tau(\sigma_\varphi(R_1 \bowtie R_2 \bowtie \ldots \bowtie R_k)))$. First, the input embedded relations $R_1, \ldots, R_k$ are joined on their shared variables using $k-1$ NRA join operations. This results in each tuple containing all the input variables $\vec{x}_i$, and the input embeddings concatenated together in the order the relations appear in the rule body. If the (optional) *filter expression* φ is specified, a selection operator is applied to the result. Next, an NRA transformation operator $\tau(z_1 \oplus \cdots \oplus z_k)$ is evaluated. The syntactic expression that specifies τ in the rule is called the *transformation expression*. Finally, an NRA projected union operation is executed over $\vec{x}$ using the aggregator described in the *aggregation expression* α. This yields the output embedded relation R. An example of a join rule can be seen in lines 3–4 of Example 1.

A *union rule* is an expression of the form

$$R(\vec{x};\, \alpha(\tau)) \;:-\; R_1(\vec{x}_1; z_1) \mid R_2(\vec{x}_2; z_2) \mid \cdots \mid R_k(\vec{x}_k; z_k),\;\; \varphi\;.$$

In this rule all the embedded relations R and R_i must share the same structure, including the content attributes (i.e., $\vec{x} = \vec{x}_i$ for all i) and the embedding shape. It is compiled into the NRA expression $\mathsf{T}_\tau(\sigma_\varphi(\cup_{\alpha,\vec{x}}(R_1, R_2, \ldots, R_k)))$. The tuples are first unified using a projected union operation (which reduces to a plain union operator because a union rule requires all its relations to share the same content attributes). They are then optionally filtered by the selection operator using the filter expression φ. Finally, the embeddings are transformed using the transformation expression τ.

The filter expression φ is defined as a comma-separated list of atomic predicates: $\varphi := \varphi_1, \ldots, \varphi_m$. Each predicate φ_i takes the form $e_1 \theta e_2$, where the operator $\theta \in \{=, \neq, <, \leq, >, \geq\}$, and e_1, e_2 are arithmetic terms over the body's content variables and constants. This list is interpreted conjunctively, meaning that its semantics correspond to the predicate $\varphi = \bigwedge_{i=1}^{m} \varphi_i$.

3.2 Fit Statements

A *fit statement* realizes the fit operation (Section 2.3) as a language construct. We maintain an assignment γ to the parameters, and each invocation of the fit operation reads the current γ, computes a new assignment from the data, and writes it back. Note that "fit" does not introduce new content; instead, it specifies how the parameters should be updated based on some given optimization hyperparameters. Syntactically, it is written as

$$\texttt{?fit}\ \langle\text{kwargs}\rangle \quad \mathrm{L}\;.$$

L plays the role of a loss function: its schema has the content arity 0 and embedding dimension 1. Therefore, it contains a single distinguished numeric cell, representing the loss value of an NRA expression. This value is then optimized over the parameters derived from the NRA expression. The optimization process is configured via optimization hyperparameters (e.g., learning rate, number of epochs, weight decay), which are specified in the keyword-argument list kwargs. Executing `?fit` updates the assignment γ by minimizing the single embedding value exposed by L (see Example 1).

3.3 Predict Statements

NRP rule statements define an NRA expression, but never execute it. For execution we introduce a *predict statement*. It takes the NRA expression that is defined by the NRP, the current assignment to the parameters (ideally, previously optimized using the fit statement), and a relation name to be evaluated. Syntactically, it is written as "`?pred R`." (See Example 1, line 11). The embedded relation R must be defined in prior rules. After processing the predict statement, the embedded relation R is included in the output database D_{out}.

3.4 Aliasing

Programs may include *aliases* that introduce readable names for expressions reused across rules. An alias is a statement of the form "`name = e`" where e is an *alias expression*, of the same syntactic form as the transformation expression τ of a rule but evaluated *once, ahead of rule execution*, rather than per tuple and per rule.

We distinguish two kinds of aliases by the form of e. In a *scalar alias*, e is an arithmetic expression over numeric constants, operators, and previously bound names (e.g., `d = 64`, `h = d/4`), which resolves to a numeric value. A *parameterized alias* introduces new parameters into the parameter set P (e.g., `p = Parameter(64)` or `A = Linear(2, 3)`). Every textual occurrence of the aliased name in a rule's transformation expression refers to the same parameters, so the alias is RelaNN's mechanism for *parameter sharing* across rules. Examples appear in line 1 of Example 1.

3.5 Function Definitions

A *function definition* (or simply a *function*) abstracts a sequence of statements into a single named statement over embedded relations. Syntactically, it takes the form

$$\texttt{def}\ F(R_1, \ldots, R_n)\ :\ \ s_1\ .\ \ldots\ s_m\ .\ \texttt{enddef}$$

where $R_1, \ldots, R_n$ are the *function parameters* standing for embedded relations supplied at each call, and the body $s_1, \ldots, s_m$ is a non-empty sequence of statements. The last statement s_m must be a rule, with its head determining the schema of the embedded relation returned by F. Intermediate statements are local to the function body. A *call* to F appears as a substitute for a relation name in a rule's body, written as $F(R_{a_1}, \ldots, R_{a_n})(\vec{x};\, z)$ with argument relations supplied positionally in the first parenthesised list and the usual content attributes $\vec{x}$ and embedding variable z in the second. Semantically, such a call is equivalent to inlining the body of F with each parameter R_i substituted by the actual argument R_{a_i}; the resulting embedded relation (produced by s_m's head) is then exposed at the call site under $\vec{x}$ and z, like any other embedded relation. Functions therefore provide *code reuse*: the same body can be applied to different input relations, without code duplication. Functions are illustrated in Example 1, with lines 2–6 including a function definition, and line 7 including a call to a function.

