IMBridge: Impedance Mismatch Mitigation between Database Engine and Prediction Query Execution

Chenyang Zhang
East China Normal University[†]
cyzhangecnu@stu.ecnu.edu.cn

Junxiong Peng
East China Normal University[†]
jxpeng@stu.ecnu.edu.cn

Chen Xu[*]
East China Normal University[†]
cxu@dase.ecnu.edu.cn

Quanqing Xu
OceanBase, AntGroup
xuquanqing.xqq@oceanbase.com

Chuanhui Yang
OceanBase, AntGroup
rizhao.ych@oceanbase.com

ABSTRACT

Prediction queries that apply machine learning (ML) models to perform analysis on data stored in the database are prevalent with the advance of research. Thanks to the prosperity of ML frameworks in Python, current database systems introduce Python UDFs into query engines for inference invocation. However, there are impedance mismatches between database engines and prediction query execution with this approach. In particular, the database engine is oblivious to the semantics within prediction functions, which incurs the repetitive inference context setup. Moreover, the evaluation of the prediction function is coupled with the operator, which results in an undesirable inference batch size with low inference throughput. To mitigate these, we propose a system called IMBridge, which leverages a *prediction function rewriter* to eliminate redundant inference context setup and introduces a *decoupled prediction operator* to ensure that the evaluation batch size matches the desirable inference batch size. In this demonstration, we will showcase how IMBridge addresses these mismatches and boosts prediction query execution.

CCS CONCEPTS

• **Information systems** → **Query optimization**.

KEYWORDS

Query Optimization, Machine Learning Prediction Query

ACM Reference Format:
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-Companion '24), June 9–15, 2024, Santiago, AA, Chile.* ACM, New York, NY, USA, 4 pages. https://doi.org/10.1145/3626246.3654754

[*]Chen Xu is the corresponding author
[†]Shanghai Engineering Research Center of Big Data Management

SIGMOD-Companion '24, June 9–15, 2024, Santiago, AA, Chile

ACM ISBN 979-8-4007-0422-2/24/06
https://doi.org/10.1145/3626246.3654754

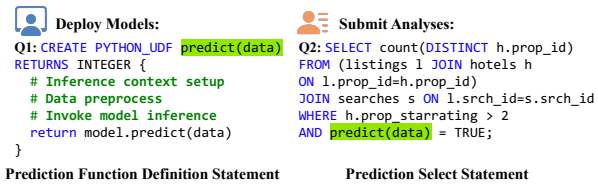


Figure 1: Prediction Query User Language

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 query that utilizes ML models to enhance data analysis, draw new insights, and discover latent patterns [4]. Due to the ease of programming and rich ecosystem of ML frameworks, current database engines leverage Python User-Defined Functions (UDFs) to support the prediction process.

Suppose a data scientist of an online travel agency applies prediction queries on the business data stored inside databases for decision-making. As shown in Figure 1, to deploy a trained ML model in database query engine, she specifies *Q1* through the prediction function definition statement. The statement includes a Python prediction function consisting of inference context setup, data preprocessing, and inference invocation. After that, she submits analyses through select statements with prediction function calls. For example, *Q2* 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 pushes the data to the `predict` function to predict whether a hotel will be recommended, and finally aggregates the result.

In database engines, the prediction query is transformed into an operator tree for execution, and prediction functions are embedded in operators as UDF expressions. The executor drives the operators iteratively to get query results. For each iteration, the operator first fetches a batch of data from downstream operators and then prepares the evaluation context and invokes the prediction function eagerly. However, when running prediction queries with this approach, there are impedance mismatches between prediction query execution and database engines, which leads to low execution performance. First, the database engine handles prediction functions as common UDFs, which is oblivious to the semantics

