Predictive Query Language: A Domain-Specific Language for Predictive Modeling on Relational Databases

Vid Kocijan
NVIDIA†

Jinu Sunil
NVIDIA†

Jan Eric Lenssen
MPI for Informatics†

Viman Deb
FlowSpec†

Xinwei He
Traceroot.ai†

Federico Reyes Gomez
Tesora†

Matthias Fey
NVIDIA†

Jure Leskovec
NVIDIA, Stanford University†

ABSTRACT

The purpose of predictive modeling on relational data is to predict future or missing values in a relational database, for example, future purchases of a user, risk of readmission of the patient, or the likelihood that a financial transaction is fraudulent. Typically powered by machine learning methods, predictive models are used in recommendations, financial fraud detection, supply chain optimization, and other systems, providing billions of predictions every day. However, training a machine learning model requires manual work to extract the required training examples—prediction entities and target labels—from the database, which is slow, laborious, and prone to mistakes. Here we present the Predictive Query Language (PQL), an SQL-inspired declarative language for defining predictive tasks on relational databases. PQL allows specifying a predictive task in a single declarative query, enabling the automatic computation of training labels for a large variety of machine learning tasks, such as regression, classification, time-series forecasting, and recommender systems. PQL is already successfully integrated and used in a collection of use cases as part of a predictive AI platform. The versatility of the language can be demonstrated through its many ongoing use cases, including financial fraud, item recommendations, and workload prediction. We demonstrate its versatile design through two implementations; one for small-scale, low-latency use, and one that can handle large-scale databases.

VLDB Reference Format:
vldbauthors. Predictive Query Language: A Domain-Specific Language for Predictive Modeling on Relational Databases. VLDB 2026 Workshop: Tabular Data Analysis (TaDA).

1 INTRODUCTION

Creating a new machine learning model based on data from a relational database traditionally requires a large engineering effort. This is due to the iterative nature of ML model development, where slow iteration dramatically lengthens development cycles, driving up costs through use of scarce expert labor and expensive computing resources. It is therefore highly desirable to accelerate the model development process through simplification and automation.

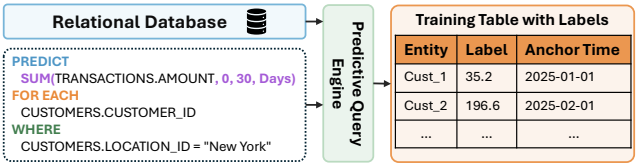

Figure 1: Predictive Query Overview. PQL allows for declaring a prediction task through a query: The **PREDICT** statement describes *what* is to be predicted and the **FOR EACH** and **WHERE** statements for *whom* the prediction is made. In this example, the query is a regression task, predicting the sum of transaction amounts over the next 30 days for all New York customers. The predictive query engine parses the query and generates a training table. The entries can be used to train a machine learning model or as in-context examples for relational foundation models.

A typical machine learning development cycle on relational databases comes with three major bottlenecks: feature engineering, training label/table generation, and model training. Input feature engineering and model training have received much attention from the research community, with recent advances in relational deep learning [4, 7, 31], foundation models [8, 13, 30], and in-context learning [24], which significantly reduce or even eliminate the need for them. However, generating training tables with training entities and their corresponding tables received much less attention.

Training table generation is a central step to machine learning on relational data. Real-world databases are dynamic: entities evolve over time, new records are appended, and historical values are updated. Constructing a correct training table, therefore, requires time travel—the ability to reason about the state of the database at a specific time—and strict point-in-time consistency, ensuring that every feature value used was available at that moment.

Failing to enforce these constraints easily leads to label or information leakage, where future information inadvertently influences the model during training, resulting in overly optimistic performance estimates that do not generalize to deployment. These issues are subtle, ubiquitous, and hard to detect, especially when feature and label engineering involves complex joins, temporal filters, and aggregations across multiple tables. As a result, the generation of

Proceedings of the VLDB Endowment ISSN 2150-8097.
†Work conducted under Kumo.ai affiliation, now part of NVIDIA.

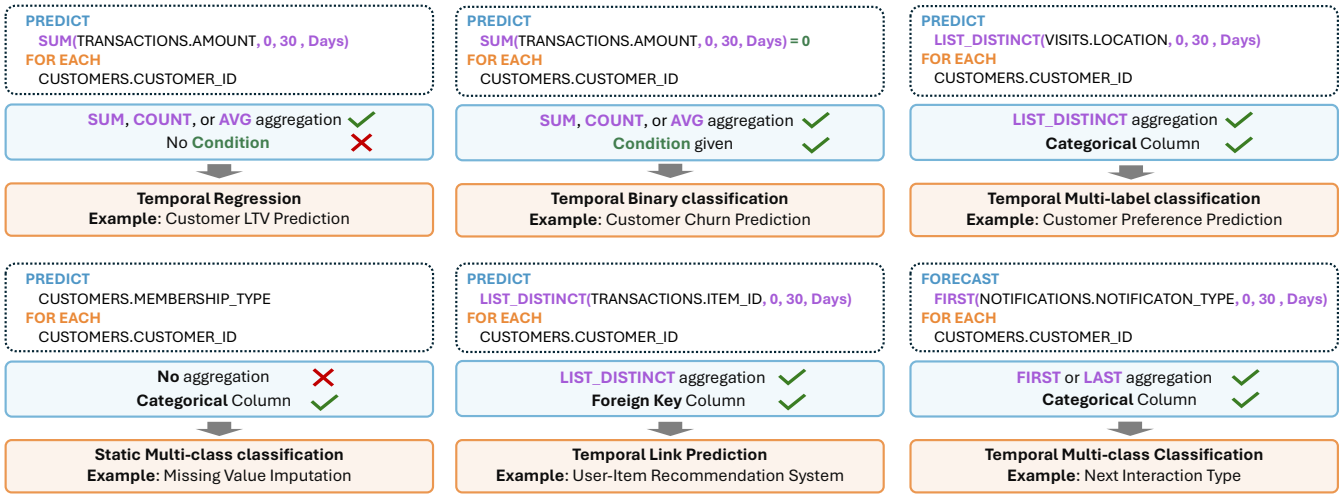

Figure 2: Taxonomy of the Predictive Query Language (PQL). Examples of predictive queries written in the proposed PQL. The language allows declaration of a wide range of prediction tasks, such as regression, classification, forecasting, and link prediction. Typical applications can be LTV prediction or forecasting, missing value imputation, churn prediction, or item recommendation for customers.

training tables is often slow, manual, and error-prone, requiring domain knowledge and careful auditing.