3.6 NRP Example

Consider a database with four embedded relations. `Drivers` and `Races` have content arity 1, each with a learned 16-dimensional embedding of a driver and of a race, respectively. `Results` has content arity 2, with a learned 16-dimensional embedding of a driver's result in a race. `Label` has content arity 1, with a 4-dimensional

```
d = 16. k = 4. Mix = Linear(2*d, d). Cls = Linear(d, k) .

def DriverProfile(Dr, Ra, Re):
  Inter(x,y; Mix(Concat(z1, z2)) + z3) :-
      Dr(x; z1), Ra(y; z2), Re(x,y; z3) .
  Out(x; sum(z)) :- Inter(x,y; z) .
enddef

Profile(x;z) :- DriverProfile(Drivers,Races,Results)(x;z) .
Loss(;CrossEntropyLoss()(Cls(z_p), z_l)) :-
    Profile(x; z_p), Label(x; z_l) .

?fit ⟨ epochs=100, lr=0.01, weight_decay=5e-4 ⟩ Loss .
?pred Profile .
```

Example 1: NRP for driver category prediction.

embedding holding a supervised category label per driver (e.g., a demographic segment). The following program computes, for each driver, a profile embedding aggregated from their results, and fits it to predict the driver's category. The first four statements are *aliases*: two scalar aliases fix the embedding width d and the number of target classes k, and two parameterized aliases allocate the shared linear maps `Mix` (pair-mixing) and `Cls` (classifier head).

The function `DriverProfile` takes `Drivers`, `Races`, and `Results` relations as arguments. For each (driver, race, result) tuple it concatenates the driver and race embeddings, applies the shared linear map `Mix`, and adds the result's embedding (rule `Inter`); it then aggregates by driver with `sum` (rule `Out`) to produce one embedding per driver. The call on the following line binds this result to `Profile`.

The `Loss` relation calculates the final loss value using cross entropy over the previously bound linear transformation `Cls` and `Label` relation. It has empty content attributes (a single distinguished row), and its embedding expression reduces each joined pair to a scalar loss by passing z_p through the shared classifier `Cls` and comparing the resulting logits against z_l.

The final statements are fit and predict. The fit is executed over the embedded relation `Loss`. It specifies the keyword-argument list ⟨`epochs=100, lr=0.01, weight_decay=5e-4`⟩ which determines how the current assignment γ of the parameter set P should be updated. Crucially, P here comprises *both* the per-tuple embeddings stored in `Drivers`, `Races`, and `Results`, *and* the weights introduced by the aliases `Mix` and `Cls`; the fit statement updates them jointly so as to minimize the loss averaged over all (driver, driver's class, label) triples. The predict statement `?pred Profile` then materializes `Profile` in the output database under the updated γ.

3.7 Language Extensions

Beyond the core rule grammar of Section 3.1, RelaNN provides two language extensions. *Encoding* and *decoding* move values between a relation's content attributes and its embedding. *Templates* factor out repeated statements and repeated body relations.

Encoding and decoding. RelaNN provides machinery for producing initial embeddings from the database and writing back the results. *Encoding* reads raw content columns into embeddings, giving a network its initial features. *Decoding* writes a learned embedding back as a content attribute, so a prediction becomes a regular column of the database. Both directions use a uniform $[\,\cdot\,]$ bracket syntax.

As an example, suppose the `Drivers` table has a `bio` column—the text of the Wikipedia page linked by each driver's `url`—and we want to learn a per-driver score and store it as a new column. The architecture *encodes* `bio` into a feature embedding, runs a network over it, and *decodes* the learned score into a new `Pred` relation:

```
DriverFeat(d_id; [TextEncoder(bio)]) :- Drivers(d_id,bio) .
// ... intermediate rules computing the score
Pred(d_id, [c]) :- Score(d_id; c) .
```

In the first rule, an *encoding* bracket inside the head's transformation expression reads the `bio` attribute into the embedding. Because `bio` is text, a rich type, the bracket wraps an encoder `nn.Module`, as in [`TextEncoder(bio)`]. Numeric and boolean attributes need no module and tensorize directly, written simply as [*col*], and multiple bracketed items concatenate along the last dimension. In the last rule, a *decoding* bracket in the head's *content* position runs the other way, writing an embedding value back as a content attribute: [*c*] decodes `Score`'s learned embedding *c* into a column of `Pred`.

Templates. Neural architectures over relational data often contain dozens of near-duplicate definitions, one per attention head, edge type, or layer. Inspired by C++, *templates* factor out this repetition: a *statement template* factors out repeated statements, and a *body template* factors out repeated body relations within a single rule. We use eight-head attention as a running example, where each head transforms an input embedding through its own learned matrix and the head outputs are combined. Written out in full, this example takes eight per-head rules and one rule that combines them:

```
AttHead1(k; Linear(d, d)(z)) :- Input(k; z) .
⋮
AttHead8(k; Linear(d, d)(z)) :- Input(k; z) .
MultiHead(k; Concat(z1, ..., z8)) :-
  AttHead1(k; z1), ..., AttHead8(k; z8) .
```

Statement templates. A *statement template* parameterizes a statement by an index in angle brackets. For the running example, this collapses the eight rules into one:

```
AttHead<i>(k; Linear(d, d)(z)) :- Input(k; z) .
```

A template is a pattern—it produces nothing until a rule *invokes* it. When `AttHead<i>` is invoked, the compiler materializes a concrete `Linear(d,d)` for each distinct argument, before term-graph construction. Arguments may range over schema type names, integers, or string labels. Invocations with identical arguments share parameters; invocations with different arguments yield independent ones.

