Mitigating the Impedance Mismatch between Prediction Query Execution and Database Engine

CHENYANG ZHANG, East China Normal University[†], China
JUNXIONG PENG, East China Normal University[†], China
CHEN XU[*], East China Normal University[†], China
QUANQING XU, OceanBase, AntGroup, China
CHUANHUI YANG, OceanBase, AntGroup, China

Prediction queries that apply machine learning (ML) models to perform analysis on data stored in the database are prevalent with the advance of research. Current database systems introduce Python UDFs to express prediction queries and call ML frameworks for inference. However, the impedance mismatch between database engines and prediction query execution imposes a challenge for query performance. First, the database engine is oblivious to the internal semantics of prediction functions and evaluates the UDF holistically, which incurs the repetitive inference context setup. Second, the invocation of prediction functions in the database does not consider that batching inference with a desirable inference batch size achieves a high performance in ML frameworks. To mitigate the mismatch, we propose to employ a prediction-aware operator in database engines, which leverages *inference context reuse cache* to achieve an automatic one-off inference context setup and *batch-aware function invocation* to ensure desirable batching inference. We implement a prototype system, called IMBridge, based on an open-source database OceanBase. Our experiments show that IMBridge achieves a 71.4x speedup on average over OceanBase for prediction query execution and significantly outperforms other solutions.

CCS Concepts: • **Information systems** → **Query optimization**.

Additional Key Words and Phrases: Query Optimization and Processing, User-Defined Functions, Machine Learning Prediction

ACM Reference Format:
Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2025. Mitigating the Impedance Mismatch between Prediction Query Execution and Database Engine. *Proc. ACM Manag. Data* 3, 3 (SIGMOD), Article 189 (June 2025), 28 pages. https://doi.org/10.1145/3725326

1 Introduction

After decades of development, machine learning (ML) is playing a crucial role in modern data analyses. Prediction queries are a kind of analytical processing (OLAP) that utilizes ML models to draw new insights, and discover latent patterns. In the scenario of enterprise applications,

[†]Engineering Research Center of Blockchain Data Management (East China Normal University), Ministry of Education
[*]Chen Xu is the corresponding author

Authors' Contact Information: Chenyang Zhang, East China Normal University, Shanghai, China, cyzhangecnu@stu.ecnu.edu.cn; Junxiong Peng, East China Normal University, Shanghai, China, jxpeng@stu.ecnu.edu.cn; Chen Xu, East China Normal University, Shanghai, China, cxu@dase.ecnu.edu.cn; Quanqing Xu, OceanBase, AntGroup, Hangzhou, China, xuquanqing.xqq@oceanbase.com; Chuanhui Yang, OceanBase, AntGroup, Hangzhou, China, rizhao.ych@oceanbase.com.

ACM 2836-6573/2025/6-ART189
https://doi.org/10.1145/3725326

```
CREATE PYTHON_UDF predict(data)
RETURNS INTEGER {
  # Inference context setup
  import pickle, joblib, pypmml.Model
  import numpy as np
  from sklearn.preprocessing import StandardScaler
  from sklearn.preprocessing import OneHotEncoder
  from sklearn.tree import DecisionTreeClassifier
  with open('/path/to/scaler', 'rb') as f:
    scaler = pickle.load(f)
  with open('/path/to/one_hot', 'rb') as f:
    enc = joblib.load(f)
  with open('/path/to/dt_model', 'rb') as f:
    model = pypmml.Model.fromFile(f)
  # Data preprocessing
  data = np.column_stack(data)
  numerical, categorical = np.split(data, np.array([8]), 1)
  X = np.hstack((scaler.transform(numerical),
  enc.transform(categorical).toarray()))
  # Model inference invocation
  return model.predict(X)
}
```

(a) Prediction Function Definition

```
SELECT count(h.prop_id)
FROM (listings l JOIN hotels h ON l.prop_id=h.prop_id)
JOIN searches s ON l.srch_id=s.srch_id
WHERE h.prop_starrating > 2 AND predict(data) = TRUE;
```

(b) Query Data with the Prediction Function

Fig. 1. An Example of Prediction Query

data is typically stored in relational databases, such as OceanBase, PostgreSQL, and SQL Server, deployed on commodity servers considering stability and cost-effectiveness [48, 59, 80, 81]. In terms of performance and data compliance, supporting prediction queries inside the database has great significance [1, 25, 28, 29] at present. According to the report of Kaggle [31], Python and SQL remain the two most common programming languages for data scientists. It is common for database engines to provide Python User-Defined Functions (UDFs) to express various data science tasks like ML inference [19, 20, 23, 35, 36]. In this work, we mainly focus on the performance enhancement for prediction queries in relational database engines with Python UDFs.

Suppose a data scientist of an online travel agency applies prediction queries on the business data stored inside databases for decision-making. She first collects historical data and applies the scikit-learn library to train a Decision Tree (DT) based machine learning pipeline to classify whether a hotel will be recommended. Then, as shown in Figure 1(a), to deploy the trained DT model into the database query engine, she specifies the prediction function definition statement. The statement includes a Python prediction function generally consisting of inference context setup, data preprocessing, and model inference invocation. After that, she can conduct predictive analysis by submitting query statements with prediction function calls. The example as shown in Figure 1(b) denotes a query counting how many hotels with a star rating of more than two will

be recommended to clients. This query first collects satisfied data from past search listings, hotel information, and search events through joining and filtering, then feeds the data to the `predict` function to determine whether a hotel will be recommended, and finally aggregates the result.

In relational database engines, the prediction query is transformed into an operator tree for execution, and prediction functions are embedded in operators as UDF expressions. As data processed is typically too large to fit into memory, the query executor drives each operator to process data from disk iteratively in a batch-wise paradigm [15, 49, 53, 81]. For every iteration, each operator fetches a batch of data from downstream operators, executes the relational operation, and finally invokes the UDF for evaluation. In contrast to traditional SQL queries, the performance bottleneck of prediction queries is typically in the computation process of the prediction function [36]. In fact, running the prediction query in Figure 1(b) on OceanBase is slow, and the prediction function part occupies 95% of the entire execution time.

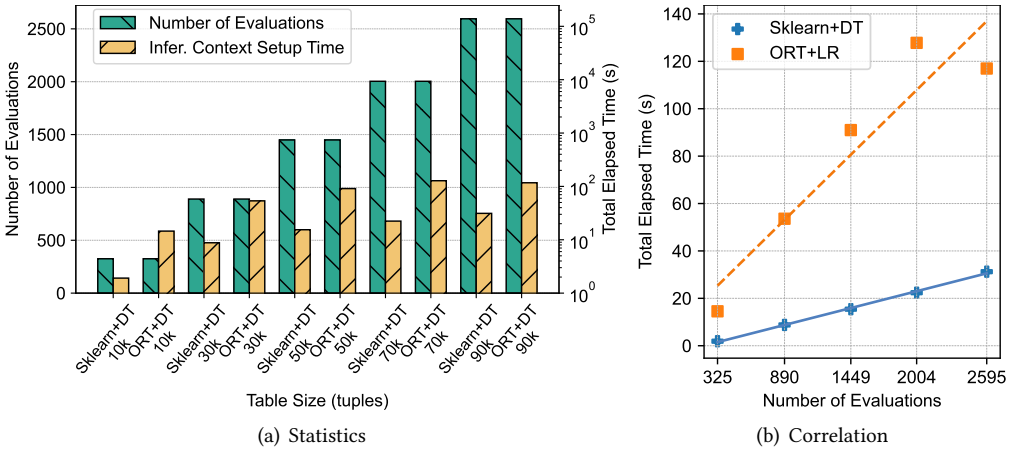
In particular, there is a bottleneck incurred by impedance mismatches between prediction query execution and database engines. First, most database engines handle prediction functions as ordinary UDFs and evaluates the prediction function without noting the internal application semantics. For example, Froid [66], UDO [73] and Tuplex [75] utilize compilation techniques to translate the UDF into high-performance codes as a whole. Babelfish [23] and Velox [60] introduce unified data representation to accelerate data exchange, which is agnostic to contents in UDFs. Despite their performance improvements, they do not consider the prediction-specific characteristic in the UDFs. This results in the identical inference context setup process inside the prediction function being executed repetitively. Second, recent works [35, 64, 82] typically apply vectorized UDF execution to reduce the overhead of language runtime switch between database engines and UDFs. Here, the prediction function as UDF expression is invoked with a batch according to the batch size variation pattern in database engines, rather than a desirable inference batch size which achieves a high processing speed in ML frameworks. Consequently, it still leaves room for performance improvement on UDFs in prediction queries.

To mitigate the impedance mismatch between prediction query execution and database engines, we propose to employ a prediction-aware operator in the database and implement a prototype system, namely IMBridge, in OceanBase [55, 80, 81], an open-source database system. To avoid repetitive inference context setup, rather than asking users to rewrite the UDF manually, IMBridge leverages *inference context reuse cache*. The cache stores the inference context after the first call of ML APIs in prediction functions and automatically reuses the context for subsequent API calls. Hence, IMBridge achieves an automatic one-off inference context setup with a 18x performance improvement on average in comparison to OceanBase. Moreover, to maximize the benefit of batching inference in ML frameworks, IMBridge introduces *batch-aware function invocation*. In particular, IMBridge exploits a batch controller to specifically match the UDF invocation batch size to the desirable inference batch size. By doing so, it realizes a desirable batching inference with a further 3.96x speedup on average. Our demonstration [84] and extensive experiments [1] in this paper show that IMBridge accelerates the prediction query execution 71.4x on average over OceanBase, and outperforms alternative solutions on PG PL/Python [61], DuckDB [16, 65], MADlib [25], and EvaDB [79], by orders of magnitude.

In the remainder of this paper, we highlight the motivations for IMBridge in Section 2 and make the following contributions.

- We propose runtime *inference context reuse cache* in Section 3, to achieve automatic one-off inference context setup.

[1] Open source at https://github.com/IM-DM4AI/IMBridge_exp



(a) Statistics (b) Correlation

Fig. 2. Repetitive Inference Context Setup

- We propose *batch-aware function invocation* in Section 4, which enables desirable batching inference on prediction functions.
- We discuss the implementation of IMBridge in OceanBase as well as the extension to other database engines in Section 5.
- Our experimental studies in Section 6 demonstrate IMBridge significantly outperforms the state-of-the-art solutions.

In addition, we introduce related works in Section 7 and conclude our work in Section 8.

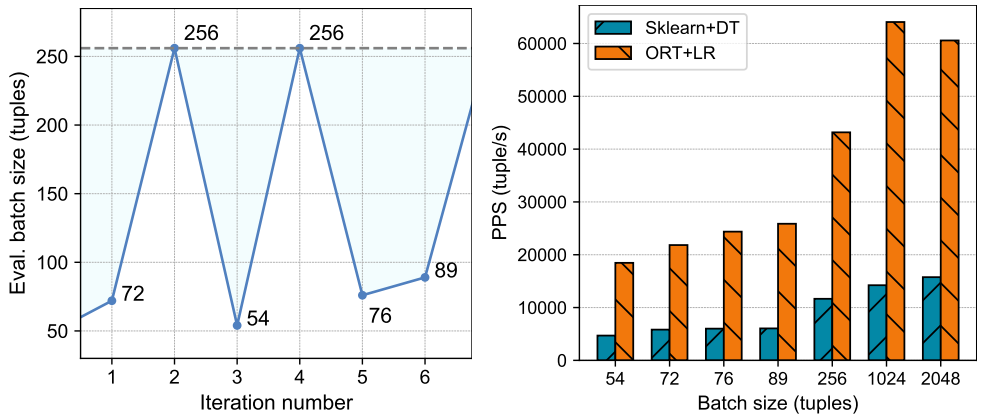
2 Motivation

We highlight the motivation for one-off inference context setup and desirable batching inference in Section 2.1 and 2.2 respectively.

