IMLane: Composable Framework for Efficient AI Function Execution in Database Engine

Chenyang Zhang
Linjun Lu
Qingfeng Pan
East China Normal University
{cyzhangecnu,linjunlu,qfpan}@stu.ecnu.edu.cn

Chen Xu*
Xianzhong Cao
East China Normal University
cxu@dase.ecnu.edu.cn
xzcao@geo.ecnu.edu.cn

Quanqing Xu
Chuanhui Yang
OceanBase, AntGroup
xuquanqing.xqq@oceanbase.com
rizhao.ych@oceanbase.com

ABSTRACT

Efficient execution of artificial intelligence (AI) functions within the database engine has become an essential requirement for AI-driven data analysis workflows. During the development of supporting AI functions in OceanBase, we identified two performance bottlenecks incurred by the internal design of the database engine. First, the Python UDF-based AI function is executed via thread-level parallelism, which causes ineffective parallel execution due to limitations of CPython runtime. Second, the scheduling of the AI function is tightly coupled with the database engine's scheduler. This scheduler fails to adapt to the diverse compute resource requirements of AI functions in different user scenarios, leading to the under-utilization of available compute resources. To address these common bottlenecks, we present a composable framework, called IMLane, that equips current database engines with efficient AI function execution. IMLane incorporates *process-level AI function execution* for effective parallel execution with low data transfer overhead, and *decoupled AI function scheduling* to match compute resource requirements. Our experiments on OceanBase and DuckDB show 7.48x and 5.04x improvements on average, respectively, after integrating IMLane. Moreover, IMLane demonstrates performance superiority over alternative solutions.

PVLDB Reference Format:
Chenyang Zhang, Linjun Lu, Qingfeng Pan, Chen Xu, Xianzhong Cao, Quanqing Xu, and Chuanhui Yang. IMLane: Composable Framework for Efficient AI Function Execution in Database Engine. PVLDB, 19(12): 4223 - 4236, 2026.
doi:10.14778/3827998.3828028

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/IM-DM4AI/IMLane0.

1 INTRODUCTION

With the rise of artificial intelligence (AI), a wide range of AI applications have emerged for automating complex workflows, uncovering critical insights from data, and empowering more intelligent decision-making. This trend drives efforts in both industry and academia to evolve labor-intensive data analysis workflows into AI-driven automated data analysis workflows [8, 24, 40, 84, 86, 89, 91, 95, 102, 107] by harnessing large language model (LLM)-based data agents. As depicted in Figure 1, we showcase a typical AI-driven data analysis workflow powered by the LLM-based agent. With this workflow, users are able to directly issue their data analysis questions to the data agent in natural language without manually writing any code. The data agent will understand the user question and orchestrate a series of actions consisting of various data analysis operations, such as data retrieval and AI model-based inference. Since LLM itself is not skilled at precise data processing, especially on private data residing in databases, and lacks the specialized capabilities for domain-specific inference tasks, the data agent employs tool calling and code generation mechanisms to compensate for this limitation and enhance the quality of generated responses. It invokes database engines via automatically generated SQL queries to query relevant data, and leverages AI function executors running generated custom model inference code, e.g. Python code, for specialized inference tasks.

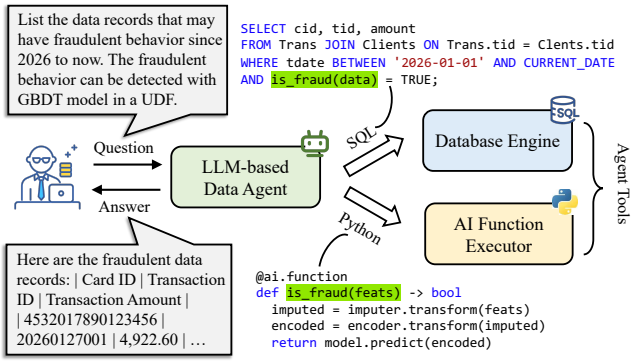


Figure 1: AI-driven data analysis workflow, where a data agent interprets natural language user questions and orchestrates data retrieval and model-based inference via SQL and Python-based AI function execution

Consider a real-world scenario from one of our customers, i.e., a bank manager who wants to identify potentially fraudulent credit card activity. He could turn to the data agent, backed by our OceanBase [96, 97] database, by submitting the following user question: List the card ID, transaction ID, and transaction amount of data records that may have fraudulent behavior since 2026 to now. Instead of pulling data out of the database and executing the AI function in an outside environment, OceanBase has supported running

*Chen Xu is the corresponding author

Proceedings of the VLDB Endowment, Vol. 19, No. 12 ISSN 2150-8097.
doi:10.14778/3827998.3828028

AI functions inside the database engine [101] through user-defined functions (UDFs). Given its ease of use and rich ecosystem of AI libraries, Python has emerged as the preferred language for implementing these AI functions [14, 41, 81, 101]. By incorporating an AI function executor in the database engine, OceanBase eliminates the overhead of data movement and ensures data security by keeping private data within the protected database environment. For the instance shown in Figure 1, by prompting the data agent that the fraudulent behavior can be detected with GBDT models in a Python UDF, it automatically generates the SQL queries and Python UDF code both executing in the database. In particular, the SQL query is a `SELECT` statement calling a Python UDF `is_fraud(feats)` to check whether a transaction has fraud risk based on the collected features of the joined table. The Python UDF performs the inference with a Gradient Boosting Decision Tree (GBDT) model using XGBoost. According to our investigation of this AI-based data analysis workflow, executing the AI function still accounts for over 80% of the overall workflow response time even after using the model caching and desirable batching in IMBridge [100]. Clearly, the AI function execution remains the dominant performance bottleneck, indicating that further improvements are both necessary and promising. Hence, in this work, we mainly focus on further optimizing the execution of such Python UDF-based AI functions.