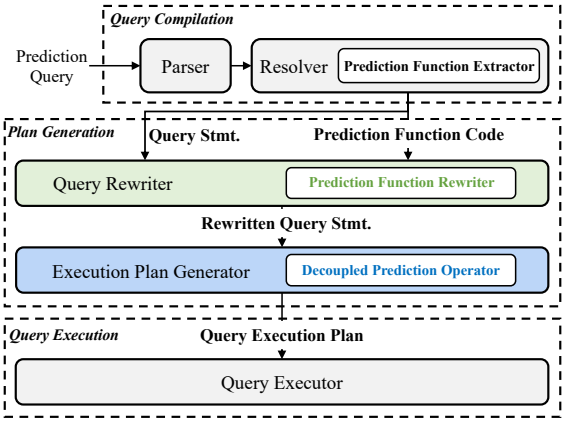


Figure 2: System Architecture of IMBridge

of inference. This causes the identical inference context setup to be executed repetitively. To address this, we propose a *prediction function rewriter* to hoist the inference context-building process, thus avoiding unnecessary inference context rebuildings. Furthermore, the database engine evaluates the prediction function as a UDF expression coupled with an operator, which results in a gap between the UDF evaluation batch size and the desirable inference batch size. This in turn causes a low inference throughput and leads to a high latency for the entire query. Therefore, we introduce a *decoupled prediction operator*, which specifically controls the batching evaluation process of each prediction function to match the desirable inference batch size.

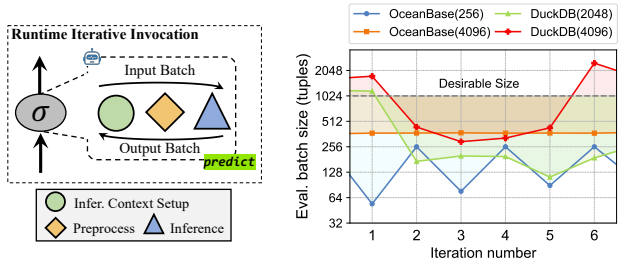
We have developed a system called IMBridge by extending the query engine of OceanBase [5, 6]. Our experiments show that IMBridge can achieve an 18.2x speedup over OceanBase. In this demonstration, we showcase two aspects: 1) how the prediction function rewriter eliminates the repetitive construction process of inference contexts, and 2) how the decoupled prediction operator ensures a desirable inference batch size for each prediction function invocation and speeds up query execution.

2 DESIGN AND IMPLEMENTATION

IMBridge is designed to run prediction queries inside database systems, which follows the architecture of traditional query engines. As illustrated in Figure 2, the compiler first parses the prediction query to an abstract syntax tree (AST) and resolves it with the schema. It also extracts the prediction function code using the prediction function extractor. Next, the query rewriter employs the prediction function rewriter to hoist the inference context setup and bind the inference context to the query statement. Then, the execution plan generator takes the rewritten query statement with inference context and generates the execution plan with the decoupled prediction operator. During execution, the prediction operator acquires the desirable inference batch size parameter and evaluates the prediction function according to this parameter.

2.1 Prediction Function Rewriter

Database engines handle prediction functions as common UDFs and are agnostic of the inference semantics inside them. As shown



(a) Repetitive Inference Context Setup (b) Undesirable Inference Batch Size

Figure 3: Impedance Mismatch in Prediction Queries

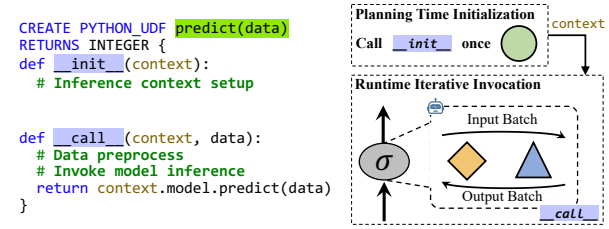


Figure 4: Prediction Function Rewrite

in Figure 3(a), prediction functions contain an indispensable routine of inference context setup, such as model loading and computation resource allocation. Due to the semantic unawareness, the inference context has a transient lifetime during the runtime iterative invocation. As a result, each function invocation leads to an overhead caused by rebuilding the same inference context.