2.1 Repetitive Inference Context Setup

In contrast to traditional relational queries, the data processing logic in prediction queries is scattered between relational operators and prediction functions. Since there are semantic differences between relational computation and ML inference, each part processes data in its own context. While the context of relational computation is maintained by the database engines, UDF-based prediction functions contain an indispensable routine of setting up the inference context to enable performing predictions under this context. Typically, the inference context setup routine is implemented by users through a set of predefined application program interfaces (APIs) provided by existing ML frameworks [46]. The APIs generally include importing library dependencies, loading trained models from storage, constructing the model object in memory, and allocating runtime resources. For instance, the inference context setup of the `predict` function in Figure 1(a) refers to importing ML frameworks and loading preprocessors and models from external files through the `pickle`, `joblib`, and `pypmml` library. Given the identical parameters, the inference context setup code in Figure 1(a) is deterministic since it performs the same operation and returns the same context.

We take an experiment to demonstrate the overhead of inference context setup in prediction query execution. We run the prediction query of Figure 1(b) on the Expedia dataset [38] in OceanBase with various table sizes and deploy the Decision Tree (DT) model with scikit-learn (Sklearn) and the



(a) FIBS Variation

(b) The impact of Batch Size to *PPS*

Fig. 3. Undesirable Batching Inference

Logistic Regression (LR) model with ONNX Runtime (ORT) in two prediction functions respectively. For each experiment, we log the number of prediction function evaluations and the total elapsed time of inference context setup. As shown in Figure 2(a), we observe that both the number of prediction function evaluations and the total elapsed time of setting up the inference context in the database increase as the table size scales up. In particular, Figure 2(b) illustrates that the total elapsed time of inference context setup displays a positive correlation relative to the number of prediction function evaluations.

Due to the deterministic and identical context setup codes, there is a repetitive inference context setup when evaluating prediction functions in the current database engine, which results in significant overhead. It motivates us to design a prediction-aware operator for UDF evaluation to eliminate the overhead of repetitive inference context setup, so as to achieve a *one-off inference context setup* and accelerate the execution of prediction queries.

2.2 Undesirable Batching Inference

The efficiency of prediction function execution, which is essential for prediction queries, is determined by its processing speed. Given a specific prediction function PF, we define its *prediction processing speed* $PPS(PF)$ as the number of tuples that complete the inference per second. Most ML frameworks are optimized for batch processing [12, 13] since batching increases the processing speed by improving resource utilization and exploiting existing high-performance computation techniques such as BLAS libraries, SIMD instructions, and GPU acceleration. In general, the input batch size affects the $PPS(PF)$ to a different degree.

When running prediction queries as mentioned in Section 2.1, the query engine of OceanBase invokes the prediction function according to the *UDF invocation batch size* (FIBS), i.e., the batch size of data tuples input to the UDF expression for computation. Figure 3(a) illustrates the FIBS for the prediction function in the first to the sixth iteration, as an example, during the iterative execution process in the vectorized iterator model [34] of OceanBase. Although OceanBase sets 256 as the default runtime vector size in the system-level configuration, the FIBS still fluctuates frequently during execution. The database system usually invokes the prediction function with the FIBS less than 256. On the other hand, we measured the impact of batch size to *PPS* with the aforementioned models. As depicted in Figure 3(b), compared to the FIBS that appeared in database

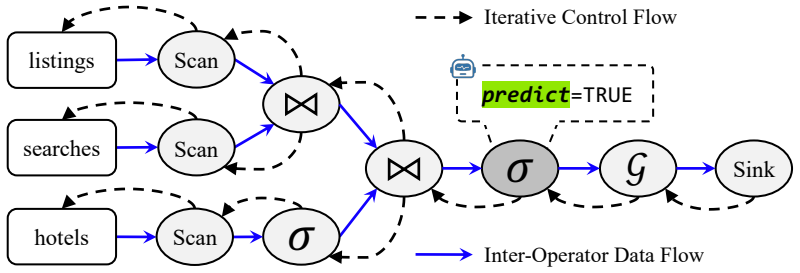


Fig. 4. Prediction Query Execution Plan of Listing 1

engines, the *PPS* with a batch size of 2048 for DT and 1024 for LR achieve a higher *PPS* respectively. Consequently, the batch size applied in the database engine leads to an inefficient execution of the prediction function.

Hence, there is an undesirable batching inference problem when invoking prediction functions, which results in a low processing speed. It motivates us to design a prediction-aware operator for inference execution to match the FIBS to a batch size with a high *PPS*, so as to realize a *desirable batching inference* and reduce the execution time of prediction queries.

3 One-off Inference Context Setup

In this section, we introduce manual inference context reuse in Section 3.1. We propose automatic inference context reuse to achieve the one-off inference context setup in Section 3.2.

3.1 Manual Inference Context Reuse

Given an in-database predictive analytical workload, the prediction query is expressed using SQL, and prediction functions are implemented as Python UDFs. We take the execution plan of Figure 1(b) as an example to elaborate on the general characteristics of prediction query execution in database engines. When the SQL and the prediction function are submitted to the query engine, they will be transformed into an execution plan. As shown in Figure 4, the execution plan is generally a tree of operators and can be split into two parts: the relational and prediction parts.

The relational part is the operator nodes evaluation process, which also controls the data flow. There are two prevalent implementation paradigms for query execution in database engines, i.e., pull-based and push-based paradigms [71, 77]. Specifically, both paradigms realize inter-operator control by implementing common control interfaces. At runtime, the query engine drives the cascade operators iteratively for computation. For instance, the pull-based paradigm is widely used in popular database engines like PostgreSQL and OceanBase. Operators in this paradigm are implemented as nested iterators. Each operator refers to its child nodes and calls the `next()` method to pull the data until the `has_next()` method returns false. During execution, the query engine iteratively calls the `next()` method of the root node, e.g., the sink node in Figure 4, to get query results until the tuples returned from downstream operators are exhausted.

The prediction part is the UDF expression evaluation process embedded in a certain operator, e.g., `predict` in the selection operator, which performs specific computations following the iterative control flow of operators. The prediction function codes include inference context setup, data preprocessing, and performing the ML inference. For each evaluation, the inference context is set up at the beginning of the prediction function. This routine makes preparations for the inference environment and returns a series of context variables after calling the APIs provided by ML

```
CREATE PYTHON_UDF predict(data)
RETURNS INTEGER {
  # Execute at the PF Initialization Stage
  def __init__(context):
    # Inference context setup
    import pickle, joblib, pypmml.Model
    import numpy as np
    from sklearn.preprocessing import StandardScaler
    from sklearn.preprocessing import OneHotEncoder
    from sklearn.tree import DecisionTreeClassifier

    with open('/path/to/scaler.pkl', 'rb') as f:
      context.scaler = pickle.load(f)
    with open('/path/to/one_hot_encoder.pkl', 'rb') as f:
      context.enc = joblib.load(f)
    with open('/path/to/dt_model.pkl', 'rb') as f:
      context.model = pypmml.Model.fromFile(f)

  # Execute at the PF Computation Stage
  def __call__(context, data):
    # Data preprocess
    data = np.column_stack(data)
    numerical, categorical = np.split(data, np.array([8]), 1)
    X = np.hstack((context.scaler.transform(numerical),
      context.enc.transform(categorical).toarray()))
    # Invoke model inference
    return context.model.predict(data)
}
```

Fig. 5. Staged Prediction Function Evaluation

frameworks [46]. Then, the rest of the prediction function program can perform other operations by using these context variables. In general, the APIs for inference context setup are typically deterministic and accept identical parameters as input for each evaluation. Hence, the inference context setup codes will perform the same operation and return the same value per iteration. As a consequence, this holistic UDF evaluation paradigm leads to the significant overhead of repetitively setting up the identical inference context.

Since the inference context setup process is invariant during the iterative execution process, hoisting them outside the entire query loop program and executing them only once to avoid repetitive operations is beneficial. It is a straightforward idea to let users identify the inference context setup codes and move them to the start of the iterative query execution. Accordingly, recent works, e.g., SQL Server [67], SparkSQL [5, 74] and EvaDB [32, 79], introduce *staged prediction function evaluation*. Instead of evaluating the prediction function holistically, this approach divides the prediction function (PF) evaluation process into two stages, i.e., the *PF initialization stage* and the *PF computation stage*, with a specialized *stage-aware user interface*. The PF initialization stage is executed only once before query execution, whereas the PF computation stage is executed during the iterative execution process.

As shown in Figure 5, the stage-aware function interface divides the prediction function into two methods: a context initialization method `__init__`, and an evaluation method `__call__`. This interface enables users to put the identified inference context setup codes into the `__init__` method, which is executed at the PF initialization stage. The operator will also pass the `context` instance as function parameters to the `__call__` method so that users are able to obtain the variables defined in

the `__init__` method at the PF computation stage. The operator will evaluate the `__call__` method at the PF computation stage, which contains the codes that perform the actual prediction process such as preprocessing and ML inference. With this stage-aware interface, users can manually place inference context setup codes at the PF initialization stage to ensure one-off inference context setup. At the the PF computation stage, the rest of the UDF codes are executed iteratively reusing the initialized context. Clearly, this manual staged evaluation approach avoids the repetitive inference context setup.

3.2 Automatic Inference Context Reuse

Although the staged prediction function evaluation achieves the one-off inference context setup, it is clear that this approach requires users to rewrite the original function codes to the above stage-aware interface form manually. For example, to rewrite the codes in Figure 5, users have to add two lines of code (LOCs) for implementing two functions and modify six LOCs. In particular, a new variable of *context* is employed for context setup and prediction. This rewriting process imposes an extra effort on the users and is also error-prone. In addition, as the number of prediction models used in the UDF increases and the control logic of user-defined codes becomes complex, this process will become more intricate. Especially for the deployed functions, users have to manually modify every function with sophisticated programming skills and ensure that the rewriting process achieves the same results as the original query. This motivates us to seek an approach that automatically reuses the initialized inference context in the prediction function.

3.2.1 Challenges of Automatic Inference Context Reuse. Intuitively, a natural extension to achieve automatic inference context reuse is to have a prediction function rewriter. It transforms user codes into the stage-aware interface form and exploits the existing staged evaluation mechanism. According to the semantics of staged prediction function evaluation, the rewriting problem can be formulated as a classical *Loop Invariant Code Motion* (LICM) problem [2] in programming language (PL) optimization. The original LICM algorithm detects loop invariant codes and moves them outside the loop to avoid redundant processing through static code analysis. However, there are two challenges to resolving the automatic inference context with this static analysis approach.

Challenge 1: Diverse Prediction Function Codes. The general practice to implement the LICM algorithm is translating the whole Python UDF codes into an intermediate representation (IR) and then searching and hoisting the loop invariant codes by running the LICM algorithm on the IR. Traditional static analyses typically work on low-level IRs (e.g., LLVM IR [39, 73, 75] and MLIR [30, 40]), which requires compiling all the Python language features into the low-level IRs for each prediction function. Another idea for the LICM problem is performing static analysis directly on the abstract syntax tree (AST) or byte code of Python codes. Nevertheless, the codes in prediction functions are rather diverse with arbitrary ML framework combinations and high-level language features, which makes compiling or analyzing the whole Python UDF codes challenging [4, 75, 83]. This is consistent with the situation that current Python static analyzers [18, 52] only provide limited programming advice and do not offer LICM optimization.

Challenge 2: Ambiguous Type and Value Information. When focusing on analyzing the inference context setup codes, the complicated Python language features [7, 43, 51] also lay a challenge for obtaining accurate types and value-variation patterns required by the LICM algorithm. First, it is impractical to obtain the exact data types that a program will manipulate without runtime information as Python is a dynamically typed language. Second, value-variation patterns of variables used in the inference context setup process are difficult to determine without running the code, since all variables in Python are object references and every operation in the code may mutate the value of the variable. Some operations in Python are even non-deterministic and return

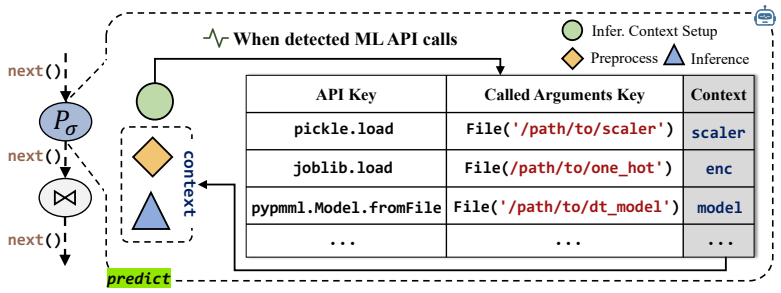


Fig. 6. Runtime Inference Context Reuse Cache

different values for each execution. Consequently, the lack of accurate types and value-variation patterns hinders the capture of semantics through static analysis.