Body templates. The combine rule still lists eight body relations. A *body template* replicates one body relation across an *iteration domain* in square brackets, either an integer range or the tuples of a relation. The ",..." replicator replicates the relation as a conjunction, collapsing the eight-way body to a single atomic expression

```
MultiHead(k; Concat(*z)) :- Att<i>(k; z) ,... [i = 1 to 8]
```

Each replica binds `i` afresh, indexing the `AttHead<i>` template, and the splat $*z$ gathers the eight head embeddings for `Concat`. The disjunctive replicator "|..." unions the replicas instead, so the rule head *adds* the heads with `sum` rather than concatenating them. Replicators expand at compile time into ordinary body relations, adding no runtime machinery. Hence, template resolution entails, conceptually, a static expansion (unrolling) over each replicator's finite iteration domain.

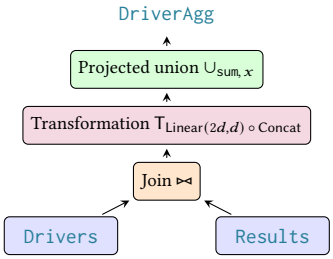

Figure 2: Term graph (logical plan) for `DriverAgg`. Leaves are database relations and inner nodes are NRA operators.

4 IMPLEMENTATION

The RelaNN pipeline (Figure 1) follows the architecture of a classic DBMS. A RelaNN program (NRP, Section 3) is translated statement-by-statement into an NRA expression, represented as a term graph (Section 4.1). When a `fit` or `predict` statement is invoked, the subgraph that produces that statement's target relation is compiled to a *physical plan* over cuDF [38] and PyTorch operations (Section 4.2) that execute joins, projections, and learned transformations end-to-end on the GPU. Data loaders then connect the corresponding database relations to the physical plan via SQL queries.

RelaNN is a Python-embedded DSL parsed with Lark [46]. Transformation operators such as `Linear` and `ReLU` are resolved by name from the caller's Python namespace, so any user-defined `nn.Module` can be referenced directly in a rule. The system is available as an open-source prototype with demos and documentation.

4.1 From RelaNN to NRA Term Graph

Each RelaNN rule translates to an NRA subexpression that captures its declarative semantics, as detailed in Section 3.1. For example, consider the rule `DriverAgg`, which combines each driver's embedding with that driver's result embeddings (Section 3.6)

```
DriverAgg(x; sum(Linear(2*d,d)(Concat(z1, z2)))) :-
    Drivers(x; z1), Results(x,y; z2) .
```

The corresponding NRA statement is

$$\texttt{DriverAgg} \;=\; \cup_{\text{sum},x}\Big(\mathsf{T}_{\text{Linear}(2d,d)} \circ \text{Concat}\,(\texttt{Drivers} \bowtie \texttt{Results})\Big)\,.$$

The compiler assembles all NRA subexpressions of a program into a single *term graph*: a DAG of NRA operators that captures every rule's semantics. Its inner nodes are NRA operators, its leaves reference the program's database relations, and its directed edges run from each operator to its consumers. The *logical query plan* for a given `fit` or `predict` statement is then the subgraph of the term graph rooted at that statement's target relation. Figure 2 illustrates the resulting term graph for `DriverAgg`.

4.2 From Term Graph to Physical Plan

The term graph defines what to compute; the physical plan defines how to execute it on a GPU. Each NRA operator compiles to a *subgraph* of physical operations. Data loaders compile to an SQL subtree that runs in the underlying database. All other operators compile to cuDF and PyTorch operations that maintain the alignment between content rows and embedding rows while preserving the gradient path through the embeddings. The core node

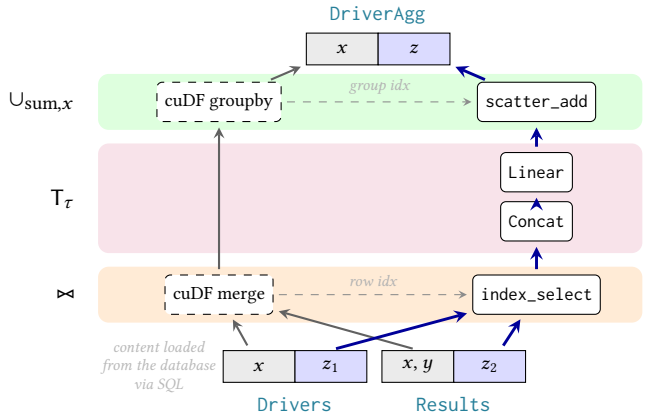

Figure 3: Physical plan for `DriverAgg`. Each NRA operator expands into cuDF operations on content and PyTorch operations on embeddings. Dashed arrows are non-differentiable index tensors, and gradients flow only along the blue chain—the differentiable path.