Here we introduce a novel domain-specific language for declarative machine learning: the *Predictive Query Language (PQL)* (Fig. 1). PQL is a declarative language for defining predictive tasks on relational databases. With PQL, a predictive task can be described in a single query, which allows automatic generation of training labels for a wide range of machine learning problems, including regression, classification, time-series forecasting, and recommendations.

```
PREDICT COUNT(TRANSACTIONS.*, 0, 3, months)
FOR EACH ARTICLES.ARTICLE_ID
WHERE ARTICLES.ARTICLE_TYPE = "shirt"
```

Figure 3: Predictive Query Example. An instance of a query to predict demand for shirts defined on the database in Fig. 4.

In PQL (Fig. 3) we can specify the predictive task in a few lines of code while enabling automatic label computation without temporal data leakage or incorrectly categorized data. PQL supports a wide range of common machine learning tasks, such as regression, classification, or link prediction (Fig. 2). Moreover, it is designed to allow easy inference with the relevant metadata such as task type, prediction timeframe, and sources of negative examples.

Given a defined set of valid entities and their corresponding targets, PQL can be used to generate training data following the Relational Deep Learning (RDL) paradigm [7]. RDL paradigm generates training instances for an ML model by looking at earlier time windows in the database. For this, *anchor times* are sampled, historical points in time for which appropriate future training label intervals exist, i.e., the 3 month intervals in Fig. 3 [7]. In the training process, one can then ensure that only the input features from before the anchor time are used to predict the corresponding target during training. We discuss details of this process in Section 3.7.

To define a set of entities, PQL supports filtering on past and present data, e.g., only predicting for customers in New York that have purchased an item in the last month, as well as making assumptions for testing counterfactuals, e.g., predicting purchases under the assumption of customer receiving a notification in the next day. To define labels and filters, PQL supports aggregations of values, e.g. `SUM` of purchased item values, or predicting new entries to the database, e.g. `LIST_DISTINCT` of foreign keys.

We present two implementations of PQL. The first is the implementation created for the Relational Deep Learning [7] framework and serves large-scale use cases where models are trained for each query. The second implementation is designed for a Relational Foundation Model [8] to answer predictive queries within a second, requiring a low-latency, interactive implementation in which PQL is used to obtain examples for in-context learning. We tested our PQL implementations on a collection of popular benchmarks for relational deep learning [32] and find that our custom implementation computes the training table up to 40-times faster compared to the baseline implementation. We find that the language is sufficiently expressive to serve multiple industry use cases, including recommendations on popular social media platforms and food delivery services, fraud detection on the Bitcoin blockchain, and detection of potential medical complications.

The rest of the paper is structured as follows: First, we discuss the related work and its shortcomings. Then, we introduce the PQL syntax and its core features. The final section is a description of the two practical implementations of the language, including a collection of optimizations specific to this problem.

2 RELATED WORK

To the best of our knowledge, there is no existing work that attempts to solve the problem of training label generation for relational deep learning. However, over the years, many systems have been introduced to simplify machine learning workflows through declarative languages and configurations. In this section, we divide them and review them according to their type and compare them with PQL.

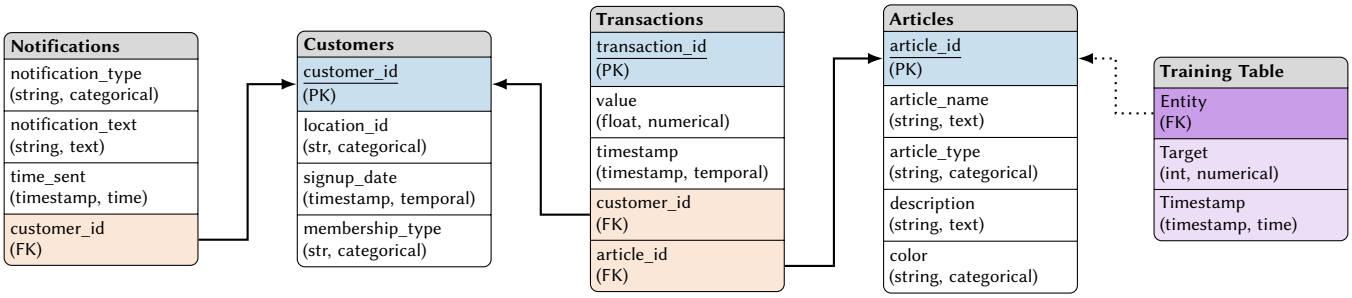

Figure 4: Example database schema. An example retail database consisting of all customers, articles, and their timestamped transactions and notifications, serving as basis for the running query examples. Each row contains information about the data type and semantic type, with links between keys marked with arrows. Primary keys (PK) are highlighted in blue and foreign keys (FK) in orange. If PQL example from Figure 3 is used on this query, the training table (violet) is added to the database, with three columns: `Entity`, pointing to the entity of the query, `Target`, denoting the label—count of the transactions, and `Timestamp`, denoting the anchor time from which the label was computed.

Configuration-Based Declarative ML. Classical approaches to declarative ML focus on translating high-level pipeline or model definition into scalable pipelines, focusing on efficiently processing input features [15, 19, 38], efficiently defining models [22, 33], or implementing ML algorithms at scale [1, 10, 21, 35]. These systems usually introduce a configuration-based declarative language that defines an abstract pipeline optimized and materialized by the system. Unlike PQL, these systems assume that labels are readily available and focus on model or data pipeline configuration. These only represent a part of the ML workflow and are increasingly sidetracked by advancements in foundation models [8] and AutoML that reduce the need for manual iteration and complex pipelines.

AutoML Platforms. Auto ML platforms and tools focus on a specific task within the ML pipeline: finding the best model for a well-defined task [5, 6, 18, 26, 37]. Using PQL and AutoML platforms is not mutually exclusive, as they ultimately tackle different problems; the latter can be used to train the model on the training table generated by the former. However, we do not use any of the existing AutoML tools in our implementations described in Section 4. The closest approach to PQL was the Feast tool [16], however, due to the limitations of the underlying models, it still heavily focused on feature development and did not provide the required full expressivity.

Integrating ML into the Database. ML2SQL [34] translates machine learning code into database-specific stored procedures, enabling end-to-end workflows within PostgreSQL. The system provides a clear separation between the data preparation and model training phases but remains limited to traditional ML algorithms. On the other hand, tools such as MAD skills [12] and SQL4ML [20] extend SQL to support machine learning operations directly within the database. These tools simplify integration of machine learning algorithms in a database environment, however, they ultimately do not reduce the ML knowledge required to train a model.