In general, rewriting the prediction function via static analysis is infeasible. Hence, we propose a novel *runtime inference context reuse cache* in our prediction-aware operator to achieve the one-off inference context setup automatically.

3.2.2 Runtime Cache-based Inference Context Reuse. As depicted in Figure 6, our prediction-aware operator binds an inference context reuse cache for each prediction function. The inference context reuse cache consists of an ML API detector and an inference context object-store. In particular, the ML API detector ensures that the prediction-aware operator detects the target ML API call during the evaluation of the prediction function. When the inference context setup API is called, the prediction-aware operator will first check the reuse opportunity by looking up the inference context object store with the detected ML API and arguments. If the context has already been set up, our prediction-aware operator will reuse it for subsequent operations in the prediction function.

Rationale 1. *Despite the user-defined Python codes within each prediction function being diverse and complex, the ML APIs used for inference context setup are limited and deterministic.*

To address Challenge 1, it is necessary to analyze the general pattern of ML APIs. Typically, the ML APIs are provided by ML frameworks. According to the survey from [31] and [62], 80% of real-world user-defined codes adopt around ten ML frameworks. Based on our observations of API documents, each ML framework contains at most three APIs for inference context setup. Thus, we only need to mark and detect the limited APIs without compiling or analyzing the whole Python prediction function. Also, following the convention of ML framework developers, APIs used for inference context setup are deterministic and the rest of the UDF codes do not mutate the object returned from these APIs. Hence, we can safely cache the context object after detecting the first API call and reuse it for subsequent calls without changing the original semantics of the prediction function. Accordingly, we define ML APIs as detection rules (API pattern strings) to identify an API provided by the ML framework. For example, to mark the `pickle.load` as the target API for reuse, the ML API detector takes string `"pickle.load"` as input and exploits the reflection mechanism to load the `load` function from the `pickle` module as a Python object in memory at runtime. Note that this detection rule mechanism is highly extensible for different ML frameworks. We currently have supported ML APIs from various prevalent ML frameworks like scikit-learn, XGBoost, and ONNX Runtime in our ML API set.

Rationale 2. *Despite the complicated language features making the type and value information ambiguous to analyze, they are determined when passing arguments to ML APIs at runtime.*

Algorithm 1 Automatic Inference Context Reuse

Input: Python Env. *interp* and ML API detection rule set *rules*

```
1:  // Initialize the cache when interp starts up
2:  store ← an empty map[ml_api, map[args, context]]
3:  // Inject reuse_callback into interp as import hooks
4:  for rule in rules do
5:      hook ← generate_hook(rule, reuse_callback)
6:      register_import_hook(interp, hook)
7:  // Trigger the reuse_callback whenever ML API is called
8:  function reuse_callback(called_func, called_args)
9:      sub_store ← find_by_key(store, called_func)
10:     if sub_store not found then
11:         context ← called_func(called_args)
12:         store[called_func][called_args] ← context
13:     else
14:         context ← find_by_key(sub_store, called_args)
15:         if context not found then
16:             context ← called_func(called_args)
17:             store[called_func][called_args] ← context
18:     return context
```

To address Challenge 2, we can leverage this characteristic to obtain the accurate data type and value information at runtime for reuse checking. In particular, reuse checking is performed right when the ML API is called, where the exact types and values of the API and called arguments are determined at this point. For instance, the type and value of `f` for loading the `scaler` appeared in Figure 1(a) is determined to be a file handler object when passing to the `pickle.load` ML API. Hence, it is desirable to cache the `scaler` object with the determined arguments and check for reuse right next time `pickle.load` is called. Since the `pickle.load` API is deterministic, and the rest of the codes do not change the `scaler` object, the result of the prediction function is the same as the original codes after reusing the `scaler` object.

To automate the reuse process without user perception, we need to obtain the target ML API, inject the reuse checking logic into the ML API, and substitute the original ML API with the injected API dynamically at runtime. The detailed process of our automatic inference context reuse is shown in Algorithm 1. First, the inference context object store is initialized as a key-value map when the Python language environment starts up (Line 2). The key-value map will store ML API, called arguments as keys, and returned context objects as values. To support fast lookup, we implement it as a hash table. After that, the ML API detector accepts the detection rules, injects the reuse-checking logic into the target ML APIs, and leverages "monkey patching" [22] to achieve the injected API substitution.

Whenever the target ML API is called during query execution, the *reuse_callback* is triggered to perform the reuse check. We look up the inference context object store with the called API *called_func* (Line 9) and called arguments *called_args* (Line 14). The *called_func* is a function reference and the *called_args* is all the types of objects accepted by the ML API. Note that we use different key-equality determination methods for *called_func* and *called_args*. Specifically, two *called_func* are considered equal (is) if they are the same reference since ML APIs are typically object instances with a unique reference ID. Two *called_args* are deemed as equal (==) if they have

the same value since *called_args* are typically different references for each iteration with the same value. For example, `pickle.load` is a function with a unique reference, and two file objects are considered equal (==) if their `path` attributes are equal (==). Besides, the ML APIs from ONNX Runtime, e.g., `InferenceSession`, allows users to pass an `option` object to tune the inference performance [56]. In this case, our reuse cache will consider that two `option` objects are equivalent if all their attributes are equal. Furthermore, if the *sub_store* or *context* object is not found in the store, we will execute the *called_func* with *called_args* to obtain the *context* and update the inference context store (Line 12 and 17). Otherwise, we will return it straightly.

Since the inference context setup is only executed once according to the above reuse process, our inference context reuse cache automatically achieves the one-off inference context setup without modifying user codes. It is also worth noting that our method can be implemented as a standalone module agnostic to specific database engines. Integrating this method into database engines only requires importing the reuse module with only a few lines of code. Thus, database developers do not have to re-implement the execution process of relational operators in the execution engine to support the staged prediction function evaluation.

3.2.3 Discussion. Rationale 1 sets a boundary of optimization effectiveness for our automatic reuse cache. Specifically, we have an ML API detection rule set for APIs satisfying this claim as a precondition check. Our approach has no impact on the called ML API which is not in this set. Clearly, this approach has the drawback that the optimizations are determined by the completeness of this API set. However, extending the rule set is an internal implementation detail for IMBridge without user perception.

4 Desirable Batching Inference

In this section, we analyze the disadvantage of batch-oblivious function invocation in database engines in Section 4.1 and depict batch-aware function invocation by which IMBridge achieves the desirable batching inference in Section 4.2.

4.1 Batch-oblivious Function Invocation

Modern database engines commonly utilize the pipeline mechanism to accelerate query execution [41, 50]. In this mechanism, the query plan is partitioned into a set of query pipelines and each query pipeline comprises a sequential chain of operators. On the pipeline, each upstream operator can launch the computation without the requirement for the downstream operator to finish computing the full amount of data. The query engine avoids materializing the intermediate data by transferring data in a sequence of batches between operators iteratively within a pipeline. Correspondingly, operations that require the materialization of intermediate data, e.g., hash table building in `JOIN`, are defined as pipeline breakers.

4.1.1 Drawback of Batch-oblivious Invocation. Current database engines invoke the prediction function according to the FIBS defined in Section 2.2, which is determined by the batch size variation pattern on a query pipeline. Similar to the Unit of Transfer (UoT) concept from [14], we define the inter-operator transfer batch size (TBS) as the batch size of tuples transferred between adjacent operators. Here, TBS describes the data-flow behavior across operators. The database engine has a design space based on how to define the TBS. For example, the TBS of classical tuple-at-a-time engines like PostgreSQL and SQLite [21] is fixed to 1. The TBS of vectorized engines, such as DuckDB [15], SparkSQL [74], ClickHouse [10, 69], and OceanBase, are typically empirical values set by engineers. As the initial batch size is independent across pipelines, we only need to focus on the batch size variation pattern within a single pipeline to analyze the FIBS of a prediction function.

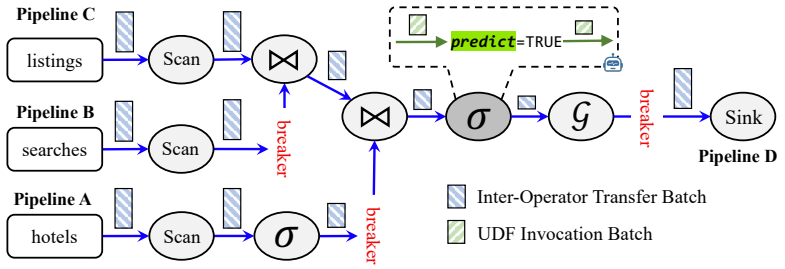


Fig. 7. Batch-oblivious Function Invocation

Given a query pipeline, the FIBS is constrained by the initial batch size, which is decided by a system-level parameter of database engines. In general, this parameter is defined diversely but behaves similarly in different database engines, thereby we denote it as θ, where $TBS \leq \theta$. For example, OceanBase defines the parameter θ called `rowsets_max_rows` considering the block size of the storage layer. In particular, θ sets an upper bound of TBS, where all the data batches transferred between operators are restricted to not exceed this parameter on the query pipeline. In addition to the configuration of the initial batch size, the FIBS is also affected cumulatively by the change in batch size from operations, e.g., the selection predicates and join conditions [54], before that UDF. In Figure 7, we take the execution plan of Figure 1(b) in OceanBase as an example. We focus on the query Pipeline C to analyze the variation pattern of the FIBS since the `predict` prediction function is embedded in the selection operator of this query pipeline. For each iteration, the initial batch size of this pipeline is 256 following the default θ in OceanBase. Then, the subsequent two join conditions will decrease the TBS. Since `predict=TRUE` is the only predicate of this selection operator, the FIBS of the `predict` function is equal to the TBS transferred to the selection operator. In conclusion, the variation of batch size imposed by the operations previously to the `predict` will make its FIBS less than 256 frequently per iteration.

As mentioned in Section 2.2, the input batch size affects the *PPS* of the prediction function *PF* substantially. We measure the total elapsed time spent on executing the *PF* as $\sum_{i=1}^{n} \frac{bs_i}{PPS(PF,bs_i)}$, where bs_i is the FIBS of i-th *PF* invocation, and n is the number of invocations for *PF*. Typically, there is a *Desirable Inference Batch Size* (DIBS), denoted as bs^*, that maximizes the $PPS(PF)$ for a specific prediction function, i.e., $PPS(PF, bs_i) \leq PPS(PF, bs^*)$. In particular, the DIBS varies widely for various prediction functions with different ML frameworks and models. Here, the total elapsed time spent on executing the *PF* is $\sum_{i=1}^{m} \frac{bs^*}{PPS(PF,bs^*)}$, where m is the number of invocations for *PF* with desirable batch size.

Considering that $\sum_{i=1}^{n} bs_i = \sum_{i=1}^{m} bs^*$, which means the same total number of processed tuples, we have

$$\sum_{i=1}^{n} \frac{bs_i}{PPS(PF, bs_i)} \geq \sum_{i=1}^{n} \frac{bs_i}{PPS(PF, bs^*)} = \sum_{i=1}^{m} \frac{bs^*}{PPS(PF, bs^*)}.$$

Clearly, the total elapsed time of *PF* execution with desirable inference batch size is lower than that with undesirable inference batch size. Hence, it is necessary to ensure that the *PF* is invoked only if the input size meets desirable batch size to achieve an efficient batching inference.