types compile as follows (others analogously): • Data loaders compile to an SQL subtree that executes in the underlying database, returning the *content* of an input relation as rows of a cuDF `DataFrame`. For example, a selection or projection on a database relation can be pushed into this subtree, so the database filters and prunes columns before any content reaches the GPU. Our current prototype materializes each relation in the GPU memory. Optimizations are left for future work, as we discuss in Section 6. • Join (⋈) compiles to a cuDF `DataFrame` merge that emits row-index tensors, which `torch.index_select` uses to fetch embeddings with full autograd support. • Selection (σ) uses the same differentiable `torch.index_select` with a boolean-mask index. • Projected union ($\cup_{\alpha,\mathrm{A}}$) compiles to a cuDF groupby that emits a group-index tensor. *Scatter operations* [13] (`scatter_add`, `scatter_mean`, or `scatter_max`, selected to match the aggregator α) apply the operator's multiset aggregation to the embeddings of each group. This is a sparse aggregation, reducing variable-sized groups directly over the index tensor, without materializing an adjacency matrix, and gradients flow back through the scatter primitive. • Transformation (T_τ) nodes apply their `nn.Module` (`nn.Linear`, `nn.LayerNorm`, or a custom layer) directly via a PyTorch forward call, so their parameters receive gradients automatically.

Together, the subgraphs of the NRA operations form a single computation graph that `torch.autograd` [41] traces dynamically. Any `nn.Module`, built-in or custom, can appear in a transformation and is trained end-to-end with the other operations. Figure 3 shows the physical plan compiled from the `DriverAgg` term graph.

5 EXPERIMENTS

Our experimental study supports three claims. First, RelaNN expresses a wide range of published graph-learning architectures—from classical message passing, through graph transformers, to those beyond message passing—reproducing each model's reference accuracy in a fraction of the code (Section 5.2). Second, its per-epoch runtime is competitive with hand-written reference implementations, remaining practical across these architectures (Section 5.3).

Third, we illustrate that a data owner can design a task-specific architecture directly over a multi-table database, expressing it in a handful of rules and refining it through local edits (Section 5.4).

5.1 Experimental Setup

Architectures and datasets. We evaluate RelaNN on five published architectures. • *Standard GNN architectures:* GCN [28] and R-GCN [43] cover standard and relational message passing. We compare against PyG [16] for GCN and a newer implementation [47] for R-GCN, on the Cora and AIFB datasets respectively, both from the original papers. • *Graph transformers:* We compare against heterogeneous graph transformer (HGT) [23] with both the original implementation and the newer PyG one, on the DBLP dataset [18] from the PyG HGT tutorial.[2] • *Beyond GNNs:* We also evaluate RelaNN on two architectures that go beyond the expressive power of GNNs: Deep Homomorphism Networks (DHNs) [34] and Hypergraph Neural Networks (HyGNNs) [42]. For both, we compare against the baselines and datasets from the original papers: the CSL and EXP expressivity benchmarks for DHNs, which are two synthetic datasets that ordinary message passing cannot solve, and the TWOSIDES and DrugBank drug-interaction datasets for HyGNNs.

Metrics. For each architecture, we report three metrics (Table 1): test-set accuracy, wall time of a single training epoch, and number of non-blank, non-comment lines of code. Accuracy and wall time are measured over five runs (seeds 42–46). Every baseline is a PyTorch implementation, the same framework RelaNN compiles to.

System setup. We used a single Linux server (Ubuntu 22.04.5) with two Intel Xeon CPUs, 503 GiB of RAM, and a single NVIDIA A40 GPU whose graphics and memory clocks were locked for stable timing. Software versions are PyTorch 2.5.0 and CUDA 12.4.

5.2 Implementing Known Architectures

We implemented each of the five architectures as a RelaNN program. RelaNN reproduces the reference accuracy while expressing each model in roughly 4× fewer lines of code compared to the baseline (Table 1). Beyond the model, a baseline also needs data-loading and training code, which a RelaNN reduces to loading its relations and a single `fit` call—widening the gap for a whole-program comparison.

On GCN and HGT, both initialized with the same weights as the baseline, RelaNN reaches accuracy identical to the baseline, confirming that the compiled execution is faithful to the reference. The other comparisons run from independent initialization: HyGNN lands within one standard deviation of the baseline on both datasets, and R-GCN reaches 91.7% on AIFB, a few points below the parameter-matched `torch-rgcn` reference. For DHN [34] we follow its two pattern sets and reproduce its results at both: the smaller set (C2:4) is insufficient for the task, while with the larger set (C2:10) RelaNN reaches 100% on CSL and 95% on EXP.

HGT: attention as rules. Figure 4 places HGT's attention head beside its RelaNN encoding: each of the equations defining K^i and Q^i become a single rule, and the ATT^i equation becomes two (a dot-product rule and a softmax rule). The same correspondence holds

[2] https://github.com/pyg-team/pytorch_geometric/blob/master/examples/hetero/hgt_dblp.py

Table 1: Architectures in RelaNN, each compared to its official reference (Section 5.1). Accuracy is the paper's test metric (%, mean$_{\pm\text{std}}$), Time is per-epoch wall-clock time (ms, median$_{\pm\text{std}}$), and LOC counts non-blank, non-comment lines of the NN model definition. GCN and HGT run from shared initial weights, so their accuracies match by construction.