To accelerate AI function execution in existing database engines like OceanBase, we identify two systematic bottlenecks based on our engineering observations. First, modern database engines typically explore a thread-level parallelism paradigm to support parallel AI function execution. While this paradigm is efficient and scalable for traditional analytical queries [15, 46, 59, 79], it hits a critical bottleneck with Python UDF-based AI functions. In particular, the parallel AI function execution is ineffective under the standard CPython runtime because of its inherently serial nature, primarily due to the Global Interpreter Lock (GIL). Second, the scheduling of AI functions is coupled with the scheduling of the database engine [4, 36], which is not matched with the diverse compute resource requirements of AI functions in different user scenarios. Moreover, this coupled scheduling forces the AI function to be scheduled following a partition-wise synchronous scheduling strategy, which results in under-utilization of available compute resources.

In light of these common performance bottlenecks, we present IMLane, a composable development framework for extending database engines to support efficient AI function execution, in this work. To achieve effective parallel execution, we adapt *process-level parallel AI function execution*. This approach employs process-level backend executors with low-overhead shared-memory-based interprocess data transfer. To address resource-mismatched scheduling, we introduce lightweight *decoupled AI function scheduling*. It first decouples the scheduling of AI functions from the database engine through a resource-aware per-function scheduler, and then utilizes a batch-wise asynchronous scheduling strategy to schedule AI functions cooperatively within the database engine. Additionally, with its modular and composable architecture, IMLane is not tightly coupled with OceanBase and can be easily integrated into other database engines for efficient AI function execution. Our experiments demonstrate that, after integrating the IMLane framework, database engines, OceanBase [96, 97] and DuckDB [13, 72], achieve speedups of 7.48x and 5.04x on average across various user scenarios, respectively. Furthermore, IMLane outperforms other state-of-the-art systems like IMBridge [101], Ray Data [74, 75], and SparkSQL [62].

In the rest of this paper, we first introduce the background of AI functions and highlight the motivations of our work in Section 2. In particular, this work makes the following contributions.

- We adapt well-established *process-level parallel AI function execution* to the database engine in Section 3, which achieves an effective parallel execution.
- We introduce *decoupled AI function scheduling* to the database engine in Section 4, which ensures a lightweight, resource-matched scheduling for diverse user scenarios.
- We outline the composable architecture of IMLane and discuss our practices of plugging IMLane into representative database engines, i.e., OceanBase and DuckDB, in Section 5.
- We conduct experimental studies to showcase the effectiveness of IMLane in Section 6.

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

2 BACKGROUND AND MOTIVATION

In this section, we introduce the background of AI functions in Section 2.1 and motivate our work in Section 2.2.

2.1 Background

2.1.1 AI Function in Database. The rapid advancement of AI models has significantly changed the paradigm of data analysis. Compared to the traditional statistics-based approach, using AI models to perform data analysis has demonstrated advantages in handling complex predictive analytics tasks. Additionally, applying model inference in the database avoids transferring large amounts of data to external environments, thereby yielding significant performance gains [50, 52, 101]. These demands have driven modern databases to support model inference via AI functions within their query engine. After registering AI functions in the database, the query can invoke the AI function to perform the model inference. Typically, an AI function in the database is a UDF that wraps AI models, like machine learning (ML) models, deep learning (DL) models, and LLMs, to perform specific AI tasks, such as regression, classification, clustering, etc. Based on our observation in practice, unlike conventional OLAP queries which are bottlenecked by I/O or join operations, the efficiency of these queries largely depends on the compute-intensive AI functions. Thus, we focus on how to support efficient AI function execution within databases in this work.

2.1.2 Supporting AI Function in OceanBase. OceanBase [97] is an open-source enterprise-level database developed by Ant Group, which provides highly available ACID transaction processing as well as high-performance analytics processing. Thanks to its novel stand-alone and distributed integrated architecture [96], in the past few years, OceanBase has been deployed in hundreds of organizations ranging from small-scale to large-scale enterprises [92]. To satisfy the emerged AI application requirements like vector search [93] and in-database model inference, OceanBase plans to evolve from a traditional relational database to a database with AI capabilities in its recent iteration roadmap [80]. As one of the important

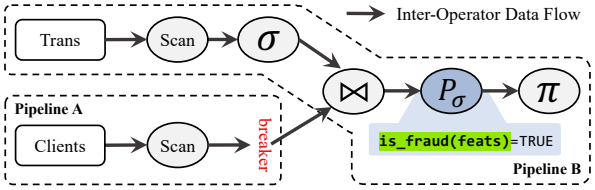


Figure 2: Execution plan with an AI function attached to a selection operator, split into pipelines for parallel execution

features, the OceanBase team has devoted significant efforts to enabling AI function execution in the database engine. To this end, OceanBase currently explores two kinds of development modes.

AI functions as built-in functions. In this development mode, users first discuss their model inference requirements with our development group. Then, the database engineer analyzes the requirement and implements AI functions using the development language of OceanBase, i.e., C++, in the codebase. Although this development mode produces high-performance AI functions by using native code, its development cycle is notably long [105]. First, we need to wrap the model inference in the built-in function either by importing AI libraries with C++ bindings or implementing the model inference manually. This is a non-trivial job since the AI ecosystem in C++ is not easy to use out of the box and maintaining external libraries in a large-scale C++ project is also cumbersome. Moreover, due to the modification of kernel codes, we have to recompile the database engine of OceanBase when adding a new AI function, which further prolongs the development cycle. Thus, we are also exploring alternative approaches to enable efficient AI functions while keeping development costs low.