Manual Hoisting Rewrite Interface. It is straightforward to introduce a new interface to let users implement the function in another way to hoist the inference context setup manually. Recent work, like YeSQL [3], introduces this kind of UDF interface to support manual hoisting. Here, we implement a similar interface in IMBridge. As depicted in Figure 4, instead of the plain UDF codes shown in *Q1*, the interface divides the prediction function into an initialization method `__init__` and an evaluation method `__call__`. To conduct the rewrite, users have to put the inference context setup codes in the `__init__` method and place the preprocessing and ML framework invocation codes in the `__call__` method. At planning time, the `__init__` method will be called to build the inference context and bind the context to the query plan for runtime reference. During execution, the `__call__` method is invoked per iteration to do the evaluation. Since the initialization method is executed once for each prediction function, this approach can avoid the repetitive inference context setup.

Automatic Hoisting Rewrite Algorithm. The manual hoisting approach requires users to rewrite the original function codes to the interface form manually. However, this approach requires users to identify the inference context setup codes and take extra effort to handle the rewrite. Especially for the deployed functions, users have to manually modify every function and make sure that the rewriting process does not have an impact on running queries. This motivates us to seek an approach to automatically detect inference

context setup codes in the UDF and hoist them to the initialization method safely. We observe that the query execution process can be mapped to equal loop program structures. The iterative invocation for each operator represents one of the loop structures, and UDFs embedded in the operator are function call expressions evaluated in the local scope of the loop body. From the semantics of the rewrite interface, the inference context setup codes (expressions and statements) produce the same value in every iteration of the entire query execution loop. Hence, we can model the automatic hoisting rewrite problem as a *Loop Invariant Code Motion* (LICM) problem in compilation optimization.

The original LICM algorithm detects loop invariant codes globally and moves them outside the loop to avoid redundant computation. To implement this algorithm, the general practice is to compile both the query plan and UDF codes into a unified intermediate representation (IR), search loop invariant codes on the IR globally, and transform the IR. Clearly, this approach has to introduce non-trivial polyglot compilation, which incurs non-negligible compilation overhead. However, variables defined in the prediction function are only valid under the local scope and do not affect variables in query operators or other UDFs. Hence, we can narrow down the detection space of loop invariant codes to the respective local scopes of the prediction functions.

We propose a local LICM algorithm on AST rather than the global LICM algorithm on unified IR. Our approach obtains the AST for each prediction function with a lower compilation overhead and performs the loop invariant code detection locally with less detection time compared to the global LICM algorithm. In particular, our algorithm takes the original prediction function code as input and outputs the code rewritten to the above interface form. It consists of two phases: marking and hoisting. In the marking phase, we traverse the AST obtained from the Python `ast` library, and select statements and expressions whose operands are either (1) constant, (2) defined outside the loop, or (3) marked as loop invariant, into a code set. In the hoisting phase, we need to move the loop invariant codes outside the loop, i.e., the `__init__` method in our case, and then transform the rest of the codes and place them into the `__call__` method.

2.2 Decoupled Prediction Operator

Most ML frameworks are optimized for batch processing, and the efficiency of ML inference is sensitive to the batch size. Typically, a *desirable inference batch size* for high throughput varies widely with different ML frameworks and models. Hence, it is reasonable to choose a suitable batch size for different prediction functions. In database engines, the prediction function is treated as a UDF expression coupled with a certain operator, like scan or selection. The inference is executed according to the *UDF evaluation batch size* which is determined by the operator. Commonly, the UDF evaluation batch size does not match the desirable inference batch size of prediction functions due to the following reasons.

First, the UDF evaluation batch size is directly constrained by the system-level inter-operator transfer batch size parameter of database engines. For engines that apply the tuple-at-a-time query execution paradigm, tuning this parameter is infeasible, since this paradigm does not have the batch processing ability by the system design. For engines that leverage the batching execution paradigm, an identical transfer batch size is adopted for each query in the system. Adjusting the inter-operator transfer batch size has to be handled manually and will affect all queries in the system. Moreover, some database engines, like OceanBase and DuckDB, set this parameter in their kernel codes, which requires recompiling the kernel for parameter change every time.

Second, the UDF evaluation batch size is also affected by the data processing logic of the operator to which the UDF is attached. For instance, UDFs on the scan operator are limited by the strategy of fetching data from tables, and UDFs on the selection operator are influenced by filter conditions. Therefore, even if we obtain a desirable value using prior knowledge and set the system-level inter-operator transfer batch size manually, it is impossible to perform inference according to the desirable inference batch size. In Figure 3(b), we illustrate this situation by sampling the evaluation batch size of the prediction function with *Q2*. The result shows that the evaluation batch size is yet frequently undesirable to the inference despite changing the inter-operator transfer batch size.