		Accuracy (%)		Time (ms/epoch)		LOC	
Arch.	Dataset	RelaNN	Baseline	RelaNN	Baseline	RelaNN	Baseline
GCN†	Cora	$80.1_{\pm 1.6}$	$80.1_{\pm 1.6}$	$6.8_{\pm 0.3}$	$4.9_{\pm 0.4}$	21	217
HGT 1L^{a,†}	DBLP	$76.8_{\pm 2.3}$	$76.8_{\pm 2.3}$	$38.6_{\pm 4.4}$	$34.8_{\pm 1.3}$	41	186
HGT 2L^{a,†}	DBLP	$72.5_{\pm 3.8}$	$72.5_{\pm 3.8}$	$103.6_{\pm 3.5}$	$52.5_{\pm 1.0}$	42	186
pyHGT^{a,†}	DBLP	$79.5_{\pm 0.4}$	$79.5_{\pm 0.4}$	$40.4_{\pm 2.5}$	$66.6_{\pm 1.3}$	43	143
HyGNN	TwoSides	$87.7_{\pm 0.2}$	$87.8_{\pm 0.2}$	$61.9_{\pm 0.9}$	$73.5_{\pm 4.8}$	41	150
HyGNN	DrugBank	$91.9_{\pm 0.5}$	$89.9_{\pm 5.6}$	$163.5_{\pm 0.6}$	$482.5_{\pm 5.9}$	41	150
DHN (C2:4)^{b}	CSL	$22.0_{\pm 8.4}$	$28.1_{\pm 0.9}$	$45.9_{\pm 4.2}$	$232.0_{\pm 8.7}$	21	259
DHN (C2:4)^{b}	EXP	$50.0_{\pm 0.0}$	$49.0_{\pm 0.6}$	$45.8_{\pm 0.7}$	$514.5_{\pm 12.8}$	21	259
DHN (C2:10)^{b}	CSL	$100.0_{\pm 0.0}$	$100.0_{\pm 0.0}$	$275.8_{\pm 25.8}$	$561.0_{\pm 4.0}$	76	259
DHN (C2:10)^{b}	EXP	$95.0_{\pm 6.4}$	$99.9_{\pm 0.1}$	$130.3_{\pm 2.3}$	$1872.5_{\pm 50.3}$	76	259
R-GCN^{c}	AIFB	$91.7_{\pm 0.0}$	$95.0_{\pm 3.0}$	$595.4_{\pm 15.8}$	$18.7_{\pm 0.6}$	20	219

[a] RelaNN and the baseline both use the Paper⟶Author scope of HGT [23] (one edge type per layer). Full-scope PyG and pyHGT baselines, traversing all six edge types, run 1.5–4× slower. [b] The `gear/dhn` baseline runs at the configuration of DHN [34], reproducing its reported accuracy. [c] `torch-rgcn` and RelaNN both use full per-relation R-GCN weights (24M parameters, no basis decomposition), so the comparison is parameter-matched. [†] Weight initialization identical to the baseline.

across the architectures we implemented: each source equation maps to just one or two rules.

DHN: cycles as joins. DHN's core construct is graph homomorphism enumeration, which detects subgraph patterns and relates to the k-WL hierarchy [35]. This enumeration has no native layer in PyG or DGL [49], so published implementations embed it in several hundred lines of custom tensor code. RelaNN expresses it declaratively in a few rules: a triangle homomorphism becomes a cyclic self-join over edge-like relations.

```
C3_Agg(g,n; sum(
  Mu<'C3',0>(h_n) * Mu<'C3',1>(h_v) * Mu<'C3',2>(h_w)) ) :-
    Edge(g,n,v), Edge(g,v,w), Edge(g,w,n),
    H0(g,n; h_n), H0(g,v; h_v), H0(g,w; h_w) .
```

The three `Edge` body relations form the triangle, the `H0` relations fetch each matched node's previous-layer embedding, and the `Mu` transformations apply a per-position learned MLP defined with templates (Section 3.7). The `sum` aggregates over all triangle matches anchored at node `n`. This single rule is a direct transcription of the per-pattern term in DHN [34]. A full equation-to-rule listing for HGT and DHN is available in the project repository.

$$K^i(s) = \mathbf{KLin}^i_{\tau(s)}\big(H^{(L-1)}[s]\big), \quad Q^i(t) = \mathbf{QLin}^i_{\tau(t)}\big(H^{(L-1)}[t]\big)$$

$$\mathrm{ATT}^i = \mathrm{softmax}_t\Big(K^i W^{\mathrm{ATT}}_{\phi(e)} {Q^i}^{\top} \cdot \mu \,/\, \sqrt{d/h}\Big)$$

```
K(s; KLin<L,tau_s,i>(z))  :- HGT<tau_s,L-1>(s; z) .
Q(t; QLin<L,tau_t,i>(z))  :- HGT<tau_t,L-1>(t; z) .
Dot(s,t; (z_k @ W_ATT<L,phi,i> @ z_q.T) * Mu/sqrt(d/h)) :-
        K(s; z_k), phi(s,t), Q(t; z_q) .
ATT(s,t; z) :- Softmax(Dot)(s,t; z) .
```

Figure 4: Equation-to-rule correspondence for the HGT attention head [23]. Each equation (top) maps to one or two RelaNN rules. KLin **and** QLin **are the key and query layers.**

HyGNN: hyperedges as relations. HyGNN predicts drug-drug interactions by modeling each drug as a hyperedge grouping its molecular substructures (the hypergraph nodes). Standard GNN APIs represent an edge as a pair of nodes and have no first-class encoding for the many-to-one incidence between substructures and the drug they belong to. PyG does ship a generic `HypergraphConv`[3], but it implements a different message-passing scheme and is not equivalent to HyGNN's double-attention mechanism. In RelaNN, the set of (drug, substructure) incidences is an ordinary two-column relation, and message passing between a drug-hyperedge and its substructure-nodes is a natural join on the shared substructure identifier, with no special hypergraph API.