AI functions as Python UDFs. In this development mode, we introduce Python runtime into our database engine so as to enable the development of AI functions as Python UDFs. Given its rich ecosystem of AI libraries, Python has become the top choice for developing AI applications. In addition, the user-friendly syntax and extensive collection of code repositories make LLM-based code generation viable [8, 102, 104]. Hence, the high development efficiency is guaranteed for both database developers and end users. Moreover, databases, e.g., PostgreSQL [60], DuckDB [14], and ClickHouse [9] already support Python UDFs within their engines, providing a useful reference for implementing Python UDF in OceanBase. However, when we apply existing methods in OceanBase, the execution performance of the AI function is unsatisfactory in practice. As a result, we think it is worthwhile to study how to support efficient execution of such Python-based AI functions.

2.1.3 Query Execution with AI Functions. When the query containing AI functions is submitted to the database, the database engine will translate it to a physical plan for execution. The physical plan is a tree consisting of relational operators, and the AI function is a function call expression attached to a specific operator. Figure 2 showcases the physical plan of Figure 1. In this plan, the AI function `is_fraud(feats)` is attached to a selection operator, denoted as P_σ, to filter out the fraudulent transactions using the GBDT model. To support pipeline data transfer between operators, database engines split the physical plan into pipelines [36, 44, 72, 79], i.e., a chain

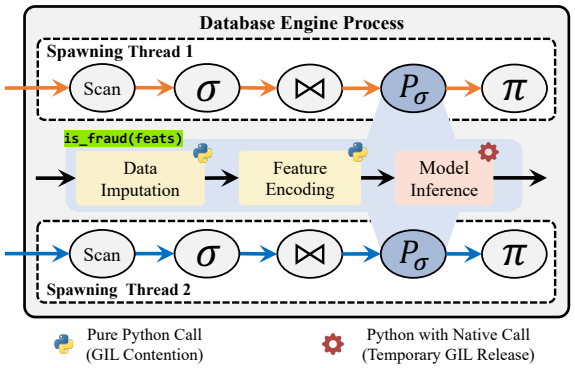


Figure 3: AI function execution under thread parallelism, limited by GIL contention and frequent context switching

of operators. The split is based on the pipeline breakers, which indicate dependency between adjacent operators with full data materialization. Then, the physical plan is scheduled for execution based on the topological order of pipelines. Parallel query execution is also achieved through this pipeline model. Each pipeline is executed in parallel with different data partitions. For instance, in Figure 2, the physical plan is partitioned into two pipelines at the `JOIN` operator since `JOIN` consists of a materialization phase like hash table building. The Pipeline A is first scheduled for execution with parallel partition-wise scan on table `Clients`. Pipeline B is then executed, scanning `Trans` in parallel while joining with the output of Pipeline A. During the execution of Pipeline B, the AI function `is_fraud(feats)` is also invoked concurrently.

2.2 Motivation

In this section, we present three key observations guiding the design of IMLane for efficient AI function execution in database engines.

2.2.1 Observation 1: Ineffective Parallel Execution. Modern database engines, including OceanBase, typically employ *thread parallelism* to support parallel query execution [36, 46, 54, 72, 79]. During execution, the query executor splits each pipeline into a set of subtasks and assigns them to multiple threads spawned within the database engine process. The AI function is accordingly invoked iteratively within the assigned thread in parallel.

In the initial version [101], OceanBase supports Python UDF execution by directly embedding a Python interpreter runtime into the database engine following existing solutions already utilized in current database engines like DuckDB [14] and PostgreSQL [60]. This implementation indicates the AI function is also executed in thread parallelism. While thread parallelism is effective for executing common analytical queries, it faces an ineffective parallel speedup issue due to the well-known Global Interpreter Lock (GIL) of CPython runtime [17, 18, 64, 69] when executing queries containing Python-based AI functions. As shown in Figure 3, consider a query with a parallelism degree of two, the execution of AI functions typically involves a mixture of pure Python code and native code running in the Python runtime shared across multiple threads in the process. The pure Python code typically contains script code of pure-Python function calls and the user-written control flow.

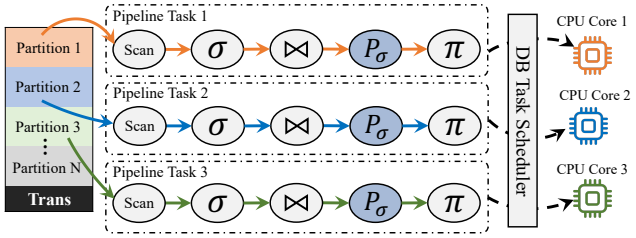


Figure 4: Coupled AI function scheduling, where scheduling is tied to pipeline tasks and limits resource utilization

For example, the `imputer.transform` and `encoder.transform` in `is_fraud` are two pure-Python function calls provided by the scikit-learn library. The native code is generally high-performance computational (HPC) kernel functions, e.g., PyTorch and NumPy, developed by experts of the ML community. Since the AI function execution is driven by the Python runtime, the native code is still called with Python glue code wrapping those HPC functions. For instance, the `model.predict` in `is_fraud` is a native call to XGBoost wrapped in Python. During parallel execution of AI functions, the segment of pure Python code only allows for a parallelism degree of one since the GIL is exclusive to just one thread. Although releasing the GIL during native calls allows for some degree of parallelism, the thread invoking the UDF must acquire the exclusive GIL again whenever the execution flow returns back to the Python glue code part. This causes frequent acquisition, contention, and release of the GIL among the threads of database engines, incurring substantial overhead from thread context switching. Therefore, executing AI functions in parallel at the thread level incurs considerable overhead because of the GIL and greatly hinders the effective speedup, i.e., the speedup of query execution is not improved with the increase of parallelism. This motivates us to study how to achieve *effective parallel AI function execution* in this work.