To sum up, the coupled design between the UDF expression and the operator leads to the issue that the UDF evaluation batch size does not match the desirable inference batch size. Therefore, we propose a novel decoupled prediction operator to address this issue. Our operator is an unary operator, which consumes data transferred from the downstream operator and produces prediction results. Instead of attaching prediction functions to other operators, IMBridge generates a standalone prediction operator for every prediction function in the query. Each operator has an independent desirable inference batch size parameter to fit the requirements of different prediction functions. To guarantee a desirable inference batch size for each prediction function invocation, the operator controls the batch pushed to the prediction function based on the amount of data transferred from downstream. In particular, batches that are equal to the desirable size are directly sent to the prediction function for inference. For batches that are smaller than the desirable size, the operator stores them in a buffer, and the prediction function is called only when the data in the buffer accumulate to desirable. For batches that exceed the desirable size, the operator slices them into suitable segments and feeds them sequentially to the prediction function. In addition, we introduce a profile-based tuner to assist in the acquisition of the desirable inference batch size. The tuner collects inference efficiency statistics and searches for the desirable value employing the AIMD scheme [2].

3 DEMONSTRATION

We have developed a web-based user interface (UI) to illustrate the above boosting techniques of IMBridge. The audience can define Python prediction functions with and issue queries with the code editor on the data residing in the database. Test tables include public datasets from Kaggle website (Expedia, Flights, and Hospital) and TPCx-AI [1] benchmark. The audience can also choose preset queries and functions in our UI. We provide the audience with prediction functions using three representative ML frameworks: scikit-learn, ONNX Runtime, and PyTorch. By switching the system setting, the audience can run queries on OceanBase or IMBridge. In particular, on IMBridge, we provide audiences with options to