5.3 Runtime

Table 1 reports per-epoch training time. On ten of the eleven rows, the per-epoch time is close to the baseline or well below it: RelaNN stays within roughly 2× on GCN and HGT, and on the four DHN rows, it is 2–14× faster. These DHN margins partly reflect that `gear/dhn` is a research prototype rather than a tuned library such as PyG. The single pronounced outlier is R-GCN on AIFB, where our unoptimized prototype is roughly 30× slower than the specialized `torch-rgcn` baseline. This likely reflects R-GCN's per-relation weights: AIFB's many relation types induce high join fan-out and many small matrix multiplications that our plan runs as separate kernels rather than batching them as `torch-rgcn` does. Even so, these numbers already show that a fully declarative, database-style pipeline trains published architectures at a practical per-epoch cost. We leave cost-based query optimization to future work (Section 6).

5.4 Case Study: Task-Specific Architecture

The previous experiments implement existing architectures. Our case study illustrates the other direction: designing a task-specific architecture and refining it through local rule-level edits.

Task. We use four tables from the rel-f1 database [15] which are extended into embedded relations: `Drivers`/1⟨`d_dr`⟩, each driver belonging to a team in `Constructors`/1⟨`d_co`⟩, with each team taking part in `Races`/1⟨`d_ra`⟩, with one row per driver per race in `Results`/4⟨`d_re+h*2`⟩, where `d_dr`, `d_co`, `d_ra`, `h` are predefined using aliases. The rules operate on these tables directly—there is no graph to construct. The binary `driver-dnf` task predicts whether a driver receives a Did-Not-Finish in the next race period, evaluated by the area under the ROC curve (AUROC).

Initial architecture. We start from an architecture that embeds each `Results` row together with its team and race context, averages these embeddings per driver into a history, and scores the driver from its own embedding and that history. It reaches 0.610 AUROC, above a random classifier's 0.5 but below the RelBench baselines (0.686 for LightGBM, 0.726 for a GNN) [15].

Refining the architecture. The initial `History` rule averages past results with a uniform `mean`. A natural refinement lets the model weight each result, with a learned $[0, 1]$ gate:

```
WAvg = Sigmoid(Linear(d_re+h*2, 1)) .
ResultGate(eid,did; WAvg(Concat(z_r, z_c, z_ra))) :-
    Results(eid,did,rid,cid; z_r), ConsEmb(cid; z_c),
    RaceEmb(rid; z_ra) .
```

We add `ResultGate` and replace `History` to average the gated embeddings:

```
History(did; mean(z_e * z_w)) :- ResultEmb(eid,did; z_e),
    ResultGate(eid,did; z_w) .
```

This one new rule and one edited rule lift performance to 0.707 AUROC, past the LightGBM baseline and close to the GNN baseline. Because aggregation is a first-class construct, the edit needs no change to index bookkeeping, parameter allocation, or gradients: the compiler re-derives them from the edited rules.

6 CONCLUSIONS AND FUTURE WORK

By extending the relational model with learned embeddings, NRA naturally merges relational and tensor algebra, enabling deep learning models to be defined directly and elegantly over the database. We presented RelaNN, a declarative system and language over NRA. We have shown that RelaNN enables the implementation of a wide variety of SOTA neural architectures in a fraction of the code of the original implementations. While it still lags behind hand-tuned systems in training runtime, it is a major step towards making learning over relational data as simple as writing queries.

To make RelaNN useful for real-world scenarios, future work will focus on two main areas. The first is *scaling for memory*. Our prototype assumes datasets fit in GPU memory, whereas real-world databases often exceed it, calling for database sampling [8, 21, 53] and mini-batch streaming from the database to the GPU [27]. The second is *query optimization*. As a DBMS that separates specification from execution, RelaNN admits term-rewriting optimizations at three levels: 1) Relational optimizations, such as selection push-downs and join reordering, that can be delegated from the NRA operators to the database via SQL. 2) Tensor optimizations that fuse consecutive embedding operators into one kernel. 3) Hybrid rewrites that exploit the content–embedding interplay—for example, pushing a transformation before a join so it runs once per node rather than once per edge [6, 29, 39].

The recent emergence of *relational foundation models* [17, 24, 51], pretrained across many databases, promises to accelerate learning and improve accuracy. We plan to investigate the implementation of foundation models in RelaNN.

ACKNOWLEDGMENTS

This work was funded by the Israel Science Foundation (ISF), Grant 863/25.

[3] https://pytorch-geometric.readthedocs.io/en/2.5.0/generated/torch_geometric.nn.conv.HypergraphConv.html

REFERENCES

[1] Serge Abiteboul, Marcelo Arenas, Pablo Barceló, Meghyn Bienvenu, Diego Calvanese, Claire David, Richard Hull, Eyke Hüllermeier, Benny Kimelfeld, Leonid Libkin, Wim Martens, Tova Milo, Filip Murlak, Frank Neven, Magdalena Ortiz, Thomas Schwentick, Julia Stoyanovich, Jianwen Su, Dan Suciu, Victor Vianu, and Ke Yi. 2018. Research Directions for Principles of Data Management. *Dagstuhl Manifestos* 7, 1 (2018), 1–29.