Querying trained models. GenSQL [14], ModelJoin [17], and Raven [28] introduce a syntax for querying a probabilistic model, allowing probability estimation and data generation while integrating with SQL. However, they assume that a model already exists and cannot be used to automatically create one.

These existing approaches share limitations when applied to RDL. First, their design heavily focuses on features that are not required for RDL, such as mapping data into feature vectors, i.e. feature engineering or defining a training pipeline. Second, none provides intuitive abstractions for expressing entity-centric predictions that depend on aggregations across related tables or temporal windows. As a consequence, preventing temporal leakage through overlap and extracting metadata is left to the manual efforts, making the process slow and prone to mistakes. And last, converting relational databases to graph structures suitable for modern deep learning approaches like Graph Neural Networks [2] or Graph/Relational Transformers [4, 31] requires preprocessing that these systems cannot provide. What is required is a declarative method that works natively with relational data, and our PQL is the first approach that meets this need.

3 THE PREDICTIVE QUERY LANGUAGE

This section introduces the Predictive Query Language and its core features. We define an entity- and timestamp-oriented training table and formalize the requirements such a language must satisfy. Building on these foundations, we present the syntax of PQL and illustrate its capabilities, including automatic task inference, the incorporation of assumptions about the future, and support for both static and temporal queries. Finally, we discuss how computing a single label can be repeated to construct the entire training and prediction table, along with key implementation details.

3.1 Background: Training Tables

Relational Deep Learning (RDL) [7] describes a blueprint for machine learning directly on relational databases. Given a training table whose examples consist of entities, anchor times, and target labels, RDL defines a procedure for automatically training a graph machine learning model. The training table is linked to the rest of the database by connecting its entities to the corresponding entries in an entity table, which is itself related to other tables through primary–foreign key relationships. From this structure, RDL extracts subgraphs of connected table rows, serving as input features for the given training example. The blueprint eliminates the need for feature engineering and allows training graph models on raw database data. The benefits of RDL are: (1) elimination of manual feature engineering, resulting in up to 20× faster development, and (2) improved model precision from learning raw data [32].

More formally, a *training table* in RDL is a collection of rows with the following columns (Fig. 4):

- *Entity*: A primary key that defines the entity for which prediction is made. In RDL, it links individual training examples to their corresponding subgraphs of input features. Example: A valid primary key from the customer table.
- *Target*: The target label which a machine learning model is trained to predict. Unlike the entity, the target can take any value or type, including lists when predicting multiple targets per entity. The only requirement is that all values in the column have the same type.
- *Timestamp* (optional): For *temporal* tasks, the training table includes an *anchor time*. For example, if the aim is to predict transactions of a user over the next 30 days, the *anchor time* denotes the beginning of this window and enables filtering of input features inserted after it.

In Fig. 1 we provide an example of a training table for an example query that aims to predict the spending patterns of New York based customers. Each entity or anchor time may appear multiple times; however, each pair of (entity, anchor time) is unique, defining its own corresponding label. During model training, this table is connected to the `customers` table via the `customer_id` primary key, after which we can apply RDL to train a purchase value predictor.

3.2 PQL Syntax

PQL decomposes the specification of a predictive problem on relational data into two components:

(1) `PREDICT`: Specifies the prediction target for each entity.
(2) `FOR EACH`: Specifies the set of entities for which predictions are made, along with any relevant filters.

The outline of the full grammar of PQL is provided in Fig. 5. PQL supports three core operations: aggregations, conditions, and filters. An *aggregation* aggregates values of a single column within a certain time window, grouped by a foreign key. For example,

```
PREDICT SUM(TRANSACTIONS.VALUE, 0, 30, days)
FOR EACH CUSTOMERS.CUSTOMER_ID
```

predicts a sum of all transaction values 30 days starting from the anchor timestamp, grouped by `CUSTOMER_ID`.

A *condition* is an operation with a boolean value, either a logical operation, or a comparison of an aggregation or a column with a value. Finally, a *filter*, expressed with the keyword `WHERE`, is a mechanism that selects a subset of the data that meets a condition. For example, counting the number of transactions above $100 for all customers with at least one transaction in the past 30 days:

```
PREDICT COUNT(
    TRANSACTIONS.* WHERE TRANSACTIONS.VALUE > 100,
    0, 30, days
)
FOR EACH CUSTOMERS.CUSTOMER_ID
WHERE COUNT(TRANSACTIONS.*, -30, 0, days) > 0
```

The following sections explain how these elements of the PQL grammar can be combined to form semantically meaningful queries.

```
grammar PQLGrammar;
prog:
    PREDICT target problem_type top_k
    'FOR EACH' entity assuming EOF;
// Query sections
target: condition | aggregation | column;
entity: column | filtered_column;
assuming: ('ASSUMING' condition)?;
problem_type: ('RANK' | 'CLASSIFY')?;
top_k: ('TOP' INT)?;
// operations
condition:
    aggregation REL_OP constant
    | column REL_OP constant
    | 'NOT' condition
    | condition 'AND' condition
    | condition 'OR' condition
    | '(' condition ')';
aggregation:
    AGGR_TYPE '(' (column | filtered_column) ','
                (INT | '-INF') ',' INT ')'
    | AGGR_TYPE '(' (column | filtered_column) ','
                (INT | '-INF') ',' INT ',' TIME_UNIT ')'
    | AGGR_TYPE '(' (column | filtered_column) ')';
filtered_column: column WHERE condition;
column: NAMED_COLUMN | WILDCARD_COLUMN;
constant:
    BOOL | INT | STR | DECIMAL | datetime | NULL | array;
NAMED_COLUMN: ID '.' ID;
WILDCARD_COLUMN: ID '.' '*';
AGGR_TYPE:
    'SUM' | 'AVG' | 'MIN' | 'MAX' | 'COUNT_DISTINCT'
    | 'FIRST' | 'LAST' | 'LIST_DISTINCT' | 'COUNT';
REL_OP:
    '!=' | '<=' | '>=' | '<' | '>' | '=' | 'IS' | 'IS NOT' | 'IN'
    | 'IS IN' | 'LIKE' | 'NOT LIKE' | 'CONTAINS' | 'NOT CONTAINS'
    | 'STARTS WITH' | 'ENDS WITH';
```

Figure 5: PQL grammar. The grammar is implemented in ANTLR4 syntax [29]. Parts of the grammar, including selected terminal symbols, are omitted for brevity.