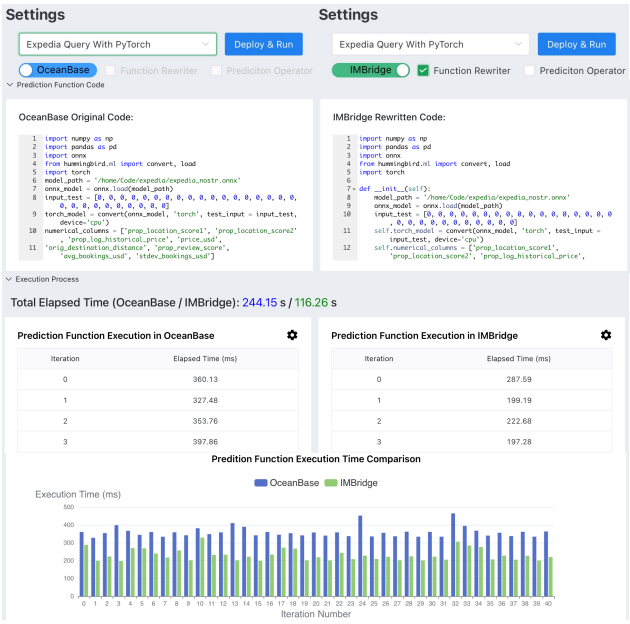


Figure 5: Demonstration of Prediction Function Rewriter

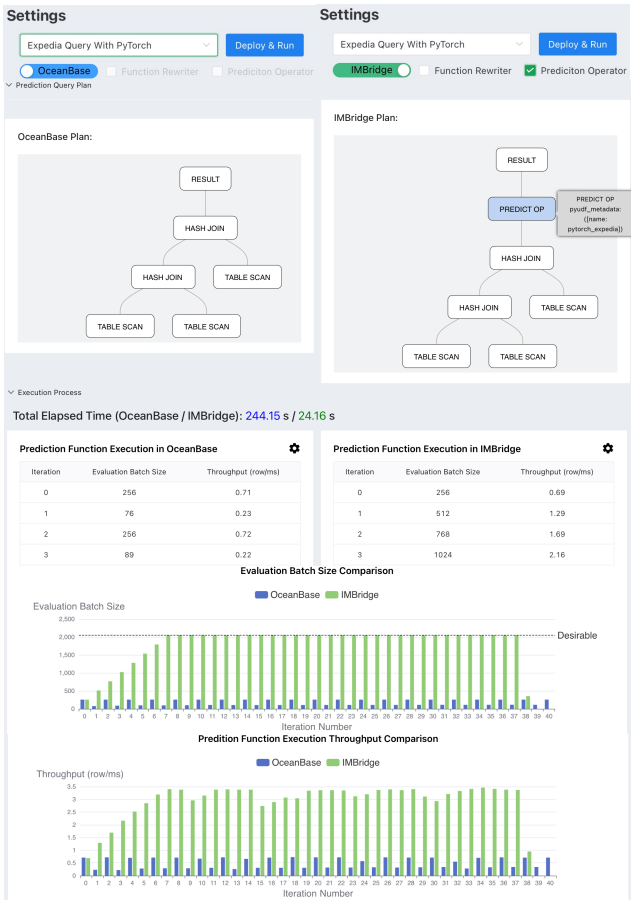


Figure 6: Showcase of Decoupled Prediction Operator

enable the above optimizations via checkboxes. Moreover, our UI demonstrates our optimization effects in the analysis report panel.

Scenario 1: Prediction Function Rewriter. To observe the optimization of *prediction function rewriter*, the audience will select a query with a prediction function, choose system settings, and finally click the "Deploy&Run" button. Specifically, the audience will run the query on OceanBase as a baseline and then run the same query on IMbridge enabling the "Function Rewriter" option.

As depicted in Figure 5, IMBridge will automatically transform original UDF codes into the hoisting rewrite interface form, and call the `__init__` method at planning time to construct the inference context. In the analysis report, the audience will see the overall speedup from the total elapsed time of query execution compared between OceanBase and IMBridge. Our UI also records the prediction function execution time per iteration in tables and visualizes the data in the bar chart. The result indicates that the prediction function rewriter eliminates the redundant inference context setup, thus reducing the elapsed time of function invocation per iteration.

Scenario 2: Decoupled Prediction Operator. Similar to scenario 1, we demonstrate the *decoupled prediction operator* with the "Prediction Operator" option of IMBridge. As shown in Figure 6, the audience will first see the query plan differences illustrated as a tree view, which shows that IMBridge extracts the prediction function and generates a standalone prediction operator with the function attached to it. In the analysis report, the total elapsed time between OceanBase and IMBridge can showcase the overall improvement of the decoupled prediction operator. In the "Evaluation Batch Size Comparison" chart, the audience will see that IMBridge can ensure a desirable inference batch size for each prediction function invocation, while the evaluation batch size in OceanBase is far from desirable. The chart also shows the dynamic process of obtaining the desirable inference batch size. Finally, our UI compares the throughput of prediction function execution between OceanBase and IMBridge. The result indicates that the prediction operator reduces the entire query latency by increasing the throughput of prediction function execution.

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] Christoph Brücke et al. 2023. TPCx-AI - An Industry Standard Benchmark for Artificial Intelligence and Machine Learning Systems. *PVLDB* 16, 12 (2023), 3649–3661.
[2] Daniel Crankshaw et al. 2017. Clipper: A Low-Latency Online Prediction Serving System. In *NSDI*. 613–627.
[3] Yannis Foufoulas et al. 2022. YeSQL: "You Extend SQL" with Rich and Highly Performant User-Defined Functions in Relational Databases. *PVLDB* 15, 10 (2022), 2270–2283.
[4] Kwanghyun Park et al. 2022. End-to-End Optimization of Machine Learning Prediction Queries. In *SIGMOD*. 587–601.
[5] Zhenkun Yang et al. 2022. OceanBase: A 707 Million TpmC Distributed Relational Database System. *Proc. VLDB Endow.* 15, 12 (2022), 3385–3397.
[6] Zhifeng Yang et al. 2023. OceanBase Paetica: A Hybrid Shared-Nothing/Shared-Everything Database for Supporting Single Machine and Distributed Cluster. *Proc. VLDB Endow.* 16, 12 (2023), 3728–3740.