2.2.2 Observation 2: Resource-mismatched Scheduling. Modern database engines, including OceanBase, typically leverage a task scheduler to coordinate those pipeline tasks during parallel query execution. As illustrated in Figure 4, each pipeline, e.g., Pipeline B shown in Figure 2, is split into a set of pipeline tasks based on partitions. At runtime, the task scheduler dispatches pipeline tasks to compute resources, i.e., CPU cores, for execution. Each pipeline task iteratively performs batch-wise scan on the assigned partition of the base table, e.g., `Trans`. Then, the scanned batch is handled by each operator and the task scheduler collects the results from each task. Specifically, OceanBase adopts exchange-operator-based parallel query execution paradigm [46] similar to PostgreSQL [59]. In this paradigm [4], exchange operators are inserted at the point of pipeline breakers. These exchange operators act as the task scheduler, which manage the worker threads (usually, each thread is bound to one CPU core) and schedule pipeline tasks for execution. The AI function, attached to a relational operator within a pipeline, is scheduled together with its corresponding pipeline task, reflecting a coupled scheduling with the database engine.

Although this scheduling paradigm is well-suited for traditional analytical queries, we observe that its resource utilization mode fails to satisfy the diverse resource requirements of AI functions. In particular, the task scheduler of the database engine is mainly designed for bulk data analyzing workloads that run with CPU-only compute resources. However, the compute resources required by executing AI functions are rather diverse in real-world user scenarios (we will elaborate on them in Section 4). First, the AI function may be invoked in an incremental inference scenario that involves medium-sized data, and the bulk partition-wise scheduling can result in an insufficient number of compute resources scheduled for running the compute-intensive AI function. Second, the AI function may utilize some heterogeneous compute resources, e.g., GPUs, to accelerate model inference or delegate the computation to some remote servers. The coupled scheduling is clearly oblivious to this heterogeneity and runs the whole pipeline task synchronously, which causes resource under-utilization. In conclusion, this coupled scheduling does not match the varied resource requirements of AI functions. This motivates us to study how to realize *resource-matched scheduling* for AI functions in this work.

2.2.3 Observation 3: Common Bottlenecks in Database Engines. In practice, these performance bottlenecks are not specific to OceanBase. Specifically, we run the query shown in Figure 1 on both OceanBase and DuckDB and observe a performance gap of at least 46.5% compared to the ideal linear speedup when scaling compute resources. To address these common issues, a straightforward idea is to analyze the bottlenecks of query execution for each database engine case by case and then implement custom optimizations into different database engines. Clearly, this approach requires significant engineering effort and may lead to a large amount of duplicated code, which results in additional maintenance work. On the other hand, the database community has witnessed a growing trend towards composable or modular database architectures [56, 57] in recent years. By carefully decoupling the modules of the database system, e.g., SQL parser [38], planner [5], and execution engine [34, 42, 55], developers can optimize each module independently. Then, the decoupled system components can be easily integrated into existing database engines to extend current database systems with powerful functionality and efficiency enhancement. This motivates us to design IMLane as a *composable framework* not coupled with OceanBase in this work, which is portable enough to easily integrate it into other existing database engines.

3 EFFECTIVE PARALLEL EXECUTION

We review related work on mitigating ineffective thread parallelism in Section 3.1 and then describe adapting process-level AI function execution to database engines in Section 3.2.

3.1 Mitigating the Ineffective Parallelism

The Python community has invested considerable effort into addressing GIL limitations. Recent CPython versions have introduced sub-interpreters with per-interpreter GIL in Python 3.12 [16], and efforts such as PEP 703 [77] explore GIL-free implementations. However, these remain largely experimental [63, 66, 68] and are incompatible with many ML frameworks used in AI functions. Alternative Python interpreters like Jython [30] and GraalPython [21, 22] eliminate the GIL but struggle to support ML frameworks that rely on CPython-specific features for native calls. Recent works have proposed batching [33, 101] to improve AI function execution,

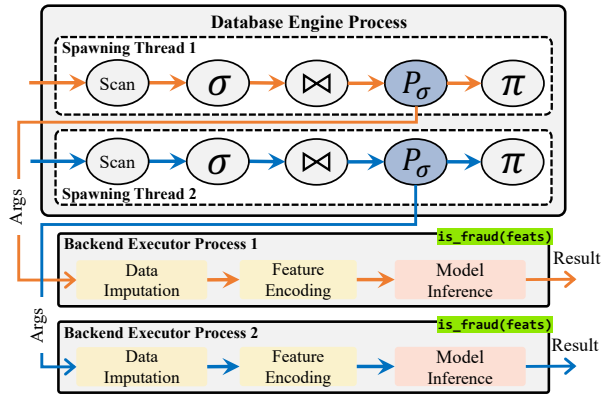


Figure 5: AI function execution under process parallelism, where independent processes avoid GIL contention and enable effective parallel execution

where the function is invoked only when a batch reaches a desirable size. This reduces invocation overhead and leverages internal parallelism in ML frameworks. However, the GIL issue remains and limits scalability as CPU cores scale up (see Section 6.2). In summary, thread-level parallelism cannot deliver effective parallel speedup for AI function execution due to GIL contention. Thus, we need an alternative execution mode.