3.3 Handling Static Data

Here, we provide an instance of a *static* query that predict whether missing transaction values of New York customers are over $100. It is a query without temporal aggregations where each entity is associated with exactly one label.

```
PREDICT TRANSACTIONS.VALUE > 100
FOR EACH TRANSACTIONS.TRANSACTION_ID
WHERE CUSTOMERS.LOCATION_ID = "New York"
```

The query can be used to make predictions any rows where the value of `TRANSACTION.VALUE` is missing. The rest are used for training and validation.

3.4 Handling Temporal Data

Most predictive tasks are not about filling in the gaps in the database. Instead, we usually want to predict behavior of the database in the future, e.g. predicting future purchases. To condense future entries of a database into a single label, PQL utilizes aggregations, e.g. predicting the `SUM` of all future purchases or generating a list of all purchased articles (aggregation `LIST_DISTINCT`). An example of a temporal query with aggregations is given in Fig. 6.

An aggregation must operate on a timestamp-annotated target table that is directly connected to the entity table with a foreign key. In Fig. 4, `transactions` contain a foreign key to the table `articles`, implying multiple transactions per article. Restricting the use of

```
PREDICT
   SUM(transactions.value, 15, 45, days) > 100
   OR
   COUNT(transactions.*, 15, 45, days) > 10
FOR EACH customers.customer_id
   WHERE COUNT(transactions.*, -40, 0, days) > 0
```

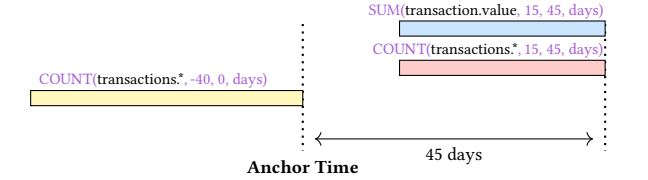

Figure 6: Timeline example. Example of a temporal predictive query that predicts whether an active user will either spend at least 100 dollars in the time period between 15 and 45 days in the future, or have at least 10 transactions. The image below illustrates the decomposition of a single timeframe that spans 85 days of data.

aggregations to tables with semantically meaningful connections in the schema is not just a syntactic convenience, but rather has a practical purpose. If a connection between keys is used in label computation, it likely carries relevant semantic information that should be explicitly included in the schema.

Note that aggregations for an individual entity are always computed relative to the anchor time. The syntax of PQL unambiguously defines the timeframe (85 days in Fig. 6), allowing the selection of anchor times that do not leak information through overlap.

3.5 Inferring the Task Type

Given a query, PQL implicitly defines its task type. Based on the data and semantic types of the referenced columns and query structure, the data and semantic type of the computed labels can be determined unambiguously. The decision process is indicated in Fig. 2 and depends on the target type. Automatic task inference is a deliberate language design choice that enables automated model selection and task-specific handling.

We give special consideration to the `LIST_DISTINCT` aggregation. Unlike most other aggregations whose output is a single value, `LIST_DISTINCT` returns a set of values, leading to a multi-label classification or a link prediction task; the latter happens when the aggregated column is a foreign key. This case requires special handling; the goal is not to predict a single value, but an existence of a future entry into the database, i.e., a future transaction. Training a link prediction model requires additional metadata that can be extracted from the query and database schema, such as negative samples and new targets that were not present in the training data, e.g., new articles.

3.6 Conditioning on the Future

The example in Fig. 7 additionally contains an `ASSUMING` clause, a future-looking filter that describes an assumption about the training data. This adds an option to express counterfactual conditions for causal inference [3]. Intuitively, it operates as a forward-looking filter applied during label-generation time.

```
PREDICT
   SUM(transactions.value, 15, 45, days) > 100
   OR
   COUNT(transactions.*, 15, 45, days) > 10
FOR EACH customers.customer_id
   WHERE COUNT(transactions.*, -40, 0, days) > 0
ASSUMING COUNT(notifications.*, 0, 15, days) > 0
```

Figure 7: Assuming example. Here, we augment the example from Fig. 6 with an assumption that a notification is sent in the next 15 days. The `ASSUMING` filter is only applied during training.

By operating as a forward-looking filter, `ASSUMING` creates a training distribution of examples that meet the assumption. In the example in Fig. 7, the assuming operator restricts the training examples to cases where the user received a notification. A machine learning model trained on such a training table will be conditioned to generate its predictions under the assumption that the notifications were sent, allowing decision making based on hypothetical scenarios. By comparing the predictions of a model obtained with the query in Fig. 7 with the query in Fig. 6, one can estimate the impact of sending a notification.

Despite acting as a filter on the training table, this operator requires separate syntax. This is both due to its semantic role and also to allow its removal when generating inputs for future predictions.

3.7 Training Table Construction

The preceding sections describe how to define the label for an individual entity or an (entity, timestamp) pair. Constructing the training table, as defined in Section 3.1, takes repeating this procedure for a set of examples. While the exact details of entity and timestamp selection are implementation-specific and can be found in Section 4, certain points are common across implementations.

For static queries, labels are computed for all entities in the entity table, wherever the label is available. The examples are split according to the training, validation, and test splits. The examples that were dropped during training due to missing labels are used as inputs during prediction time, i.e. we predict the values for entities where it could not be computed.

If the query is temporal, the timestamps corresponding to the anchor times can be selected gradually, moving back in time, as demonstrated in Fig. 8. To avoid any temporal overlap between examples in different sets that could lead to accidental information leakage, timestamps are usually selected one timeframe apart. In this way, data separation boundaries can be safely selected at any timestamp. The label is then generated for every valid pair of entities and selected timestamps. Because the number of selected timestamps directly determines the size of the training table and the computational cost, it is highly implementation- and use case–dependent, as discussed in Section 4.

During prediction time, the **`PREDICT`** and **`ASSUMING`** clauses are ignored, as labels are predicted rather than computed. The exception is the link prediction setting, such as the one given in Fig. 9 — here, the target table of the foreign key in the **`LIST_DISTINCT`** aggregation is used as the source of valid targets.