4.1.2 Mitigating the Undesirable Batching Inference. An intuitive method for mitigating the undesirable batching inference in database engines is to acquire a DIBS outside the database and tune the θ to make the FIBS close to the DIBS. However, setting the θ parameter according to the DIBS

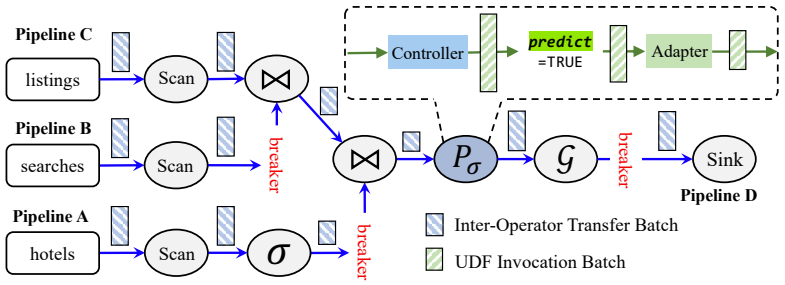


Fig. 8. Batch-aware Function Invocation

is inflexible in current database engines. For engines that apply the tuple-at-a-time execution, the θ is fixed to 1 by the system design, thereby tuning this parameter is infeasible. For engines that leverage the vectorized execution, an identical θ is adopted for each query in the system. Adjusting the θ will affect all queries in the system. Moreover, some database engines, like DuckDB [15], even set the θ in kernel codes, which requires recompiling the kernel for parameter change every time. Besides, the cumulative effect on the query plan will make the final FIBS not match the DIBS, which we will showcase in Section 6.3. Thus, it is necessary to introduce a DIBS-aware batch control mechanism to achieve a desirable batching inference.

4.2 Batch-aware Function Invocation

According to the above analysis, the main reason for the mismatch between FIBS and the DIBS of prediction functions is the lack of a control logic specifically for batch-size manipulation among existing operators in database engines. To address this issue, we introduce the batch-aware function invocation as another core design of the prediction-aware operator in IMBridge. As shown in Figure 8, instead of attaching the `predict` function to the original selection operator, we explore a prediction-aware selection operator on the query plan holding the original semantics. The operator maintains an independent DIBS parameter to meet the performance requirements, e.g., 2048, of the `predict` function. Also, we propose a specific batch controller to decide when to invoke the prediction function considering the DIBS. The batch controller ensures that the FIBS satisfies the DIBS for each prediction function invocation to keep a high *PPS*. Moreover, we add a batch adapter to interact with subsequent operators in the query execution plan.

4.2.1 Batch Controller for Prediction Functions. Algorithm 2 depicts how IMBridge controls the batch size according to the TBS transferred from the previous operator. Given a prediction function PF and its DIBS, let $batch_i$ be the data batch transferred from the adjacent operator for the i-th iteration of execution of the query pipeline. The batch controller consists of a buffer to hold the intermediate data in memory temporarily. Typically, this buffer is a ring-queue-based buffer that manages the data in a FIFO manner to ensure the ordering semantics when order operations such as `ORDER BY` appear in a query. Also, the batch controller maintains two states, `BUFFERING` and `SLICING` to trigger different operations for the prediction-aware operator. Initially, the buffer is empty and the controller is at the `BUFFERING` state (Line 2-3). Based on the current state of the batch controller during the iterative execution process, we present the control logic in the following two cases.

Buffering State. When the batch controller is at the `BUFFERING` state, the intermediate data stored in the buffer is less than the desirable inference batch size, i.e., $len(ctrl.buffer) < bs^*$. This indicates that the batch size is insufficient to invoke the prediction function. Thus, the batch controller has

Algorithm 2 The Process of Batch Control

Input: Prediction Function PF and its DIBS bs^*

```
1:  // Initialize the batch controller ctrl before query execution
2:  ctrl.buffer ← an empty queue
3:  ctrl.state ← BUFFERING
4:  // Perform control on batch_i according to the state of ctrl
5:  switch ctrl.state do
6:      case BUFFERING:
7:          ctrl.buffer.push(batch_i)
8:          if len(ctrl.buffer) >= bs* then ctrl.state ← SLICING
9:          else   ctrl.state ← BUFFERING
10:             Request for the next batch batch_{i+1}
11:     case SLICING:
12:         for i = 0 to len(ctrl.buffer) | bs* do
13:             slice ← ctrl.buffer.pop_front(bs*)
14:             Invoke PF with slice and do batch adapt
15:         ctrl.state ← BUFFERING
16:         Request for the next batch batch_{i+1}
17: // Drain the rest of batch in the buffer at last
18: procedure final_execute
19:     for i = 0 to ⌈len(buffer)/bs*⌉ do
20:         slice ← ctrl.buffer.pop_front(bs*)
21:         Invoke PF with slice and do batch adapt
```

to continue storing the $batch_i$ transferred from downstream in the back of the buffer (Line 7). Note that we design the FIBS of the prediction function to conform to bs^* strictly. Thus, the controller state will switch to BUFFERING or SLICING state depending on the buffer size after en-queuing $batch_i$. Whenever the buffer size is no less than the bs^*, the controller state will immediately switch to SLICING (Line 8), indicating that the prediction-aware operator is able to invoke the prediction function for further computation. Otherwise, the controller state will switch to BUFFERING (Line 9) and the prediction-aware operator will request its child operator for more data batches (Line 10). Typically, query operators in database engines all implement the same interfaces. Hence, we are able to leverage this characteristic to fetch the next data batch, i.e., $batch_{i+1}$, from downstream.

Slicing State. When the batch controller is at the SLICING state, the intermediate data stored in the buffer is more than the desirable inference batch size, i.e., $len(ctrl.buffer) \geq bs^*$. This indicates that the batch size is compliant for invoking the prediction function. Accordingly, the prediction-aware operator will slice the data for function invocation. In particular, the operator will pop the data batch with suitable segments from the front of the buffer and feed them to the prediction function (Line 12) sequentially. Note that our prediction-aware operator will compact each slice into a vector with continuous memory layout before prediction function invocation to support vectorized UDF execution, since the data in the buffer may come from different iterations. This compaction process is the main overhead for introducing the batch control. The slicing process will continue until the data remaining in the buffer does not satisfy the bs^*. Finally, the controller state will switch to BUFFERING and the prediction-aware operator will request for the next batch as the buffer size does not meet the bs^* requirement at this moment.

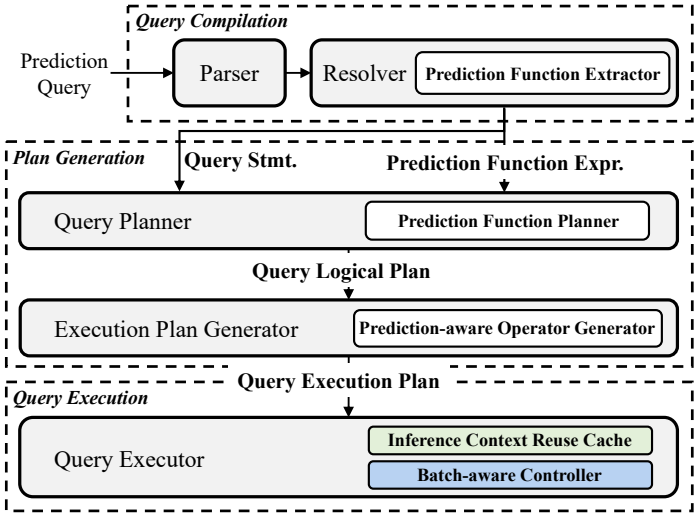


Fig. 9. System Architecture of IMBridge

To ensure that all the data is processed by the operator, the batch control mechanism also contains a draining process to flush the data remaining in the buffer after the downstream operator is exhausted (Line 18). In particular, the operator will also pop and process the data slice by slice until the buffer becomes empty.

Instead of an unpredictable FIBS constrained by the batch size variation pattern of database engines, the FIBS in IMBridge is guaranteed to be bs^*, suggesting $PPS(PF, bs_i) = PPS(PF, bs^*)$ and a much lower total time for prediction function execution. Hence, IMBridge is able to achieve a desirable batching inference through the batch-aware function invocation mechanism.

4.2.2 Batch Adapter for Subsequent Operators. Note that our operator is designed as a regular operator compatible with other database engine operators. The operator can be used in arbitrary queries also further processing the result, e.g. the `AGGREGATE` operation in Figure 1(b). Since the TBS of other operators is designed based on the constraint of θ in database engines, we add a batch adapter into the prediction-aware operator in IMBridge to transfer the amount of data required by its adjacent operator for the compatibility consideration (Line 14 and 21). In particular, the batch adapter will store the results temporarily and transfer the results on demand to ensure that the transferring results do not violate the θ setting.

5 System Implementation

We have implemented IMBridge in OceanBase [55, 80, 81]. As illustrated in Figure 9, the query compiler of IMBridge first parses the prediction query to an AST and resolves it with the schema. It also extracts the prediction function expression using the prediction function extractor. The prediction function planner identifies the prediction function expression and binds the Python UDF metadata to the logical query plan. Then, the execution plan generator takes the query logical plan and generates the query execution plan with the prediction-aware operator. During execution, IMBridge automatically reuses the inference context from the cache and evaluates the prediction function according to the desirable inference batch size with the batch-aware controller.

We inherit the base class of physical operators in OceanBase as a startup and register the prediction-aware operator to the query planner and executor. Based on this approach, we can

Table 1. Experimental Prediction Queries

ID	Use Case	Dataset	Model	Frameworks
Q1	Rank Prediction	Expedia	DT	ORT
Q2	Codeshare Prediction	Flights	RF	Sklearn
Q3	Fraud Detection	Credit Card	GBDT	XGBoost
Q4	Local Supplier Volume	TPC-H	MLP	PyTorch
Q5	Returned Item Reporting	TPC-H	GBDT	LightGBM
Q6	Sales Forecasting	TPCx-AI	HW	Statsmodels
Q7	Spam Detection	TPCx-AI	NB	Sklearn
Q8	Product Rating	TPCx-AI	CF	Surprise

```
SELECT predict(data), <Projection1>, <Projection2>, ...
FROM (listings l JOIN hotels h ON l.prop_id=h.prop_id)
JOIN searches s ON l.srch_id=s.srch_id
WHERE <Predicate1> AND <Predicate2> AND <Predicate3> AND ...;
```

Fig. 10. SPJ-based Query Template on Expedia Dataset

reuse common data processing methods like data fetching and transferring. We also implement parallel prediction query execution with the volcano exchange operator [6, 60]. Specifically, the inference context reuse cache is integrated as a standalone Python module and pre-imported when the Python environment is initialized at the launch time of OceanBase. In addition, we implement a heuristic-based desirable inference batch size estimator as an independent plugin for IMBridge. The estimator collects efficiency statistics for each prediction function invocation and searches for the DIBS employing an additive-increase scheme inspired by the batch-size tuning scheme adopted in inference servers [3, 11, 12]. With the estimator, IMBridge will automatically obtain a DIBS at runtime and cache it for future reuse. Also, users can disable the estimator and manually specify a DIBS when creating the prediction function.

Extension to Other Database Engines. OceanBase represents the pull-based paradigm, a.k.a, iterator model. Thus, extending IMBridge to database engines with this paradigm, e.g., PostgreSQL, SQLite [21] is similar to the above discussion. For databases like ClickHouse [10], DuckDB [65], and PolarDB [78] that exploit the push-based paradigm, operators are deemed as consumers and producers. Taking our implementation atop DuckDB as an example, the query executor iteratively drives the leaf nodes to push data to the root node by calling the `source` and `sink` interface during execution. The push-based paradigm also embeds the prediction function in a certain operator. Thus, we can port the inference context reuse cache to the push-based execution engine by importing our cache module similarly in the DuckDB Python extension. The core logic of the batch-aware controller in batch-aware function invocation is how to fetch the next batch. Different from the pull-based paradigm, the push-based operator cannot directly request the next batch from its downstream operator as the data is pushed from the bottom to up. Hence, it is indispensable to extend the push-based operator by proactively sending requests to the query executor to schedule leaf operators to push the next batch.

6 Experimental Studies

In this section, we show the experimental setting in Section 6.1. Then, we study the efficiency of IMBridge in Section 6.2 and Section 6.3. In addition, we make a further discussion in Section 6.4.

6.1 Experimental Setting

We run all experiments on a server node with Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz (48 physical cores) with 64 KB, 1024 KB, and 36608 KB L1, L2, and L3 caches, 128 GB DDR4 RAM and a 24 TB AVAGO MR9361-8i RAID Controller. The software stack comprises Ubuntu 20.04, Python 3.11, OceanBase Paetica 4.3 [80], EvaDB 0.39 [32], PostgreSQL 15.5, DuckDB 0.10.1, and MADlib 2.1.0 [25].

Datasets. We use two kinds of datasets in our experimental studies.