3.2 Process-level Parallel Execution

We explore process-level parallel execution in our AI function executor. As shown in Figure 5, unlike thread-level execution, the process-level approach runs each AI function in a standalone backend executor process. Specifically, each executor process maintains its own Python interpreter, meaning that the GIL associated with each interpreter is exclusive to that process throughout AI function execution. When invoking an AI function, the database engine uses inter-process communication (IPC) for data transfer. This design offers three main benefits. First, it enables fully parallel execution of pure Python code without GIL contention. Second, it reduces GIL acquisition and context-switch overhead in the Python-to-native call path, as there is no contention for a single GIL and the interpreter optimizes GIL state marking with low overhead [69]. Third, the process-level execution shows superiority in fault isolation compared to the thread-level execution. That is, a failure that occurs during the execution of an AI function will have only a minor impact on the operation of the database engine. This feature is rather important especially for enterprise-level database products, like OceanBase, since it is crucial for maintaining high system stability in enterprise deployments. In summary, the process-level parallel execution is a promising architecture for achieving the effective speedup for parallel AI function execution.

3.2.1 Shared-Memory-based Data Transfer. Although the process-based execution is effective in overcoming the GIL issue, the data transfer with the database engine for function arguments and results introduces additional overhead. To mitigate this overhead, we begin by reviewing existing mechanisms for cross-system data transfer. Platforms such as Dask [11], and Python multiprocessing [65] are designed as general-purpose systems for scaling Python code execution. They depend heavily on heavyweight serialization/deserialization mechanisms, like pickle [67] and cloudpickle [10], to pass arbitrary Python objects across processes, which introduces considerable overhead. Similarly, PySpark [62] uses socket-based communication via Py4J [61], which also incurs serialization overhead and IPC latency. Inference serving systems such as Triton [45] add further network RPC overhead for data transfer.

Listing 1: Data conversion interface

```
template<typename ARGS, typename RET>
class DataConverter {
public:
    // ArrowLane is the intermediate type used by IMLane.
    ArrowLane ToLaneFormat(ARGS *args);
    void FromLaneFormat(ArrowLane lane, RET &ret);
} // End class DataConverter.
```

To support efficient data transfer between the database engine and the AI function executor, we need a low-latency IPC mechanism and a lightweight serialization method. Hence, instead of using network or socket-based I/O as the media of IPC between the database engine process and AI function executor process, we leverage shared memory to enable direct data access between processes. The shared memory mechanism allows us to establish a memory region directly accessible by both the database engine and the AI function executor, thereby avoiding the time-consuming software stack of network or socket-based I/O and substantially reducing IPC latency [28, 53, 98]. Since the shared memory region may be concurrently accessible by multiple processes, proper synchronization mechanisms must be implemented to coordinate read and write operations and ensure data consistency. To this end, we employ low-overhead semaphores to manage access to the shared memory region. We group these shared resources into logical units called *Lanes* to manage their lifecycle collectively. Each Lane consists of two buffers (one for input arguments, one for output results) and two corresponding semaphores to synchronize buffer accesses. During AI function execution, the caller and callee acquire an available Lane and perform data transfer through it. The complexity of low-level read/write operations and synchronization is encapsulated within Lane manipulation routines, making them transparent to developers. We choose Apache Arrow [1, 42, 55] as the unified intermediate format due to its portable and efficient in-memory representation. Arrow also aligns well with the internal data formats of column-based database engines (e.g., `ObVector` in OceanBase, `DataChunk` in DuckDB). Thus, converting the data format of database engines to Arrow introduces only minor serialization overhead. Furthermore, the zero-copy feature of Arrow works with shared memory to further reduce copying overhead across processes.

3.2.2 Template-based Data Converter Interface. Recall that IMLane is intended to be a composable component that can be integrated into different database engines. To accommodate the variety of intermediate data formats used by different database engines, we introduce a *template-based data converter interface*. As depicted in Listing 1, this interface is a template class with two core methods

Algorithm 1 Process-level parallel AI function execution

Input: AI Function *AIFunc* and an available *Lane* for data transfer

```
// AI function invocation from the database engine.
function invoke_func(call_args)
    converter ← an instantiated DataConverter
    args ← converter.ToLaneFormat(call_args)
    Lane.WriteAndPost(args, INPUT)
    res ← Lane.WaitAndRead(OUTPUT)
    converter.FromLaneFormat(res, ret)
    return ret // Return results back to the database engine.
// AI Function running in the process-level backend executor.
procedure run_func
    args ← Lane.WaitAndRead(INPUT)
    Call AIFunc with args to get results res
    Lane.WriteAndPost(res, OUTPUT)
```

to be implemented. To integrate IMLane into a specific database engine, database engineers utilize the template specialization feature of C++ [87] to provide concrete implementations with concrete argument and return types within database engines (e.g., `ObVector` for OceanBase, `DataChunk` for DuckDB) mapping to the `ArrowLane` type (a wrapped type of Arrow Table). Based on the specialized class, the compiler will automatically generate instantiation code for the routine of AI function execution, eliminating boilerplate and reducing integration effort to just implementing two conversion functions. This design makes IMLane easily pluggable into different database engines without modifying their core code, which is a key enabler for practical adoption.

3.2.3 Extensible Backend Executor. It is also worth noting that we design the backend executor responsible for running the AI function as an extensible component in the framework. In particular, each kind of backend executor inherits and implements the same set of interface methods, allowing new executors to be added into the framework. Currently, we incorporate two backend executors: the default Python executor and Ray executor. The default Python executor is a lightweight executor running AI functions directly in the CPython interpreter, requiring few external dependencies and offering a ready-to-use solution. In contrast, the Ray executor delegates the execution of AI functions to the Ray runtime [43, 74], which also provides the process-level parallel execution. This executor is particularly suitable for enterprises that already have Ray deployed. Specifically, we encapsulate each parallel executor as a stateful Ray `Actor`. The data wrapped in the `ArrowLane` is transferred to the Ray `Actor` via the global object store, which also accepts Arrow as the intermediate format. This extensible design allows IMLane to adapt to diverse production environments with little integration effort. As part of future work, we plan to implement additional backend executors for other inference engines such as ONNX Runtime [48] to further enrich the AI function ecosystem.