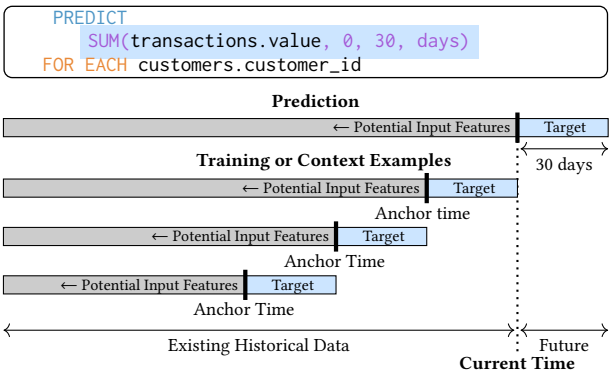

Figure 8: Anchor time sampling. Anchor times can be sampled from historical data. Each sampled anchor time defines an example, with the target is computed from data after the anchor time and input features from data before it, following the RDL blueprint [7].

```
PREDICT LIST_DISTINCT(TRANSACTIONS.ARTICLE_ID
        WHERE TRANSACTIONS.VALUE > 50
        AND ARTICLES.COLOR = "blue",
    0, 30, days)
FOR EACH CUSTOMERS.CUSTOMER_ID
```

Figure 9: Recommendation example. An instance of a recommendation query that only looks at transactions with value at least $50 and articles with blue color. The label definition is still relevant at prediction time. The condition `ARTICLES.COLOR = "blue"` is extracted and applied to the list of valid targets (table `ARTICLES`).

3.8 Implementation

The PQL parsing and validation follow the standard practices for compiler implementation [23]. Grammar parsing is implemented using ANTLR4 [29]. The obtained abstract syntax tree is validated for consistency with the database schema and transformed in a logical plan. The logical plan serves as a data platform-agnostic representation which defines the order of operations, data access, internal variables, etc., following best practices [23].

However, predictive task modeling differs from a regular querying language. It requires the following additional transformations:

- **Join Inferral**: Based on the database schema, we infer the joins and keys used for each aggregation.
- **Mismatched Timestamp Filtering**: Entities in a temporal task can come with a start and end date, e.g. user sign-up and churn date. We drop any examples with a timestamp outside of the entity validity period.
- **Target Filter Separation for Prediction**: While label computation is usually ignored during prediction time, this is not the case for recommendations. In Fig. 9, the filter within `LIST_DISTINCT` conditions on both transaction value and article color. The condition `ARTICLES.COLOR = "blue"` can be extracted and applied at prediction time to only rank articles of blue color.

Finally, the obtained logical plan undergoes implementation-specific optimization steps, described in further detail in Section 4.

```
PREDICT LIST_DISTINCT(ORDERS.STORE_ID, 0, 7, days)
  RANK TOP 12
FOR EACH USERS.CUSTOMER_ID WHERE USERS.LOC = 'NYC'
```

Figure 10: Commercial recommendation example. Predict where a NYC customer will order from; used by a delivery platform.

```
PREDICT COUNT(PURCHASES.*, 1, 4, days) > 0
FOR EACH USERS.USER_ID
WHERE USERS.NOTIFICATION_ELIGIBLE = 1
ASSUMING COUNT(
   NOTIFICATIONS.* WHERE NOTIFICATIONS.TYPE = 'PUSH',
   0, 1, days) > 0
```

Figure 11: Commercial usecase of counterfactual analysis. Predict whether a customer will make a purchase within 3 days of receiving a notification. The example illustrates the use of the `ASSUMING` operator, used to provide conditional estimation.

The optimized logical plan is translated into operations supported by data processing libraries, allowing support for multiple backends.

4 IMPLEMENTATION AND CASE STUDIES

As a general specification framework for machine learning over both static and temporal relational data, PQL has been successfully used across a range of use cases. In this section, we describe several queries used in a production setting. We then describe two implementations and discuss practical aspects of their execution.

4.1 Examples of Existing Usecases

Figures 2, 10, and 11 show example PQL queries inspired by actual queries used in the production setting. Some of these have been successfully applied to databases with over 10 billion entries.

These examples showcase the flexibility of PQL in handling prediction tasks that involve complex temporal dependencies and filters. It has been used to deploy models in domains such as:

- **Recommendation:** Predicting the top restaurants for users on a major food delivery platform.
- **Fraud Detection:** Identifying suspicious transaction patterns for a large financial institution.
- **Customer Lifetime Value (LTV):** Estimating the LTV for users of various online retail platforms.

In most of these cases, the scale exceeds billions of rows and terabytes of data, requiring a robust and scalable implementation.

4.2 Implementation: Batch Processing for Relational Deep Learning

This implementation is primarily driven by scalability requirements. Enterprise databases often contain terabytes of data with billions of rows, and training high-quality models requires leveraging as many labels as possible. As a result, scalability and robustness are prioritized over low-latency execution.

To compute labels at scale, the logical plan obtained in Section 3.8 is translated into a physical Spark plan [36] and batch-computed for every combination of entity and anchor timestamp. Anchor timestamps are usually taken one timeframe apart to make use of the full history while avoiding overlap between examples.

To utilize Spark optimizations, we avoid triggering any Spark actions until the entire plan is generated, executing it in a single large Spark query. Nevertheless, we find that in certain situations, Spark optimizations fall short, and a collection of high-level optimizations significantly improves the processing.

Avoiding a timestamp and entity cross join: Combining a collection of timestamps with all entities is equivalent to a cross join. In practice, using the `crossJoin` operation anywhere is slow as it performs unnecessary duplication and sharding. To mitigate this, we make use of the small number of timestamps (often fewer than 1000). We replace the cross join with a user-defined table function (UDTF) that does the same mapping faster. It is particularly effective when entity tables include validity intervals, such as user sign-up and churn dates, as invalid (entity, timestamp) pairs can be discarded within the UDTF (e.g., timestamps preceding the sign-up date).

Avoiding computing labels for filtered-out entities: In most production queries, we are only interested in a subset of entities defined by entity filters (e.g., `FOR EACH CUSTOMER.CUSTOMER_ID WHERE CUSTOMER.LOCATION_ID='NY'`). Ideally, all expensive operations should be restricted to this subset. However, depending on the join order, Spark may compute targets for entities that are later filtered out. To reduce this overhead, we decompose the Spark plan into four stages, materializing intermediate results to prevent recomputation:

- **Static Entity Filters** : If a query has a static entity filter that can be applied independently of timestamps, it is applied ahead of time. E.g., if the entity definition is
  ```
  FOR EACH CUSTOMERS.CUSTOMER_ID
    WHERE CUSTOMERS.LOCATION_ID = 'NY' AND
    COUNT(TRANSACTIONS.*,-10, 0, days) > 0,
  ```
  condition `CUSTOMERS.LOCATION_ID = 'NY'` is applied here.
- **Entity – Timestamp cross-join** : All remaining entities are combined with valid timestamps in a UDTF, as introduced earlier in this section.
- **Temporal Filters** All filters that depend on timestamps are applied. In the example from step 1, this means applying the filter `COUNT(TRANSACTIONS.*,-10, 0) > 0`.
- **Target Computation** Target computation is conducted last as it is the least likely step to discard rows. Rows can still be dropped in the case of malformed data or undefined aggregations (e.g., `MAX` of an empty table); however, this usually represents only a small fraction of the total rows.