[2] Linus Bao, Emily Jin, Michael M. Bronstein, İsmail İlkan Ceylan, and Matthias Lanzinger. 2025. Homomorphism Counts as Structural Encodings for Graph Learning. In *ICLR*. OpenReview.net.

[3] Thomas Bonald, Nathan de Lara, Quentin Lutz, and Bertrand Charpentier. 2020. Scikit-network: Graph Analysis in Python. *Journal of Machine Learning Research* 21, 185 (2020), 1–6. http://jmlr.org/papers/v21/20-412.html

[4] Rajesh Bordawekar and Oded Shmueli. 2017. Using Word Embedding to Enable Semantic Queries in Relational Databases. In *DEEM@SIGMOD*. ACM, 5:1–5:4.

[5] Riccardo Cappuzzo, Paolo Papotti, and Saravanan Thirumuruganathan. 2020. Creating Embeddings of Heterogeneous Relational Datasets for Data Integration Tasks. In *SIGMOD Conference*. ACM, 1335–1349.

[6] Lingjiao Chen, Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2017. Towards Linear Algebra over Normalized Data. *Proc. VLDB Endow.* 10, 11 (2017), 1214–1225.

[7] Tianlang Chen, Charilaos Kanatsoulis, and Jure Leskovec. 2025. RelGNN: Composite Message Passing for Relational Deep Learning. In *ICML*.

[8] Wei-Lin Chiang, Xuanqing Liu, Si Si, Yang Li, Samy Bengio, and Cho-Jui Hsieh. 2019. Cluster-GCN: An Efficient Algorithm for Training Deep and Large Graph Convolutional Networks. In *KDD*. ACM, 257–266.

[9] E. F. Codd. 1970. A Relational Model of Data for Large Shared Data Banks. *Commun. ACM* 13, 6 (1970), 377–387.

[10] Tamara Cucumides and Floris Geerts. 2025. From Features to Structure: Task-Aware Graph Construction for Relational and Tabular Learning with GNNs. In *Tabular Data Analysis Workshop (TaDA) at VLDB*.

[11] Alexis Cvetkov-Iliev, Alexandre Allauzen, and Gaël Varoquaux. 2023. Relational data embeddings for feature enrichment with background information. *Mach. Learn.* 112, 2 (2023), 687–720.

[12] Pedro Domingos. 2025. Tensor Logic: The Language of AI. *arXiv preprint arXiv:2510.12269* (2025).

[13] Matthias Fey. 2019. PyTorch Scatter: Optimized Scatter Operations for PyTorch. https://github.com/rusty1s/pytorch_scatter. GPU-native scatter_add, scatter_mean, scatter_max with autograd support.

[14] 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 *ICML*. OpenReview.net.

[15] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. 2024. RelBench: A Benchmark for Deep Learning on Relational Databases. In *Advances in Neural Information Processing Systems 37 (NeurIPS), Datasets and Benchmarks Track*.

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

[17] Billy Joe Franks, Moshe Eliasof, Semih Cantürk, Guy Wolf, Carola-Bibiane Schönlieb, Sophie Fellenz, and Marius Kloft. 2025. Towards Graph Foundation Models: A Study on the Generalization of Positional and Structural Encodings. *Trans. Mach. Learn. Res.* 2025 (2025).

[18] Xinyu Fu, Jiani Zhang, Ziqiao Meng, and Irwin King. 2020. MAGNN: Metapath Aggregated Graph Neural Network for Heterogeneous Graph Embedding. In *WWW*. ACM / IW3C2, 2331–2341.

[19] Boris Glavic. 2021. Data Provenance - Origins, Applications, Algorithms, and Models. *Foundations and Trends® in Databases* 9, 3-4 (2021), 209–441.

[20] Todd J. Green, Gregory Karvounarakis, and Val Tannen. 2007. Provenance semirings. In *PODS*. ACM, 31–40.

[21] William L. Hamilton, Rex Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems* (Long Beach, California, USA) *(NIPS'17)*. Curran Associates Inc., Red Hook, NY, USA, 1025–1035.

[22] Joseph M. Hellerstein, Christoper 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 (Aug. 2012), 1700–1711. https://doi.org/10.14778/2367502.2367510

[23] Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. 2020. Heterogeneous Graph Transformer. In *Proceedings of The Web Conference 2020 (WWW)*. 2704–2710. https://arxiv.org/abs/2003.01332

[24] Valter Hudovernik, Federico López, Vid Kocijan, Akihiro Nitta, Jan Eric Lenssen, Jure Leskovec, and Matthias Fey. 2026. KumoRFM-2: Scaling Foundation Models for Relational Learning. *CoRR* abs/2604.12596 (2026).

[25] Hasan M. Jamil. 2024. Toward a Declarative Query Language for Machine Learning. In *VLDB Workshops*.

[26] Matthias Jasny, Tobias Ziegler, Tim Kraska, Uwe Roehm, and Carsten Binnig. 2020. DB4ML - An In-Memory Database Kernel with Machine Learning Support. In *SIGMOD* (Portland, OR, USA). Association for Computing Machinery, New York, NY, USA, 159–173. https://doi.org/10.1145/3318464.3380575

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

[28] Thomas N. Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *ICLR*.