The detailed procedure of process-level parallel AI function execution is shown in Algorithm 1. When the database engine performs an AI function invocation, it first initializes the data converter with the implemented interface and converts the format of arguments into the Arrow-based intermediate format. The converted data is then written into the input buffer of the assigned Lane, after which the database engine posts a signal to notify the corresponding process of the AI function backend executor. This signal prompts the executor process to read the input and invoke the AI function within its process to obtain the results *res*. Next, the executor writes the results into the output buffer and posts a signal to notify the database engine. The database engine then reads the output data from the Lane and converts *res* back into its internal representation *ret* using the data converter. Finally, the *ret* are returned to the database engine to proceed with the rest of query execution.

4 RESOURCE-MATCHED SCHEDULING

We analyze resource utilization patterns and limitations of coupled scheduling in Section 4.1, and introduce decoupled scheduling for resource-matched execution in Section 4.2.

4.1 Deficiency of Coupled Scheduling

To systematically analyze the related resource utilization patterns and scheduling requirements for AI functions, we identify and examine the following typical user scenarios.

Scenario 1: Bulk Data Inference. In this scenario, users perform the AI function using host CPU-only computation on bulk data, such as full historical datasets, resulting in a data-intensive workload similar to a traditional OLAP workload. In general, data are distributed evenly across CPU cores. During execution, the CPU utilization remains consistently high, leaving little room for further parallelization. Therefore, the coupled scheduling is generally adequate for AI function scheduling in this scenario.

Scenario 2: Incremental Inference. In this scenario, users run the AI function on medium-sized data, which typically appears in the use case of incremental inference [12, 90, 106]. The AI function is performed on an interval of newly arrived data, e.g., on the last seven days of data, rather than full historical data. This scenario is essential in practice since inference is usually compute-intensive, and running it on the entire dataset would hinder timely analysis.

However, the coupled scheduling follows the partition-wise scheduling of database engines, where a large partition serves as the scheduling unit of pipeline tasks to improve scan I/O throughput. For example, OceanBase uses the size of multiple macro data blocks [47], i.e., 2 MB - 128 MB, as a partition, and DuckDB [15] sets a row group with 122880 rows as a partition. Consequently, splitting the medium-sized dataset into several large partitions may cause tasks not distributed evenly among available compute resources. For example, as shown in Figure 6, the incremental `Trans` table is split into only two partitions, causing the scheduler to assign pipeline tasks to just two CPU cores while leaving a third core idle. As a result, following the coupled scheduling fails to assign enough compute resources for AI function execution in this scenario.

Scenario 3: Heterogeneous Inference. In this scenario, the AI function, e.g., `is_fraud`, is accelerated with accelerators like GPUs, or delegated to a remote environment, such as dedicated model-serving services, e.g., Ray Serve [73], or a serverless ML endpoint, e.g., AWS SageMaker Endpoints [3, 37]. Since these heterogeneous resources can operate independently of the host CPUs, they offer additional compute resources beyond host CPU cores.

However, the coupled scheduling is unaware of these additional compute resources and schedules the AI function to run synchronously

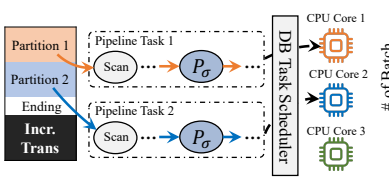


Figure 6: Load imbalance in partition-wise scheduling

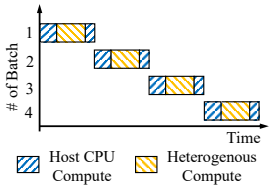


Figure 7: Sync hetero-unaware scheduling

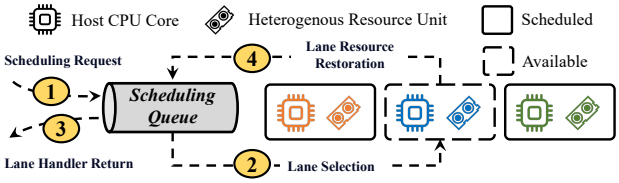


Figure 8: Resource-aware per-function scheduling, where functions are scheduled independently by resource needs

with other operations within each pipeline task. This mainly causes two issues. First, the task scheduler in database engines determines parallelism solely based on host CPU resources. It does not account for the independent parallelism of heterogeneous resources, leading to their under-utilization. Second, as depicted in Figure 7, when the AI function is offloaded to accelerators or remote environments, the host CPUs are idle. Similarly, when the computation is on host CPUs, the heterogeneous resources are idle. This under-utilization cumulatively degrades the end-to-end query performance. In summary, the coupled scheduling ignores the diverse resource requirements of AI functions and forces a *partition-wise synchronous scheduling strategy*, which under-utilizes compute resources.

4.2 Decoupled AI Function Scheduling

To fully utilize available compute resources for AI function execution, we decouple AI function scheduling from the database engine's pipeline task scheduling. We introduce a resource-aware per-function scheduler that scales independently based on each AI function's resource demands. On top of this, we propose a batch-wise asynchronous scheduling strategy that enables the database engine to schedule AI functions according to their scalability.