Implementation Execution Time Experiments.

We benchmark these optimizations on three relational deep learning datasets of increasing scale. All experiments are implemented in Spark [36] and run on the same EMR cluster. We select queries with aggressive filters to measure the effect of entity pruning. In practice, we observe that the two contributing factors to the execution time are the entity table size and the number of timeframes.

The small-scale experiment runs on the rel-Amazon dataset, which contains Amazon reviews of book-related products [25, 32]. It contains three tables consisting of 1.8M customers, 506k products, and 20M reviews. The task is to predict whether the count of distinct ratings for products that have not received a rating above 3 in the past month is non-zero:

```
PREDICT COUNT_DISTINCT(RATINGS.RATING,0,60,days)> 0
FOR EACH PRODUCTS.PRODUCT_ID
WHERE MAX(RATINGS.RATING, -30, 0, days) < 3
```

The training table spans 88 time frames; however, only 1573 entities meet the entity filter.

Our medium-scale experiment runs on the Fannie Mae credit risk dataset[1]. It, contains 15M loans and 600M payments. The task predicts the total sum of credit payments over the next 60 days for loans with a single borrower:

```
PREDICT SUM(
    PAYMENT_HISTORY.MONTHLY_PAYMENT 0, 60, days
)
FOR EACH DIM_LOANS.LOAN_IDENTIFIER
WHERE DIM_LOANS.NUMBER_OF_BORROWERS <= 1
```

The query produces 25 timeframes of data, with 4M entities meeting the filter condition.

Finally, we run a large-scale experiment on a synthetically-upscaled H&M e-commerce dataset [32]. It contains three tables that describe 1.5M customers, 31M transactions, and 105k articles, but we scale it to 275M customers and 6.3B transactions by duplicating the entries of the customer and transaction tables. The task predicts the count of transactions in the next 7 days for all customers over the age of 30:

```
PREDICT COUNT(TRANSACTIONS.* 0, 7, days)
FOR EACH CUSTOMERS.CUSTOMER_ID
WHERE CUSTOMERS.AGE > 30
```

The query produces 70 timeframes of data with 143M entities that meet the filter conditions.

Dataset	none	X-join	Filter	both
rel-Amazon (small)	2.5	2.5	2.5	2.5
Fannie Mae (medium)	15	12	14	10
H&M (large)	timeout	115	timeout	96

Table 1: Spark execution times: Running time (in min) for three queries on datasets of increasing size. We compare four setups: no optimizations, avoiding timestamp–entity cross joins (*X-join*), avoiding computation for filtered entities (*Filter*), and both optimizations. The results show substantial gains on medium and large datasets, with the cross-join optimization essential to complete the large-scale experiment within the 10-hour timeout.

Table 1 summarizes the execution times: the unoptimized configuration fails to complete within the 10-hour limit on the largest dataset, whereas optimized variants succeed. As expected, performance gains are negligible for small datasets.

The benefits of these optimizations are twofold. First, they achieve a speedup of over 6 times or more on queries that would otherwise require extensive resources, depending on the filters and scale. Second, they allow the user to reliably speed up the label generation by adding additional filters, e.g. using only a subset of entities.

4.3 Usecase: Relational Foundation Model

The Relational Foundation Model [8] is a large pre-trained model for relational deep learning that can generate predictions in seconds.

[1]Available at https://capitalmarkets.fanniemae.com/credit-risk-transfer/single-family-credit-risk-transfer/fannie-mae-single-family-loan-performance-data

Table 2: PQL for RFM execution time. Comparison of PQL execution time for different context sizes, compared to a naive Pandas implementation. Experiments were run on the H&M dataset [32]. All running times are given in seconds, averaged over 10 random seeds and do not involve data reading. PQL implementation is significantly faster due to efficient access to the relevant subset of the database. Thus, the size of the context does not impact the run time.

	`PREDICT SUM( transactions.price, 0, 30, days ) > 0 FOR EACH customers.customer_id WHERE COUNT(transactions.*, -INF, 0) > 0`		`PREDICT COUNT( transactions.*, 0, 90, days ) > 0 FOR EACH customers.customer_id`	
#	PQL	Pandas	PQL	Pandas
10^0	0.2214	1.2526	0.0786	1.2466
10^1	0.2203	2.2603	0.0851	2.1950
10^2	0.2172	1.5917	0.0853	1.5808
10^3	0.2241	1.9744	0.0799	1.9055
10^4	0.2204	2.8926	0.0787	2.9211

The speed-up comes from the fact that this model does not perform any traditional parameter-tuning and instead extrapolates directly from context examples provided as part of the input.

The typical context window of a Relational Foundation Model contains around 10,000 examples, and the model can be used interactively. Unlike in the RDL setting, we do not generate data from all combinations of users and timestamps. Instead, the context window contains a collection of the most relevant entities and most recent timestamps. The key to achieving low-latency label computation is an efficient sampling implementation that, collects only the required rows from the otherwise large database.

To achieve such efficient sampling, we take advantage of database pre-processing, performed for the Relational Foundation Model. In this step, the database is transformed into a heterogeneous graph: rows become nodes, and primary–foreign key links become edges.

Because all aggregations and filters follow connections defined in the schema, the rows required to compute a query form a connected subgraph of the heterogeneous graph. This subgraph can be efficiently collected with a custom Python module with a C++ backend based on the GraphSAGE [9, 11].

Once the relevant rows of the database have been extracted, the execution of the logical plan from Section 3.8 is executed in Pandas [27]. In practice, the total amount of data at this stage typically does not exceed a few megabytes, requiring no sophisticated plan optimizations. In practical experiments, we find that label computation reaches sub-second latency, acceptable for a user application.

Implementation Execution Time Experiments. To assess the efficiency of the implementation, we compare the Relational Foundation Model implementation of PQL to a naive Pandas implementation. We benchmark the two on the H&M e-commerce dataset [32] which contains 2 years of clothes purchases at a major retailer with $1.3M$ customers, $105k$ articles, and $31M$ transactions. The implementations are compared on two queries; predicting the count of transactions in the next month and predicting the spend of all customers with at least one historical transaction. Unlike in the Relational Deep Learning implementation, the number of timeframes and the size of the entity table do not significantly affect the execution time. Instead, the main bottleneck is caused by the number of operations required to execute the query. To this end, we choose queries with a larger aggregation window.