[29] Arun Kumar, Jeffrey Naughton, and Jignesh M. Patel. 2015. Learning Generalized Linear Models Over Normalized Data. In *SIGMOD Conference*. ACM, 1969–1984.

[30] Andreas Kunft, Asterios Katsifodimos, Sebastian Schelter, Sebastian Breß, Tilmann Rabl, and Volker Markl. 2019. An Intermediate Representation for Optimizing Machine Learning Pipelines. *Proc. VLDB Endow.* 12, 11 (2019), 1553–1567.

[31] Guoliang Li, Ji Sun, Lijie Xu, Shifu Li, Jiang Wang, and Wen Nie. 2024. Gaussml: An end-to-end in-database machine learning system. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. IEEE, 5198–5210.

[32] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. MLog: Towards Declarative In-Database Machine Learning. *Proc. VLDB Endow.* 10, 12 (2017), 1933–1936.

[33] Yuval Lev Lubarsky, Jan Tönshoff, Martin Grohe, and Benny Kimelfeld. 2023. Selecting Walk Schemes for Database Embedding. In *CIKM*. ACM, 1677–1686.

[34] Takanori Maehara and Hoang NT. 2024. Deep Homomorphism Networks. In *Advances in Neural Information Processing Systems 37 (NeurIPS)*.

[35] Haggai Maron, Heli Ben-Hamu, Hadar Serviansky, and Yaron Lipman. 2019. Provably Powerful Graph Networks. In *Advances in Neural Information Processing Systems 32 (NeurIPS)*.

[36] Sidharth Mudgal, Han Li, Theodoros Rekatsinas, AnHai Doan, Youngchoon Park, Ganesh Krishnan, Rohit Deep, Esteban Arcaute, and Vijay Raghavendra. 2018. Deep Learning for Entity Matching: A Design Space Exploration. In *SIGMOD Conference*. ACM, 19–34.

[37] Luis Müller, Mikhail Galkin, Christopher Morris, and Ladislav Rampásek. 2024. Attending to Graph Transformers. *Trans. Mach. Learn. Res.* 2024 (2024).

[38] NVIDIA. 2024. RAPIDS cuDF: GPU DataFrame Library. https://github.com/rapidsai/cudf. Pandas-compatible API with GPU acceleration.

[39] Dan Olteanu. 2020. The Relational Data Borg is Learning. *Proc. VLDB Endow.* 13, 12 (2020), 3502–3515.

[40] Paolo Papotti and Carsten Binnig. 2025. Panel on Neural Relational Data: Tabular Foundation Models, LLMs... or both? *Proc. VLDB Endow.* 18 (2025), 5513–5515.

[41] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in PyTorch. In *NIPS Autodiff Workshop*.

[42] Khaled Mohammed Saifuddin, Briana Bumgardner, Farhan Tanvir, and Esra Akbas. 2023. HyGNN: Drug-Drug Interaction Prediction via Hypergraph Neural Network. In *ICDE*. 1503–1516. https://doi.org/10.1109/ICDE55515.2023.00119

[43] 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 *ESWC*, Vol. 10843. Springer, 593–607.

[44] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *DEEM@SIGMOD*.

[45] Maximilian E. Schüle, Matthias Bungeroth, Dimitri Vorona, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. ML2SQL - Compiling a Declarative Machine Learning Language to SQL and Python. In *EDBT*.

[46] Erez Shinan. 2017. Lark: A Parsing Library for Python. https://github.com/lark-parser/lark. Accessed: 2026.

[47] Thiviyan Thanapalasingam, Lucas van Berkel, Peter Bloem, and Paul Groth. 2022. Relational Graph Convolutional Networks: A Closer Look. *PeerJ Computer Science* 8 (2022), e1073. https://doi.org/10.7717/peerj-cs.1073

[48] Jan Tönshoff, Neta Friedman, Martin Grohe, and Benny Kimelfeld. 2023. Stable Tuple Embeddings for Dynamic Databases. In *ICDE*. IEEE, 1286–1299.

[49] Minjie Wang, Lingfan Yu, Da Zheng, Quan Gan, Yu Gai, Zihao Ye, Mufei Li, Jinjing Zhou, Qi Huang, Chao Ma, Ziyue Huang, Qipeng Guo, Hao Zhang, Haibin Lin, Junbo Zhao, Jinyang Li, Alexander J. Smola, and Zheng Zhang. 2019. Deep Graph Library: Towards Efficient and Scalable Deep Learning on Graphs. *ICLR Workshop on Representation Learning on Graphs and Manifolds* (2019).

[50] Xiao Wang, Houye Ji, Chuan Shi, Bai Wang, Yanfang Ye, Peng Cui, and Philip S. Yu. 2019. Heterogeneous Graph Attention Network. In *WWW*.

[51] 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 *ICML*. PMLR / OpenReview.net.

[52] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S. Yu Philip. 2020. A comprehensive survey on graph neural networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2020), 4–24.

[53] Hanqing Zeng, Hongkuan Zhou, Ajitesh Srivastava, Rajgopal Kannan, and Viktor K. Prasanna. 2020. GraphSAINT: Graph Sampling Based Inductive Learning

Method. In *ICLR*. OpenReview.net.
[54] Jianan Zhao, Xiao Wang, Chuan Shi, Binbin Hu, Guojie Song, and Yanfang Ye. 2021. Heterogeneous Graph Structure Learning for Graph Neural Networks. In *AAAI*. AAAI Press, 4697–4705.