4.2.1 Resource-aware Per-function Scheduler. As depicted in Figure 8, we decouple the scheduling procedure of AI functions from the database engine via maintaining a standalone scheduler for each AI function. Each scheduler is aware of the compute resource requirements for its corresponding AI function and can scale independently according to the available compute resources. In particular, to represent the resource requirements for an AI function, we logically bind each AI function executor with one *Lane*, which aggregates a group of resource units, such as CPU cores, GPU units, and remote resource units. Accordingly, the number of available Lanes reflects the estimated resource demand of the AI function and determines the degree of parallelism available for scheduling. The number of Lanes can be determined in two ways. First, users

Listing 2: Asynchronous scheduling primitive

```
1 template<typename ARGS, typename RET>
2 class SchedPrimitive {
3 public:
4     // Asynchronous primitive, does not block the
          scheduling thread.
5   std::optional<std::future<RET>> AsyncSchedFunc(string
        &func, ARGS &args);
6 } // End class SchedPrimitive.
```

can manually specify it based on their knowledge of the AI function's workload. Second, our system can estimate it automatically. At runtime, the scheduler starts with a conservative Lane count and progressively increases it until performance stops improving or resource allocation fails, e.g., insufficient CPU cores, GPU memory, or remote workers to launch a new backend executor instance, adapting to workload variations without oversubscription. During the execution of query with AI functions, the pipeline tasks are first scheduled to threads by the task scheduler within the database engine. (1) Whenever the AI function is called within a pipeline task, it will send a scheduling request to its corresponding AI function scheduler rather than following the scheduling of the database engine. Specifically, the AI function scheduler mainly maintains a scheduling queue containing the schedulable units, i.e., *Lanes*. (2) Upon receiving the scheduling request, the scheduling queue will select an available Lane for scheduling. (3) Then, it will send the handler of this scheduled Lane to the pipeline task where the AI function is invoked, so that the pipeline task is able to utilize the data buffer within the Lane for data transfer and runs the AI function in the backend executor. (4) Once the execution is finished, the scheduled Lane will be returned to the scheduling queue, representing the restoration of compute resources.

Limitations. Our scheduler adopts a unified abstraction to support scheduling local and heterogeneous AI functions for lightweight design and generality. While this simplifies implementation, it may overlook differences in latency, data movement, and resource ownership, potentially limiting scheduling effectiveness. We leave more specialized scheduling for future work.

4.2.2 Batch-wise asynchronous scheduling strategy. Then, we can utilize this independently scaled scheduler to design a dedicated scheduling strategy specifically for AI functions. To this end, we propose a *batch-wise asynchronous scheduling strategy* for AI functions instead of using the partition-wise synchronous scheduling strategy adopted by the database engine. In particular, we first expose the *asynchronous scheduling primitive* of the per-function scheduler so that the database engine can work cooperatively with this scheduler to schedule the AI function to run asynchronously with other relational operations during query execution. As depicted in Listing 2, this scheduling primitive is also a template-based interface, which takes the name of the AI function and calling arguments as input and returns a `std::future` object wrapped in `std::optional` as output. In comparison to the synchronous scheduling primitive that blocks the scheduling thread waiting for the execution result, the asynchronous one returns the wrapped `future` object directly

Algorithm 2 Batch-wise asynchronous scheduling strategy

Input: AI Function *AIFunc* and SchedPrimitive instance *sched*

```
1:  // Initialize local variables for each AI function.
2:  pending ← an empty queue
3:  state ← SCHEDULE
4:  // Perform AI Function scheduling for each batch_n.
5:  function sched_batch(batch_n)
6:      switch state do
7:          case SCHEDULE:
8:              task ← sched.AsyncSchedFunc(AIFunc, batch_n)
9:              if task.has_value() then
10:                 pending.push_back(task.value())
11:                 Request for the next batch batch_{n+1}
12:             else state ← OUTPUT
13:         case OUTPUT:
14:             ready_futrs ← Poll pending futures
15:             if ready_futrs.size() then state ← SCHEDULE
16:                 Collect results from ready_futrs
17: // Wait for the rest of futures in the pending at last.
18: procedure sched_final
19:     while pending.size() do
20:         futr ← pending.pop_front()
21:         res ← futr.result() // Block waiting.
22:         Collect res into final results
```

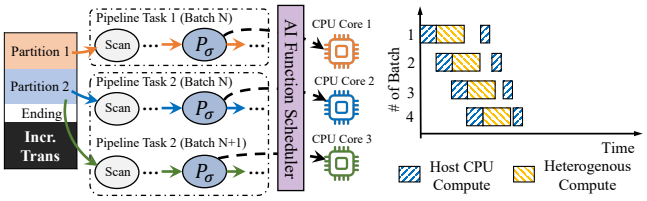


Figure 9: Load balance in batch-wise scheduling

Figure 10: Async hetero-aware scheduling

after calling the primitive. The execution result of the AI function is obtained from the wrapped `future` object at a given appropriate time in the future. Hence, before getting the execution result, the scheduling thread, which also executes the pipeline task in the database engine, is able to perform other relational operations on that pipeline task instead of making the host CPU idle during AI function execution. Second, to overcome the drawback of the partition-wise scheduling of the database engine, we need to seek for a more fine-grained scheduling unit than the coarse-grained large data partition. To achieve this goal, we notice that modern database engines, like OceanBase and DuckDB, typically explore the vectorized query execution paradigm to mitigate the virtual function call overhead of row-at-a-time paradigm and utilize SIMD instructions for acceleration. In this paradigm, the data partition for each pipeline task is further split into several small chunk batches [70]. During execution, the pipeline task will run each operator on the chain with each data batch iteratively. Thus, the AI function is inherently executed within a strategy that employs a more fine-grained data unit in the database engine. Accordingly, we can utilize this existing characteristic and combine it with the asynchronous scheduling primitive to schedule as many AI function tasks for execution as possible, so as to fully utilize the available compute resources managed by the resource-aware scheduler.