- *Publicly Available Datasets*: We use publicly available datasets, including Expedia[2], Flights[2], and Credit Card[3], to discuss the optimization effects. All the selected datasets are widely used in data science tasks [20, 59, 75]. Since the original size is small, we scale the sizes of these datasets to about 45M-65M rows (after joining) by replicating each dataset several folds following the experiments in Raven [59] and YeSQL [20].
- *Benchmark Datasets*: We use benchmark datasets from TPCx-AI [9] and TPC-H to evaluate the optimization effects and usability in typical industrial scenarios. We exploit the scale method in TPC toolkits and set the scale factor to 10 (SF=10) by default.

Prediction Queries. A Prediction query is composed of both SQL and prediction functions, thereby we choose workloads as follows:

- *SPJ-based Queries (Q1-Q3)*: Following Raven [59], the prediction queries we adopt on publicly available datasets are SPJ-based queries generated from basic Select-Project-Join SQL templates. As depicted in Figure 10, Q1-Q3 consist of the prediction function and one or two projections at the `SELECT` clause. The predicates in the `WHERE` clause range from one to six. As summarized in Table 1, we use various models and ML frameworks for prediction functions to showcase the variety of workloads that IMBridge boosts. We pick typical ML models widely used in relational data [31], namely Decision Tree (DT), Random Forest (RF), and Gradient Boosting Decision Tree (GBDT) with prevalent ML frameworks to perform prediction tasks on publicly available datasets. For the prepossessing step, we use a standard scaler to normalize the numerical columns and a one-hot encoder to encode the categorical columns.
- *TPC-H-based Queries (Q4-Q5)*: We further conduct experiments for decision support scenarios using TPC-H-based queries. To construct Q4, we select the query of local supplier volume in TPC-H and add prediction function into the `WHERE` clause to predict the order priority. To construct Q5, we pick up the query of return item reporting in TPC-H and add prediction function into the `WHERE` clause to predict whether an user will return an item. Besides, we employ Multi-Layer Perceptron (MLP) and GBDT as the models used in the Python UDFs with the PyTorch and LightGBM frameworks in Q4 and Q5 respectively.
- *TPCx-AI-based Queries (Q6-Q8)*: We also conduct experiments using TPCx-AI-based queries with more complex queries representing ML scenarios. For example, Q8 uses the score prediction function and `rank()` window function to recommend top-10 items for each user. In particular, we rewrite the original pandas-based Python codes to equivalent SQL-based queries. We extract prediction functions from the use cases in TPCx-AI, namely Holt-Winters (HW), Naive Bayes

[2]https://adalabucsd.github.io/hamlet.html
[3]https://www.kaggle.com/datasets/mlg-ulb/creditcardfraud

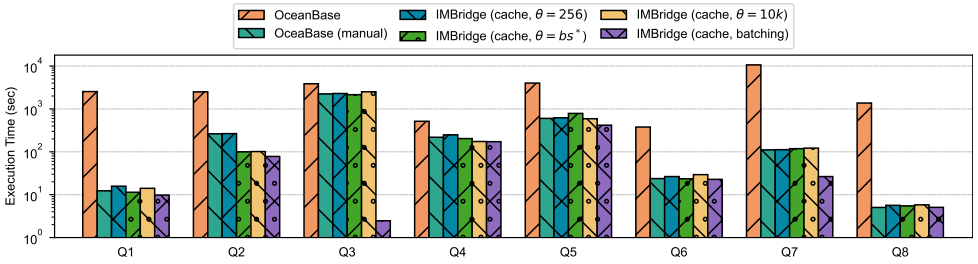


Fig. 11. End-to-End Execution Time

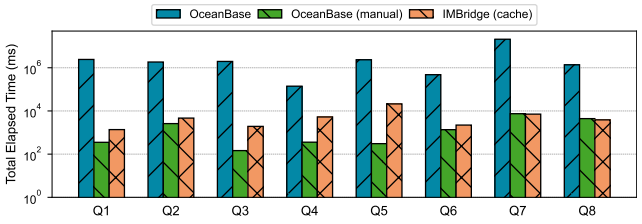


Fig. 12. Overhead of Context Setup

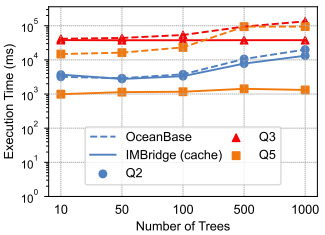


Fig. 13. Impact of Model Sizes

(NB), Collaborative Filtering (CF) with Sklearn, Statsmodels [76], and Surprise [27], and rewrite the serving process to equivalent Python UDFs.

In the following, we study the efficiency of IMBridge primarily based on our implementation in OceanBase. We compare it with existing Python UDF solutions and also provide a study on extending the implementation of IMBridge on DuckDB. It is worth noting that compile-time UDF optimization techniques using IR, like UDO [73] and Tuplex [75], are not included, since they are orthogonal to our runtime optimization of IMBridge.

6.2 Efficiency of One-off Context Setup

In this section, we study the performance of the one-off inference context setup. Also, we demonstrate the overhead reduction and impact of model sizes for setting up the inference context.

6.2.1 End-to-End Execution Time. We enable the automatic one-off inference context setup in IMBridge, denoted as IMBridge (cache, $\theta = 256$), which employs inference context reuse cache. Differently, Oceanbase explores holistic prediction function evaluation and provides an interface for manual inference context reuse. We also set $\theta = 256$ in OceanBase and denote them as OceanBase and OceanBase (manual) respectively.

As depicted in Figure 11, IMBridge (cache, $\theta = 256$) achieves 18.04x improvements on average against OceanBase and only has a 3.4% performance gap on average against the manual reuse approach, i.e., OceanBase (manual). The performance improvements of the one-off inference context setup vary widely across various model types and ML frameworks. Simple model like MLP takes only a little time to perform the inference context setup. For Q4, IMBridge (cache, $\theta = 256$) is only 2.1x faster than OceanBase. For ensemble forest models like RF and GBDT used in Q2, Q3, and Q5, IMBridge (cache, $\theta = 256$) achieves a 1.7x-9.4x speedup over OceanBase. For Q6-Q8, IMBridge (cache, $\theta = 256$) even achieves 6.4x, 94.8x, and 243.3x speedup over OceanBase, respectively. This is because the size of the models in TPCx-AI-based queries is larger than those of other queries.

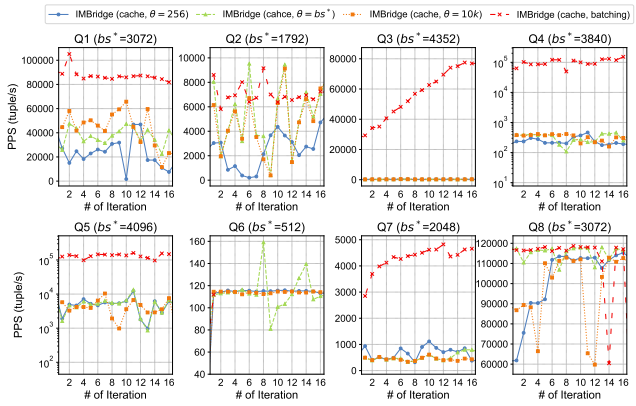


Fig. 14. The *PPS* of Prediction Function Execution

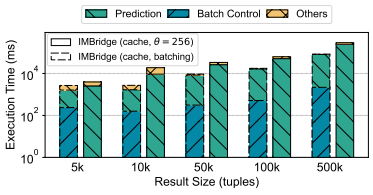


Fig. 15. Overhead of Batch Control

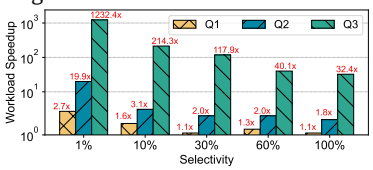


Fig. 16. Impact of Selectivity

In particular, Q6 contains multiple time series models, Q7 contains a TF-IDF token table, and Q8 contains a huge collaborative matrix. Thus, IMBridge (cache, $\theta = 256$) achieves a significant speedup over OceanBase for TPCx-AI-based queries.

In general, IMBridge is able to speed up query execution significantly across various workloads since it reuses the initialized inference context automatically.

6.2.2 Overhead of Inference Context Setup. To demonstrate the effect of inference context reuse cache, we measure the total elapsed time of inference context setup in OceanBase, OceanBase (manual), and IMBridge (cache). Figure 12 shows the overhead of inference context setup on four representative prediction queries. In general, IMBridge (cache) has a 96.2%-99.9% reduction of the elapsed time of inference context setup over OceanBase, and OceanBase (manual) achieves a 99.6%-99.9% reduction over the original OceanBase for all four queries. The performance gap between IMBridge (cache) and OceanBase (manual) is around 64.3% on average, since our reuse cache method introduces a minor extra cache lookup overhead for each iteration. Also, the inference context setup time of both IMBridge (cache) and OceanBase (manual) only accounts for a small portion of the whole query execution time. In summary, the above results indicate that IMBridge achieves comparable performance to the manually rewritten codes for one-off inference context setup with the inference context reuse cache.

6.2.3 Impact of Model Size. Since the model loading process affects the inference context setup time greatly, we study the impact of model size on the inference context reuse cache by evaluating the speedup of IMBridge (cache) over OceanBase. In particular, we tune the number of trees from 10 to 1000 for the ensemble forest used in Q2, Q3, and Q5 to train models with different sizes, and then execute these queries by varying the model size. Figure 13 shows that along with the growth of model sizes, the optimization effect of inference context reuse cache increases. This is because the overhead of model loading during inference context setup increases when the model used in the prediction function is more complex.

6.3 Efficiency of Desirable Batching Inference

In this section, we study the efficiency of desirable batching inference. Also, we show the improvement of prediction processing speed, the batch control overhead, the impact of selectivity, and the impact on the response time.

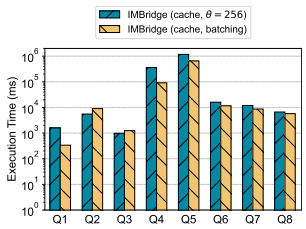


Fig. 17. Response Time

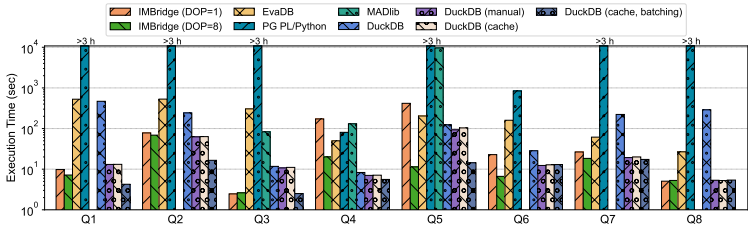


Fig. 18. System Comparison

6.3.1 End-to-End Execution Time. We enable desirable batching inference on the basis of inference context reuse cache, denoted as IMBridge (cache, batching), which applies batch-aware function invocation. Differently, IMBridge (one-off, $\theta = 256$) remains to adopt the batch-oblivious function invocation with the θ recommended in OceanBase. As mentioned in Section 4.1, the FIBS of a prediction function is affected by θ. Hence, we show the results by only tuning the θ parameter according to bs^* acquired from our estimator, i.e., IMBridge (cache, $\theta = bs^*$), and an empirical upper bound, i.e., 10k, utilized in SparkSQL [74], denoted as IMBridge (cache, $\theta = 10k$).

As illustrated in Figure 11, IMBridge (cache, batching) boosts query execution by 3.96x on average over IMBridge (cache, $\theta = 256$) through the batch-aware function invocation. Obtaining a bs^* previously and setting the θ manually does not exhibit a satisfactory performance improvement for most cases. Only Q1, Q2, and Q4 with IMBridge (cache, $\theta = bs^*$) and IMBridge (cache, $\theta = 10k$) have a 1.22x-2.66x speedup over IMBridge (cache, $\theta = 256$). This is because the FIBS of a prediction function is also influenced by the batch size variation imposed by operators on the query plan (see Section 4.1). Surprisingly, for Q3, Q5, Q6, Q7, and Q8, there are even 5.5%-26.5% and 2.5%-10.8% performance degradation in IMBridge (cache, $\theta = bs^*$) and IMBridge (cache, $\theta = 10k$) against IMBridge (cache, $\theta = 256$). This is due to the fact that changing the θ parameter of the database affects both the prediction function execution part and the relational operation execution part. For instance, a higher θ may incur a higher I/O latency when scanning tables asynchronously in OceanBase. In addition, the efficiency of the IMBridge (cache, batching) over IMBridge (cache, $\theta = 256$) is different across model types and ML frameworks. For Q3 which uses the XGBoost, the performance acceleration is significant, since they require a large inference batch size for parallel processing with OpenMP. For Q6 and Q8 using the HW and CF models, the performance improvement is minor since these models are not sensitive for batching inference.