The execution times are plotted in Table 2, demonstrating up to a 40-fold faster execution of our custom PQL implementation. Using a custom PQL engine for the relational foundation model is thus essential for achieving sub-1 second latency.

5 DISCUSSION AND CONCLUSION

This paper introduces PQL, a novel language for predictive modeling on relational databases. The syntax supports versatile uses across a collection of real-world predictive models and can be implemented both at-scale and in a low-latency regime. Currently, it underpins predictions within multiple popular applications, including social networks, food delivery services, and financial institutions. Future expected extensions of PQL will aim to increase the expressivity of the language through a collection of features. While we find that the existing syntax covers the majority of practical use cases, we aim to reduce the amount of pre-processing required for more involved queries. The limitations of this language can be split into two groups: the limitations arising from the assumptions and the missing functionality that we aim to add in future iterations.

Through practical use of PQL on multiple real-world databases we found that some of them do not follow best practices of database construction. For example, if records are inserted without the appropriate timestamp annotation or are modified retroactively without an appropriate timestamped record, the procedure of generating training data from the past records is bound to produce inaccurate labels. In majority of such cases, we found that additional manual pre-processing and data cleaning can make PQL and RDL applicable. Unfortunately, such pre-processing does not only present additional labor, but also a model accuracy reduction as otherwise valuable data needs to be avoided to avoid information leakage.

Even though the existing PQL syntax covers most of the problems we have encountered in practical applications, we find that additional syntax could further improve its usability:

- **Explicit Join Support**: Explicit syntax for data joins would allow queries where label is computed from data that does not sit in a table directly linked to the entity table. Moreover, it provides verbosity where tables are connected with more than one foreign key.
- **Operations between Columns and Aggregations:** In current syntax, only comparisons between a column/aggregation and a constant is permitted. Addressing this limitation is planned for future iterations of PQL.
- **Non-Atomic Primary Key Support**: In the current language design, each prediction corresponds to exactly one entity, defined by a primary key. Certain usecases do not fit within this assumption, e.g. when a prediction is made for a pair of entities. These require additional pre-processing where pairs are introduced as an auxiliary table. This effort could be avoided through an enhanced syntax in the `PREDICT` clause.

The ongoing deployments and active users of PQL demonstrate that, combined with RDL and Relational Foundation Models, it unlocks rapid model development, paving the way to faster development and novel use cases.

REFERENCES

[1] Matthias Boehm, Michael W. Dusenberry, Deron Eriksson, Alexandre V. Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R. Reiss, Prithviraj Sen, Arvind C. Surve, and Shirish Tatikonda. 2016. SystemML: declarative machine learning on spark. *Proc. VLDB Endow.* 9, 13 (Sept. 2016), 1425–1436. https://doi.org/10.14778/3007263.3007279

[2] Tianlang Chen, Charilaos Kanatsoulis, and Jure Leskovec. 2025. Relgnn: Composite message passing for relational deep learning. *arXiv preprint arXiv:2502.06784* (2025).

[3] Victor Chernozhukov, Christian Hansen, Nathan Kallus, Martin Spindler, and Vasilis Syrgkanis. 2024. Applied Causal Inference Powered by ML and AI. arXiv:2403.02467 [econ.EM] https://arxiv.org/abs/2403.02467

[4] Vijay Prakash Dwivedi, Sri Jaladi, Yangyi Shen, Federico López, Charilaos I Kanatsoulis, Rishi Puri, Matthias Fey, and Jure Leskovec. 2026. Relational Graph Transformer.

[5] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. 2020. Autogluon-tabular: Robust and accurate automl for structured data. *arXiv preprint arXiv:2003.06505* (2020).

[6] Matthias Feurer, Aaron Klein, Katharina Eggensperger, Jost Tobias Springenberg, Manuel Blum, and Frank Hutter. 2015. Efficient and robust automated machine learning. In *Proceedings of the 29th International Conference on Neural Information Processing Systems - Volume 2* (Montreal, Canada) *(NIPS'15)*. MIT Press, Cambridge, MA, USA, 2755–2763.

[7] Matthias Fey, Weihua Hu, Kexin Huang, Jan Eric Lenssen, Rishabh Ranjan, Joshua Robinson, Rex Ying, Jiaxuan You, and Jure Leskovec. 2023. Position: Relational Deep Learning - Graph Representation Learning on Relational Databases. In *Forty-first International Conference on Machine Learning*.

[8] Matthias Fey, Vid Kocijan, Federico Lopez, Jan Eric Lenssen, and Jure Leskovec. 2025. KumoRFM: A Foundation Model for In-Context Learning on Relational Data. https://kumo.ai/research/kumo_relational_foundation_model.pdf

[9] 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. arXiv:2507.16991 [cs.LG] https://arxiv.org/abs/2507.16991

[10] Amol Ghoting, Rajasekar Krishnamurthy, Edwin Pednault, Berthold Reinwald, Vikas Sindhwani, Shirish Tatikonda, Yuanyuan Tian, and Shivakumar Vaithyanathan. 2011. SystemML: Declarative machine learning on MapReduce. In *Proceedings of the 2011 IEEE 27th International Conference on Data Engineering (ICDE '11)*. IEEE Computer Society, Washington, DC, USA, 231–242. https://doi.org/10.1109/ICDE.2011.5767930

[11] 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.

[12] 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

[13] Noah Hollmann, Samuel Müller, Lennart Purucker, Arjun Krishnakumar, Max Körfer, Shi Bin Hoo, Robin Tibor Schirrmeister, and Frank Hutter. 2025. Accurate predictions on small data with a tabular foundation model. *Nature* (09 01 2025). https://doi.org/10.1038/s41586-024-08328-6

[14] Mathieu Huot, Matin Ghavami, Alexander K. Lew, Ulrich Schaechtle, Cameron E. Freer, Zane Shelby, Martin C. Rinard, Feras A. Saad, and Vikash K. Mansinghka. 2024. GenSQL: A Probabilistic Programming System for Querying Generative Models of Database Tables. *Proc. ACM Program. Lang.* 8, PLDI, Article 179 (June 2024), 26 pages. https://doi.org/10.1145/3656409

[15] James Kanter and Kalyan Veeramachaneni. 2015. Deep feature synthesis: Towards automating data science endeavors. 1–10. https://doi.org/10.1109/DSAA.2015.7344858

[16] James Max Kanter, Owen Gillespie, and Kalyan Veeramachaneni. 2016. Label, Segment, Featurize: A Cross Domain Framework for Prediction Engineering. In *2016 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*. 430–439. https://doi.org/10.1109/DSAA.2016.54

[17] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. 2023. Exploration of Approaches for In-Database ML. In *International Conference on Extending Database Technology*. https://api.semanticscholar.org/CorpusID:253270276

[18] Tim Kraska, Ameet Talwalkar, John C Duchi, Rean Griffith, Michael J Franklin, and Michael I Jordan. 2013. MLbase: A Distributed Machine-learning System.. In *Cidr*, Vol. 1. 2–1.