To sum up, tuning the θ does not mitigate the undesirable batching inference and even worsens the end-to-end query performance. In contrast, IMBridge is able to achieve a desirable batching inference regardless of the θ settings as it leverages the batch-aware function invocation to control and adapt the batch size specifically.

6.3.2 Prediction Processing Speed. To demonstrate the performance of desirable batching inference, we collect the result of *PPS* in the first 16 iterations during query executions in IMBridge. In particular, to calculate PPS, we divide the batch size by the execution time of prediction function for each iteration.

As depicted in Figure 14, with the guarantee of desirable inference batch size, the results of *PPS* per iteration in IMBridge (cache, batching) are generally higher than the ones in other settings. For all the queries except for Q6, tuning θ improves the processing speed of prediction functions to a certain extent. Since the database engine is still unaware of the prediction semantics, the cumulative variation effect imposed by operators on the query pipeline will inevitably make the FIBS far from the bs^*. Clearly, the *PPS* of IMBridge (cache, $\theta = bs^*$) and IMBridge (cache, $\theta = 10k$) is unstable,

which cannot ensure a high execution efficiency for each prediction function invocation. On the contrary, IMBridge is able to bridge this efficiency gap by exploring our batch control mechanism to ensure a bs^* with high *PPS* for each prediction function invocation. Note that there are only two iterations for IMBridge (cache, batching) in Q6. This is because the data selected for inference in Q6 only have hundreds of rows. As a result, the number of prediction function invocations of IMBridge (cache, batching) in Q6 is fewer than that of IMBridge (cache $\theta = 256$), IMBridge (cache $\theta = 10k$), and IMBridge (cache $\theta = bs^*$). This indicates that the performance improvement of Q6 also comes from reducing the overhead of the language runtime switch between Python and the database engine.

To sum up, the desirable batching inference in IMBridge accelerates the query execution by improving the *PPS* metric with our batch-aware control for prediction function invocation.

6.3.3 Overhead of Batch Control. To demonstrate the extra overhead of introducing batch control, we measure the total execution time of the prediction function and batch control in Q2 between IMBridge (cache, batching) and IMBridge (cache, $\theta = 256$). In particular, we use the `LIMIT` clause to vary the query result size from 5k to 500k, where the number of iterations increases. Since the batch control happens for each iteration, the number of batch controls also increases with the query result size. As depicted in Figure 15, the overhead of batch control increases as the query result size increases. The batch control execution time accounts for 8.8% of the whole query execution time when the query result size is 5k, whereas this percentage is 2.6% when the query result size increases to 500k. However, introducing the batch control reduces the prediction function execution time by a range from 46.1% to 65.9%. Hence, it is worthwhile to introduce batch control for desirable batching inference with a minor overhead.

6.3.4 Impact of Query Selectivity. Since the selection operation is a common factor in affecting the batch size on a certain query pipeline, we next study the selectivity impact on the desirable batching inference by comparing the speedup of IMBridge (cache, batching) over IMBridge (cache, $\theta = 256$). We adopt all the SPJ-based queries in this study, since we can control the selectivity of these queries by adding extra predicates at the `WHERE` clause as shown in Figure 10. In particular, we control the selectivity to around 1%, 10%, 30%, 60%, and 100%. As depicted in Figure 16, along with the increase in selectivity, the acceleration effect of the desirable batching gradually decreases. This is due to the reason that a larger selectivity will impose less effect on the FIBS, and the FIBS is more likely to be close to the bs^*. Besides, IMBridge (cache, batching) still outperforms IMBridge (cache, $\theta = 256$) when the selectivity is 100%, especially for Q3. The reason is the default setting (i.e., $\theta = 256$) in OceanBase does not meet the bs^* requirements for different prediction query workloads.

6.3.5 Impact on Response Time. To study the impact on the query response time, i.e., the time to produce the first result, imposed by the desirable batching, we add a `LIMIT 1` clause into each query. As depicted in Figure 17, in comparison to IMBridge (cache, $\theta = 256$), IMBridge (cache, batching) has a 62.9% and 26.6% response time increase for Q2 and Q3. This is consistent with the mechanism of desirable batching that the buffering state requires the execution engine to accumulate several batches, which slows the time needed to produce the first answer. However, for other queries, the query response time of IMBridge (cache, batching) is even lower than the one of IMBridge (cache, $\theta = 256$). This is because the prediction function execution time of a large batch size is comparable to a small one since ML frameworks are typically optimized for batching inference. Moreover, producing the first answer needs performing inference on all the data for Q4 and Q5, since they are aggregation queries. Hence, the query response time of IMBridge (cache, batching) is significantly lower than the one of IMBridge (cache, $\theta = 256$) for Q4 and Q5. In general, the response time is still affected greatly by the desirable batching inference.

6.4 Discussion

We make further discussion on the comparison with other systems, performance for implementing IMBridge on DuckDB, as well as query performance on large-scale data for IMBridge.

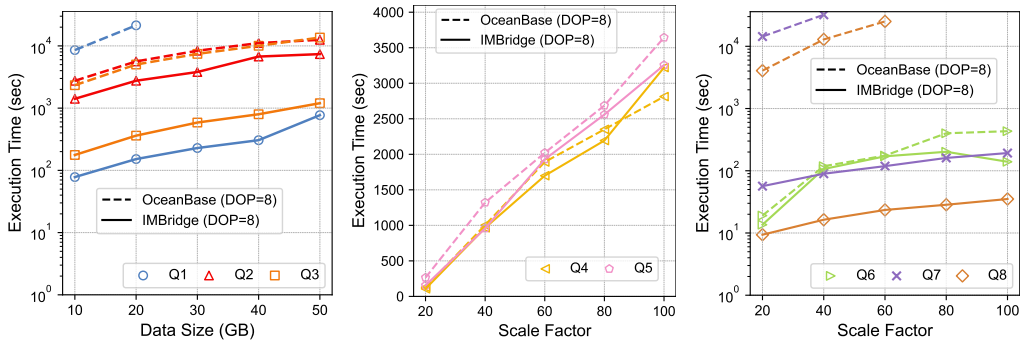
6.4.1 Comparison with Other Systems. We compare IMBridge with other solutions for executing prediction queries over data residing in databases. These systems include EvaDB [32, 79], PG MADlib [25], DuckDB [16], and PG PL/Python [61], corresponding to the extension of prediction query outside/inside the database, and Python UDF. For EvaDB, we use OceanBase as the back-end and set up the inference context before pulling data from the database. MADlib and PG PL/Python use PostgreSQL as the back-end database, and PostgreSQL has a database engine similar to OceanBase. Note that we train models individually for MADlib, since it only supports prediction with its own trained model formats. Besides, we showcase the result of IMBridge with both single thread (DOP=1) and eight threads (DOP=8) since EvaDB and MADlib only support executing prediction queries with one thread, while PG PL/Python and DuckDB provide parallel prediction query execution.

In Figure 18, we showcase the overall performance on all the prediction queries. Compared with EvaDB, IMBridge (DOP=1) has a 5.22x speedup on average, since EvaDB has to load large amounts of data from the database without performing desirable batching inference. We use MADlib to train models and generate corresponding prediction functions. For Q1 and Q2, the one-hot encoder used in the prediction functions leads to surpassing the maximum number of columns allowed in a table in PostgreSQL. Therefore, we skip the experiments for Q1 and Q2 on MADlib. We do not run TPCx-AI-based queries on MADlib either, since it does not support the preprocessing processes in these prediction functions. For Q3-Q5, IMBridge (DOP=1) performs 8.38x faster than PG MADlib on average, as MADlib applies the tuple-at-a-time inference without batching inference. For parallel prediction query execution, IMBridge (DOP=8) outperforms PG PL/Python by two orders of magnitude. This is because PG PL/Python explores tuple-at-a-time execution which suffers from the repetitive context setup and undesirable batching inference with holistic prediction function evaluation. Besides, IMBridge (DOP=8) is 7.75x faster than DuckDB on average, since DuckDB also ignores the effect of inference context setup and performs batch-oblivious prediction function invocation.

In conclusion, IMBridge delivers both high performance and usability compared to alternative systems for running prediction queries with data residing in the database.

6.4.2 Performance for Implementation on DuckDB. To demonstrate the improvement of IMBridge on other databases, we implement IMBridge on top of a modern OLAP database engine DuckDB. As shown in Figure 18, DuckDB (cache) is 4.78x faster than DuckDB on average and comparable to DuckDB (manual), which indicates that our reuse cache ensures a one-off inference context setup. Moreover, the vectorization execution mode in DuckDB cannot ensure a desirable batching inference either. In particular, DuckDB (cache, batching) performs a 2.19x speedup over DuckDB (cache) on average, which shows the benefits of the batch-aware function invocation. In summary, our implementation for DuckDB demonstrates the benefits of extending IMBridge to other database engines for accelerating prediction query execution.

6.4.3 Query Performance on Large-scale Data. To study the performance of IMBridge on large-scale data, we compare IMBridge against the original Oceanbase by varying data size with eight threads. For publicly-available datasets, we scale the data from 10G to 50G. For benchmark datasets, we turn the scale factor from 20 to 100. Note that we set ten hours as the timeout for query execution. As depicted in Figure 19(a) and 19(c), for SPJ-based and TPCx-AI-based queries, IMBridge scales super-linearly with the data size compared to OceanBase. On these workloads, the prediction function accounts for a large proportion of execution time. Differently, Figure 19(b) shows the



(a) SPJ-based Queries (b) TPC-H-based Queries (c) TPCx-AI-based Queries

Fig. 19. Query Performance on Large-scale Data

speedup of IMBridge over OceanBase on TPC-H-based queries is lower than the other queries. In these queries, as the data size grows, `JOIN` on more than four tables rather than prediction functions dominates the execution time.

7 Related Work

Integrating ML inference in database engines, a.k.a, prediction query is a promising research field at present. In general, there are three directions for enhancing prediction query execution.

Supporting ML inference through UDFs. Some database engines leverage native UDFs like C++ UDF to support both ML model training and inference, such as MADlib [25], MLog [42], Vertica-ML [29], UDO [73], and BigQuery [28]. While they have native speed when running ML algorithms, the ML pipelines supported in these systems are limited. Extending new ML functionality requires significant effort for native UDF developers. On the contrary, Python thrives in the ML community for its ease of programming and rich ecosystem of ML frameworks. Hence, current database engines, such as PG PL/Python [61], MonetDB [63], Raven [33, 59], Tuplex [75], and DuckDB [16], tend to exploit Python UDFs to embrace the enriched ML features. Among them, Raven provides end-to-end optimizations of prediction queries with model pruning optimizations on Raven IR and physical optimizations for runtime selection. UDO and Tuplex compile DB operators and UDFs into a unified IR to conduct cross-optimizations at compile time. The execution codes generated by IR typically include the prediction function, where the issue of repetitive inference context setup and undesirable batching inference still exists. Since these optimizations are agnostic to the execution process at runtime, they are orthogonal to IMBridge.

Translating ML operations to SQLs. MASQ [57] translates traditional ML pipelines such as encoders, scalers, and tree-based models from Sklearn APIs to SQLs according to SQL features like `CASE` and `JOIN` operations. LMFAO [68] and JoinBoost [26] implement ML algorithms with aggregation operations utilizing join graphs and factorized learning. DL2SQL [44] and sql-einsum [8] translate tensor computations in deep learning (DL) models to SQL with `JOIN`, `GROUP BY` and `AGGREGATION` statements. SmartLite [45] uses model quantization to optimize DL models in computation and storage, translating model computation to SQL-based bit-wise operations. Weld [58] and Amir et al. [72] translate ML operations by designing a unified IR combining the semantics from both SQL and ML operators. PyTond [70] and Tim et al. [17] translate data science workloads into pure SQL by performing static analysis on Python codes. The SQL translation approach focuses

on generating better SQLs to fully utilize database engines for fast ML operations, whereas IMBridge aims at optimizing the cooperation between database engines and ML frameworks in Python.