[19] Hoang Thanh Lam, Johann-Michael Thiebaut, Mathieu Sinn, Bei Chen, Tiep Mai, and Oznur Alkan. 2017. One button machine for automating feature engineering in relational databases. *arXiv preprint arXiv:1706.00327* (2017).

[20] Nantia Makrynioti, Ruy Ley-Wild, and Vasilis Vassalos. 2019. sql4ml A declarative end-to-end workflow for machine learning. arXiv:1907.12415 [cs.DB] https://arxiv.org/abs/1907.12415

[21] Akshay Naresh Modi, Chiu Yuen Koo, Chuan Yu Foo, Clemens Mewald, Denis M. Baylor, Eric Breck, Heng-Tze Cheng, Jarek Wilkiewicz, Levent Koc, Lukasz Lew, Martin A. Zinkevich, Martin Wicke, Mustafa Ispir, Neoklis Polyzotis, Noah Fiedel, Salem Elie Haykal, Steven Whang, Sudip Roy, Sukriti Ramesh, Vihan Jain, Xin Zhang, and Zakaria Haque. 2017. TFX: A TensorFlow-Based Production-Scale Machine Learning Platform. In *KDD 2017*.

[22] Piero Molino, Yaroslav Dudin, and Sai Sumanth Miryala. 2019. Ludwig: a type-based declarative deep learning toolbox. arXiv:1909.07930 [cs.LG] https://arxiv.org/abs/1909.07930

[23] Steven S. Muchnick. 1998. *Advanced compiler design and implementation*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[24] Samuel Müller, Simon Kornblith, and Ioannis Pitas. 2022. Transformers Can Do Bayesian Inference. In *International Conference on Learning Representations*.

[25] Jianmo Ni, Jiacheng Li, and Julian McAuley. 2019. Justifying Recommendations using Distantly-Labeled Reviews and Fine-Grained Aspects. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing and the 9th International Joint Conference on Natural Language Processing (EMNLP-IJCNLP)*, Kentaro Inui, Jing Jiang, Vincent Ng, and Xiaojun Wan (Eds.). Association for Computational Linguistics, Hong Kong, China, 188–197. https://doi.org/10.18653/v1/D19-1018

[26] Randal S. Olson and Jason H. Moore. 2016. TPOT: A Tree-based Pipeline Optimization Tool for Automating Machine Learning. In *Proceedings of the Workshop on Automatic Machine Learning (Proceedings of Machine Learning Research)*, Frank Hutter, Lars Kotthoff, and Joaquin Vanschoren (Eds.), Vol. 64. PMLR, New York, New York, USA, 66–74. https://proceedings.mlr.press/v64/olson_tpot_2016.html

[27] The pandas development team. 2020. *pandas-dev/pandas: Pandas*. https://doi.org/10.5281/zenodo.3509134

[28] Kwanghyun Park, Karla Saur, Dalitso Banda, Rathijit Sen, Matteo Interlandi, and Konstantinos Karanasos. 2022. End-to-end optimization of machine learning prediction queries. In *Proceedings of the 2022 International Conference on Management of Data*. 587–601.

[29] Terence Parr. 2013. *The Definitive ANTLR 4 Reference* (2nd ed.). Pragmatic Bookshelf.

[30] Jingang Qu, David Holzmüller, Gaël Varoquaux, and Marine Le Morvan. 2025. TabICL: A Tabular Foundation Model for In-Context Learning on Large Data. *arXiv preprint arXiv:2502.05564* (2025).

[31] Rishabh Ranjan, Valter Hudovernik, Mark Znidar, Charilaos Kanatsoulis, Roshan Upendra, Mahmoud Mohammadi, Joe Meyer, Tom Palczewski, Carlos Guestrin, and Jure Leskovec. 2026. Relational Transformer: Toward Zero-Shot Foundation Models for Relational Data. (April 2026).

[32] Joshua Robinson, Rishabh Ranjan, Weihua Hu, Kexin Huang, Jiaqi Han, Alejandro Dobles, Matthias Fey, Jan E. Lenssen, Yiwen Yuan, Zecheng Zhang, Xinwei He, and Jure Leskovec. 2024. RelBench: A Benchmark for Deep Learning on Relational Databases. arXiv:2407.20060 [cs.LG] https://arxiv.org/abs/2407.20060

[33] Christopher Ré, Feng Niu, Pallavi Gudipati, and Charles Srisuwananukorn. 2019. Overton: A Data System for Monitoring and Improving Machine-Learned Products. In *CIDR*. https://arxiv.org/pdf/1909.05372.pdf

[34] Maximilian E. Schüle, Matthias Bungeroth, Alfons Kemper, Stephan Günnemann, and Thomas Neumann. 2019. MLearn: A Declarative Machine Learning Language for Database Systems. In *Proceedings of the 3rd International Workshop on Data Management for End-to-End Machine Learning* (Amsterdam, Netherlands) *(DEEM'19)*. Association for Computing Machinery, New York, NY, USA, Article 7, 4 pages. https://doi.org/10.1145/3329486.3329494

[35] Evan R Sparks, Shivaram Venkataraman, Tomer Kaftan, Michael J Franklin, and Benjamin Recht. 2017. Keystoneml: Optimizing pipelines for large-scale advanced analytics. In *2017 IEEE 33rd international conference on data engineering (ICDE)*. IEEE, 535–546.

[36] The Apache Software Foundation. 2025. *SparkR: R Front End for 'Apache Spark'*.

[37] Chris Thornton, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2013. Auto-WEKA: Combined selection and hyperparameter optimization of classification algorithms. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*. 847–855.

[38] Doris Xin, Litian Ma, Jialin Liu, Stephen Macke, Shuchen Song, and Aditya Parameswaran. 2018. Helix: Accelerating human-in-the-loop machine learning. *arXiv preprint arXiv:1808.01095* (2018).