Adding native ML operators into database engines. For high-performance considerations, current works propose to add native ML operations at the operator level in modern database engines. LingoDB [30] designs a novel query compilation stack to make optimizations on cross-domain (ML and relational) operators viable. Luo et al. [47] extends database engines with linear algebra operations that are essential to ML algorithms. Kläbe et al. [36] introduce a native ModelJoin operator utilizing HPC libraries. TQP [24, 37] proposes a tensor-based general data processing engine, which supports running both SQL and DL workloads on common high-performance tensor computing frameworks. Unlike the mature ML frameworks in Python, most works in this direction require developing new operators to support various ML algorithms.

IMBridge follows the Python UDF approach for its fertile ML community and user-friendliness. Differently, IMBridge focuses on addressing execution bottlenecks caused by the impedance mismatches between prediction query execution and database engine. In particular, IMBridge supports automatic one-off inference context setup for arbitrary codes with various ML frameworks in Python. Besides, current database engines typically apply undesirable batching inference in UDF invocation. In contrast, IMBridge fully exploits the desirable inference batch size to accelerate the processing speed of prediction functions.

8 Conclusion

In this work, we propose a prediction-aware operator designed for mitigating the impedance mismatch between prediction query execution and database engine. To mitigate the repetitive inference context setup, it leverages an inference context reuse cache. To bridge the performance gap incurred by undesirable batching inference, it controls the batch size before the invocation of prediction functions. Moreover, we have implemented a prototype called IMBridge. Our experimental results show that IMBridge significantly accelerates the prediction query execution over OceanBase and outperforms alternative solutions on PG PL/Python, MADlib, DuckDB, and EvaDB, by orders of magnitude. Nevertheless, it is possible to integrate IMBridge into other databases.

Acknowledgments

This work was supported by the National Natural Science Foundation of China (No. 62272168), the Natural Science Foundation of Shanghai (No. 23ZR1419900) and the ECNU-OceanBase Joint Research Lab.

References

[1] Ashvin Agrawal, Rony Chatterjee, Carlo Curino, Avrilia Floratou, Neha Godwal, Matteo Interlandi, Alekh Jindal, Konstantinos Karanasos, Subru Krishnan, Brian Kroth, Jyoti Leeka, Kwanghyun Park, Hiren Patel, Olga Poppe, Fotis Psallidas, Raghu Ramakrishnan, Abhishek Roy, Karla Saur, Rathijit Sen, Markus Weimer, Travis Wright, and Yiwen Zhu. 2020. Cloudy with high chance of DBMS: A 10-year prediction for Enterprise-Grade ML. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.

[2] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2nd Edition)*.

[3] Ahsan Ali, Riccardo Pinciroli, Feng Yan, and Evgenia Smirni. 2020. BATCH: Machine Learning Inference Serving on Serverless Platforms with Adaptive Batching. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '20)*. 1–15.

[4] Jason Ansel, Edward Yang, Horace He, Natalia Gimelshein, Animesh Jain, Michael Voznesensky, Bin Bao, Peter Bell, David Berard, Evgeni Burovski, Geeta Chauhan, Anjali Chourdia, Will Constable, Alban Desmaison, Zachary DeVito, Elias Ellison, Will Feng, Jiong Gong, Michael Gschwind, Brian Hirsh, Sherlock Huang, Kshiteej Kalambarkar, Laurent Kirsch, Michael Lazos, Mario Lezcano, Yanbo Liang, Jason Liang, Yinghai Lu, C. K. Luk, Bert Maher, Yunjie Pan, Christian Puhrsch, Matthias Reso, Mark Saroufim, Marcos Yukio Siraichi, Helen Suk, Shunting Zhang, Michael

Suo, Phil Tillet, Xu Zhao, Eikan Wang, Keren Zhou, Richard Zou, Xiaodong Wang, Ajit Mathews, William Wen, Gregory Chanan, Peng Wu, and Soumith Chintala. 2024. PyTorch 2: Faster Machine Learning Through Dynamic Python Bytecode Transformation and Graph Compilation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 929–947.

[5] Michael Armbrust, Reynold S. Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K. Bradley, Xiangrui Meng, Tomer Kaftan, Michael J. Franklin, Ali Ghodsi, and Matei Zaharia. 2015. Spark SQL: Relational Data Processing in Spark. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 1383–1394.

[6] Henri Bal, Peter Boncz, and Marcin Żukowski. 2010. *Multi-core parallelization of vectorized query execution.* Ph.D. Dissertation. University of Warsaw.

[7] Stefanos Baziotis, Daniel Kang, and Charith Mendis. 2024. Dias: Dynamic Rewriting of Pandas Code. *Proc. ACM Manag. Data (PACMMOD)*, Article 58 (2024), 27 pages.

[8] Mark Blacher, Julien Klaus, Christoph Staudt, Sören Laue, Viktor Leis, and Joachim Giesen. 2023. Efficient and Portable Einstein Summation in SQL. *Proc. ACM Manag. Data (PACMMOD)* 1, 2 (2023).

[9] Christoph Brücke, Philipp Härtling, Rodrigo D Escobar Palacios, Hamesh Patel, and Tilmann Rabl. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *Proc. VLDB Endow. (PVLDB)* 16, 12 (2023), 3649–3661.

[10] ClickHouse. 2024. ClickHouse. https://clickhouse.com/docs/en/intro.

[11] Daniel Crankshaw, Gur-Eyal Sela, Xiangxi Mo, Corey Zumar, Ion Stoica, Joseph Gonzalez, and Alexey Tumanov. 2020. InferLine: latency-aware provisioning and scaling for prediction serving pipelines. In *Proceedings of the 11th ACM Symposium on Cloud Computing (SoCC)*. 477–491.

[12] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J. Franklin, Joseph E. Gonzalez, and Ion Stoica. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *Proceedings of the 14th USENIX Symposium on Networked Systems Design and Implementation (NSDI)*. 613–627.

[13] Weihao Cui, Han Zhao, Quan Chen, Hao Wei, Zirui Li, Deze Zeng, Chao Li, and Minyi Guo. 2022. DVABatch: Diversity-aware Multi-Entry Multi-Exit Batching for Efficient Processing of DNN Services on GPUs. In *Proceedings of the 2022 USENIX Annual Technical Conference (USENIX ATC)*. 183–198.

[14] Harshad Deshmukh, Bruhathi Sundarmurthy, and Jignesh M. Patel. 2022. On inter-operator data transfers in query processing. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE)*. 820–832.

[15] DuckDB. 2024. Documentation: Execution Format. https://duckdb.org/internals/vector.html.

[16] DuckDB. 2024. From Waddle to Flying: Quickly expanding DuckDB's functionality with Scalar Python UDFs. https://duckdb.org/2023/07/07/python-udf.html.

[17] Tim Fischer, Denis Hirn, and Torsten Grust. 2022. Snakes on a Plan: Compiling Python Functions into Plain SQL Queries. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 2389–2392.

[18] Flake8. 2024. Flake8: Your Tool For Style Guide Enforcement. https://flake8.pycqa.org/en/latest/.

[19] Yannis Foufoulas and Alkis Simitsis. 2023. Efficient Execution of User-Defined Functions in SQL Queries. *Proc. VLDB Endow. (PVLDB)* 16, 12 (2023), 3874–3877.

[20] Yannis Foufoulas, Alkis Simitsis, Lefteris Stamatogiannakis, and Yannis Ioannidis. 2022. YeSQL: "You Extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *Proc. VLDB Endow. (PVLDB)* 15, 10 (2022), 2270–2283.

[21] Kevin P. Gaffney, Martin Prammer, Larry Brasfield, D. Richard Hipp, Dan Kennedy, and Jignesh M. Patel. 2022. SQLite: past, present, and future. *Proc. VLDB Endow. (PVLDB)* 15, 12 (2022), 3535–3547.

[22] Stefan Grafberger, Paul Groth, and Sebastian Schelter. 2023. Automating and Optimizing Data-Centric What-If Analyses on Native Machine Learning Pipelines. *Proc. ACM Manag. Data (PACMMOD)*, Article 128 (2023), 26 pages.

[23] Philipp Marian Grulich, Steffen Zeuch, and Volker Markl. 2021. Babelfish: Efficient Execution of Polyglot Queries. *Proc. VLDB Endow. (PVLDB)* 15, 2 (2021), 196–210.

[24] Dong He, Supun C Nakandala, Dalitso Banda, Rathijit Sen, Karla Saur, Kwanghyun Park, Carlo Curino, Jesús Camacho-Rodríguez, Konstantinos Karanasos, and Matteo Interlandi. 2022. Query processing on tensor computation runtimes. *Proc. VLDB Endow. (PVLDB)* 15, 11 (2022), 2811–2825.

[25] 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. (PVLDB)* 5, 12 (2012), 1700–1711.

[26] Zezhou Huang, Rathijit Sen, Jiaxiang Liu, and Eugene Wu. 2023. JoinBoost: Grow Trees over Normalized Data Using Only SQL. *Proc. VLDB Endow. (PVLDB)* 16, 11 (2023), 3071–3084.

[27] Nicolas Hug. 2020. Surprise: A Python library for recommender systems. *Journal of Open Source Software* 5, 52 (2020), 2174.

[28] Inc. Google. 2024. Big Query ML. https://cloud.google.com/bigquery/docs/bqml-introduction.

[29] Arash Jalal Zadeh Fard, Anh Le, George Larionov, Waqas Dhillon, and Chuck Bear. 2020. Vertica-ML: Distributed Machine Learning in Vertica Database. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 755–768.
[30] Michael Jungmair, André Kohn, and Jana Giceva. 2022. Designing an open framework for query optimization and compilation. *Proc. VLDB Endow. (PVLDB)* 15, 11 (2022), 2389–2401.
[31] Kaggle. 2022. State of Data Science and Machine Learning 2022. https://www.kaggle.com/kaggle-survey-2022.
[32] Gaurav Tarlok Kakkar, Aryan Rajoria, Myna Prasanna Kalluraya, Ashmita Raju, Jiashen Cao, Kexin Rong, and Joy Arulraj. 2023. Interactive Demonstration of EVA. *Proc. VLDB Endow. (PVLDB)* 16, 12 (2023), 4082–4085.
[33] Konstantinos Karanasos, Matteo Interlandi, Doris Xin, Fotis Psallidas, Rathijit Sen, Kwanghyun Park, Ivan Popivanov, Supun Nakandal, Subru Krishnan, Markus Weimer, Yuan Yu, Raghu Ramakrishnan, and Carlo Curino. 2020. Extending Relational Query Processing with ML Inference. In *Proceedings of the 10th Conference on Innovative Data Systems Research (CIDR)*.
[34] Timo Kersten, Viktor Leis, Alfons Kemper, Thomas Neumann, Andrew Pavlo, and Peter Boncz. 2018. Everything you always wanted to know about compiled and vectorized queries but were afraid to ask. *Proc. VLDB Endow. (PVLDB)* 11, 13 (2018), 2209–2222.
[35] Steffen Kläbe, Robert DeSantis, Stefan Hagedorn, and Kai-Uwe Sattler. 2022. Accelerating Python UDFs in Vectorized Query Execution. In *Proceedings of the 12th Conference on Innovative Data Systems Research (CIDR)*.
[36] Steffen Kläbe, Stefan Hagedorn, and Kai-Uwe Sattler. 2023. Exploration of Approaches for In-Database ML. In *Proceedings of the 26th International Conference on Extending Database Technology (EDBT)*.
[37] Dimitrios Koutsoukos, Supun Nakandala, Konstantinos Karanasos, Karla Saur, Gustavo Alonso, and Matteo Interlandi. 2021. Tensors: an abstraction for general data processing. *Proc. VLDB Endow. (PVLDB)* 14, 10 (2021), 1797–1804.
[38] Arun Kumar, Jeffrey Naughton, Jignesh M. Patel, and Xiaojin Zhu. 2016. To Join or Not to Join? Thinking Twice about Joins before Feature Selection. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD)*. 19–34.
[39] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. (CGO)*. 75–86.
[40] Chris Lattner, Mehdi Amini, Uday Bondhugula, Albert Cohen, Andy Davis, Jacques Pienaar, River Riddle, Tatiana Shpeisman, Nicolas Vasilache, and Oleksandr Zinenko. 2021. MLIR: Scaling Compiler Infrastructure for Domain Specific Computation. In *2021 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 2–14.
[41] Viktor Leis, Peter Boncz, Alfons Kemper, and Thomas Neumann. 2014. Morsel-driven parallelism: a NUMA-aware query evaluation framework for the many-core age. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 743–754.
[42] Xupeng Li, Bin Cui, Yiru Chen, Wentao Wu, and Ce Zhang. 2017. MLog: towards declarative in-database machine learning. *Proc. VLDB Endow. (PVLDB)* 10, 12 (2017), 1933–1936.
[43] Zhaoheng Li, Pranav Gor, Rahul Prabhu, Hui Yu, Yuzhou Mao, and Yongjoo Park. 2023. ElasticNotebook: Enabling Live Migration for Computational Notebooks. *Proc. VLDB Endow. (PVLDB)* 17, 2 (2023), 119–133.
[44] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Feifei Li, and Gang Chen. 2022. A Comparative Study of in-Database Inference Approaches. In *Proceedings of the 38th IEEE International Conference on Data Engineering (ICDE)*. 1794–1807.
[45] Qiuru Lin, Sai Wu, Junbo Zhao, Jian Dai, Meng Shi, Gang Chen, and Feifei Li. 2023. SmartLite: A DBMS-Based Serving System for DNN Inference in Resource-Constrained Environments. *Proc. VLDB Endow. (PVLDB)* 17, 3 (2023), 278–291.
[46] Yuhan Liu, Chengcheng Wan, Kuntai Du, Henry Hoffmann, Junchen Jiang, Shan Lu, and Michael Maire. 2024. ChameleonAPI: Automatic and Efficient Customization of Neural Networks for ML Applications. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI)*. 365–386.
[47] Shangyu Luo, Zekai J. Gao, Michael Gubanov, Luis L. Perez, Dimitrije Jankov, and Christopher Jermaine. 2020. Scalable linear algebra on a relational database system. *Commun. ACM* 63, 8 (2020), 93–101.
[48] Zhenghua Lyu, Huan Hubert Zhang, Gang Xiong, Gang Guo, Haozhou Wang, Jinbao Chen, Asim Praveen, Yu Yang, Xiaoming Gao, Alexandra Wang, Wen Lin, Ashwin Agrawal, Junfeng Yang, Hao Wu, Xiaoliang Li, Feng Guo, Jiang Wu, Jesse Zhang, and Venkatesh Raghavan. 2021. Greenplum: A Hybrid Database for Transactional and Analytical Workloads. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 2530–2542.
[49] Mark Raasveldt. 2024. Memory Management in DuckDB. https://duckdb.org/2024/07/09/memory-management.html.
[50] Prashanth Menon, Todd C. Mowry, and Andrew Pavlo. 2017. Relaxed operator fusion for in-memory databases: making compilation, vectorization, and prefetching work together at last. *Proc. VLDB Endow. (PVLDB)* 11, 1 (2017), 1–13.
[51] Jungmair Michael, Engelke Alexis, and Giceva Jana. 2024. HiPy: Extracting High-Level Semantics from Python Code for Data Processing. *Proc. ACM Program. Lang. (OOPSLA)* 8 (2024).
[52] mypyproject. 2024. Mypy: Static Typing for Python. https://www.mypy-lang.org/.
[53] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow. (PVLDB)* 4, 9 (2011), 539–550.

[54] Amadou Ngom, Prashanth Menon, Matthew Butrovich, Lin Ma, Wan Shen Lim, Todd C. Mowry, and Andrew Pavlo. 2021. Filter Representation in Vectorized Query Execution. In *Proceedings of the 17th International Workshop on Data Management on New Hardware (DAMON)*.
[55] OceanBase. 2024. OceanBase Database. https://github.com/oceanbase/oceanbase.
[56] ONNX Runtime. 2024. Python API documentation. https://onnxruntime.ai/docs/api/python/tutorial.html.
[57] Matteo Paganelli, Paolo Sottovia, Kwanghyun Park, Matteo Interlandi, and Francesco Guerra. 2023. Pushing ML Predictions Into DBMSs. *IEEE Trans. on Knowl. and Data Eng. (TKDE)* 35, 10 (2023), 10295–10308.
[58] Shoumik Palkar, James J. Thomas, Anil Shanbhag, Deepak Narayanan, Holger Pirk, Malte Schwarzkopf, Saman P. Amarasinghe, Matei A. Zaharia, and Stanford InfoLab. 2017. A Common Runtime for High Performance Data Analysis. In *Conference on Innovative Data Systems Research (CIDR)*.
[59] 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 (SIGMOD)*. 587–601.
[60] Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Laith Sakka, Krishna Pai, Wei He, and Biswapesh Chattopadhyay. 2022. Velox: meta's unified execution engine. *Proc. VLDB Endow. (PVLDB)* 15, 12 (2022), 3372–3384.
[61] PostgreSQL. 2024. PostgreSQL PL/Python. https://www.postgresql.org/docs/current/plpython-funcs.html.
[62] Fotis Psallidas, Yiwen Zhu, Bojan Karlas, Jordan Henkel, Matteo Interlandi, Subru Krishnan, Brian Kroth, Venkatesh Emani, Wentao Wu, Ce Zhang, Markus Weimer, Avrilia Floratou, Carlo Curino, and Konstantinos Karanasos. 2022. Data Science Through the Looking Glass: Analysis of Millions of GitHub Notebooks and ML.NET Pipelines. *SIGMOD Rec.* (2022), 30–37.
[63] Mark Raasveldt, Pedro Holanda, Hannes Mühleisen, and Stefan Manegold. 2018. Deep Integration of Machine Learning Into Column Stores. In *Proceedings of the 21st International Conference on Extending Database Technology (EDBT)*. 473–476.
[64] Mark Raasveldt and Hannes Mühleisen. 2016. Vectorized UDFs in Column-Stores. In *Proceedings of the 28th International Conference on Scientific and Statistical Database Management (SSDBM)*.
[65] Mark Raasveldt and Hannes Mühleisen. 2019. DuckDB: an Embeddable Analytical Database. In *Proceedings of the 2019 International Conference on Management of Data (SIGMOD)*. 1981–1984.
[66] Karthik Ramachandra, Kwanghyun Park, K. Venkatesh Emani, Alan Halverson, César Galindo-Legaria, and Conor Cunningham. 2017. Froid: optimization of imperative programs in a relational database. *Proc. VLDB Endow. (PVLDB)* 11, 4 (2017), 432–444.
[67] Rathijit Sen. 2024. Inference of ML Models in SQL Server via External Languages. https://techcommunity.microsoft.com/t5/sql-server-blog/inference-of-ml-models-in-sql-server-via-external-languages/ba-p/2216226.
[68] Maximilian Schleich and Dan Olteanu. 2020. LMFAO: An engine for batches of group-by aggregates: layered multiple functional aggregate optimization. *Proc. VLDB Endow. (PVLDB)* 13, 12 (2020), 2945–2948.
[69] Robert Schulze, Tom Schreiber, Ilya Yatsishin, Ryadh Dahimene, and Alexey Milovidov. 2024. ClickHouse - Lightning Fast Analytics for Everyone. *Proc. VLDB Endow. (PVLDB)* 17, 12 (2024), 3731–3744.
[70] Hesam Shahrokhi, Amirali Kaboli, Mahdi Ghorbani, and Amir Shaikhha. 2024. PyTond: Efficient Python Data Science on the Shoulders of Databases. In *2024 IEEE 40th International Conference on Data Engineering (ICDE)*. 423–435.
[71] Amir Shaikhha, Mohammad Dashti, and Christoph Koch. 2018. Push versus pull-based loop fusion in query engines. *Journal of Functional Programming* 28 (2018).
[72] Amir Shaikhha, Mathieu Huot, Jaclyn Smith, and Dan Olteanu. 2022. Functional collection programming with semi-ring dictionaries. *Proc. ACM Program. Lang. (OOPSLA)* 6, Article 89 (2022).
[73] Moritz Sichert and Thomas Neumann. 2022. User-Defined Operators: Efficiently Integrating Custom Algorithms into Modern Databases. *Proc. VLDB Endow. (PVLDB)* 15, 5 (2022), 1119–1131.
[74] SparkSQL. 2024. Apache Arrow in PySpark. https://spark.apache.org/docs/latest/api/python/user_guide/sql/arrow_pandas.html.
[75] Leonhard Spiegelberg, Rahul Yesantharao, Malte Schwarzkopf, and Tim Kraska. 2021. Tuplex: Data Science in Python at Native Code Speed. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 1718–1731.
[76] Statsmodels. 2024. Statsmodels Document. https://www.statsmodels.org/stable/index.html.
[77] Ruby Y. Tahboub, Grégory M. Essertel, and Tiark Rompf. 2018. How to Architect a Query Compiler, Revisited. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 307–322.
[78] Jianying Wang, Tongliang Li, Haoze Song, Xinjun Yang, Wenchao Zhou, Feifei Li, Baoyue Yan, Qianqian Wu, Yukun Liang, ChengJun Ying, Yujie Wang, Baokai Chen, Chang Cai, Yubin Ruan, Xiaoyi Weng, Shibin Chen, Liang Yin, Chengzhong Yang, Xin Cai, Hongyan Xing, Nanlong Yu, Xiaofei Chen, Dapeng Huang, and Jianling Sun. 2023. PolarDB-IMCI: A Cloud-Native HTAP Database System at Alibaba. *Proc. ACM Manag. Data (PACMMOD)* 1, 2 (2023).
[79] Zhuangdi Xu, Gaurav Tarlok Kakkar, Joy Arulraj, and Umakishore Ramachandran. 2022. EVA: A Symbolic Approach to Accelerating Exploratory Video Analytics with Materialized Views. In *Proceedings of the 2022 International Conference*

on Management of Data (SIGMOD). 602–616.

[80] Zhifeng Yang, Quanqing Xu, Shanyan Gao, Chuanhui Yang, Guoping Wang, Yuzhong Zhao, Fanyu Kong, Hao Liu, Wanhong Wang, and Jinliang Xiao. 2023. OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. *Proc. VLDB Endow. (PVLDB)* 16, 12 (2023), 3728–3740.

[81] Zhenkun Yang, Chuanhui Yang, Fusheng Han, Mingqiang Zhuang, Bing Yang, Zhifeng Yang, Xiaojun Cheng, Yuzhong Zhao, Wenhui Shi, Huafeng Xi, Huang Yu, Bin Liu, Yi Pan, Boxue Yin, Junquan Chen, and Quanqing Xu. 2022. OceanBase: A 707 Million TpmC Distributed Relational Database System. *Proc. VLDB Endow. (PVLDB)* 15, 12 (2022), 3385–3397.

[82] Bowen Yu, Guanyu Feng, Huanqi Cao, Xiaohan Li, Zhenbo Sun, Haojie Wang, Xiaowei Zhu, Weimin Zheng, and Wenguang Chen. 2021. Chukonu: A Fully-Featured High-Performance Big Data Framework That Integrates a Native Compute Engine into Spark. *Proc. VLDB Endow. (PVLDB)* 15, 4 (2021), 872–885.

[83] Chen Zhang, Rongchao Dong, Haojie Wang, Runxin Zhong, Jike Chen, and Jidong Zhai. 2024. MAGPY: Compiling Eager Mode DNN Programs by Monitoring Execution States. In *2024 USENIX Annual Technical Conference (USENIX ATC 24)*. 683–698.

[84] Chenyang Zhang, Junxiong Peng, Chen Xu, Quanqing Xu, and Chuanhui Yang. 2024. IMBridge: Impedance Mismatch Mitigation between Database Engine and Prediction Query Execution. In *Companion of the 2024 International Conference on Management of Data (SIGMOD)*.

Received October 2024; revised January 2025; accepted February 2025