The detailed procedure of the batch-wise asynchronous scheduling strategy is shown in Algorithm 2. Given an AI function *AIFunc* and an instance of the SchedPrimitive interface *sched*, this strategy maintains two local variables for each AI function. In particular, *pending* is a queue used for storing those asynchronous AI function tasks, and *state* has two states, `SCHEDULE` and `OUTPUT`, to trigger scheduling new tasks or handling the output of AI function execution (Line 2 and 3). Let $batch_n$ be the current batch for scheduling during query execution, the scheduling strategy is performed to that batch according to the current state of *state*. When the current *state* is `SCHEDULE`, we will schedule the AI function using the asynchronous scheduling primitive shown in Listing 2. Once returned, we will determine whether the scheduling is successful, i.e., has resources available, by checking whether the `optional`-wrapped `future` contains a value (Line 9). If successful, we will add the `future` object into the pending queue, and request scheduling for the next batch $batch_{n+1}$. Otherwise, the *state* will change to `OUTPUT`, signaling that execution results should be handled first. When the current *state* is `OUTPUT`, we will poll each `future` in the *pending* queue and store those finished ones into *ready_futrs*. If *ready_futrs* is not empty, the *state* changes to `SCHEDULE` again, and we will collect the execution results (Line 14 to 16). To ensure that all the results of the AI function are obtained, at the end of AI query execution, we will perform blocking waits and collect results for every `future` left in the *pending* queue (Line 18 to 22).

We now discuss how this scheduling strategy influences the user scenarios described above. For the scenario 1, using batch-wise scheduling may increase the frequency of data transfer and scheduling. However, our experiments in Section 6.3 show this does not translate into measurable overhead. For scenario 2, as depicted in Figure 9, exploring batch-wise scheduling distributes AI function execution tasks evenly across available compute resources most of the time. For the scenario 3, as shown in Figure 10, performing asynchronous scheduling can realize the overlapped utilization for both host CPUs and heterogeneous compute resources, thus boosting the overall query performance.

5 COMPOSABLE FRAMEWORK DESIGN

We design IMLane as a pluggable, database-agnostic C++ framework that extends database engines to support efficient AI function execution by resolving the common bottlenecks above.

5.1 System Architecture

As shown in Figure 11, IMLane consists of three components, i.e., DBEnd Library, Coordinator, and Backend Executors. Specifically, the DBEnd library serves as the main entry point provided to database engine developers. It is designed as a plugin that can be compiled and linked into existing database engines, such as DuckDB and OceanBase. This library includes a set of DBEnd interfaces, i.e., the data conversion interface and the scheduling primitive, for developers to implement and call within the database engine

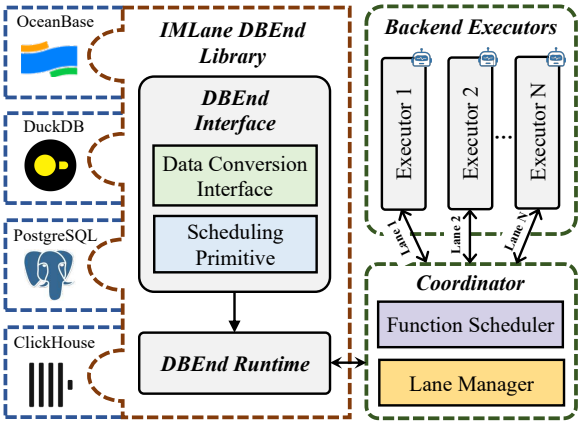


Figure 11: IMLane architecture with DBEnd library, coordinator, and backend executors

code. The DBEnd Runtime utilizes scheduling primitives and implemented data conversion interfaces to interact with the Coordinator and drive the scheduling of AI functions. The Coordinator serves as the core element of the IMLane framework. It includes the AI Function Scheduler, which handles scheduling requests from the DBEnd Runtime, and the Lane Manager, which manages the lifecycle of *Lanes* containing the inter-process shared resources and compute resource units. In addition, each Backend Executor embeds an execution runtime, e.g., a Python interpreter, in its corresponding process, which executes the specified AI function code and returns the execution result to the database engine via Lane.

5.2 System Integration

In practice, we have integrated IMLane into OceanBase and another open-source analytical database engine, DuckDB.

5.2.1 Integration with OceanBase. The codebase for integrating IMLane into OceanBase includes around 800 lines of code. Specifically, we encapsulate the core implementations in a new physical operator, called AI operator. This integration has two core parts: implementing the table converter interface and using the asynchronous scheduling primitive to realize the batch-wise asynchronous scheduling strategy. Following Listing 1, we specialize the `TableConverter` to realize the concrete data conversion logic for OceanBase's input and output type `ObVector`. Moreover, following Algorithm 2, we implement the batch-wise asynchronous scheduling strategy by reusing the pull-based vectorized volcano interfaces, such as `next()`, in OceanBase.

5.2.2 Integration with DuckDB. Similarly, we have integrated IMLane into DuckDB with around 500 lines of code. Also, we encapsulate the core implementations in an AI operator within DuckDB. The implementation of the table converter interface is much easier compared to OceanBase's integration since we can reuse the existing `ArrowConverter` codes to convert DuckDB's input type, `DataChunk`, and output type, `Vector`. Besides, the realization of the batch-wise asynchronous scheduling strategy is generally similar to OceanBase except for how the next batch is requested (Line 11).

Table 1: Experimental queries

ID	Application	Dataset	Model	User Scenario
Q1	Codeshare Prediction	Flights	RF	Bulk Data Inference
Q2	Rank Prediction	Expedia	GBDT	
Q3	Hardware Failure	TPCx-AI	SVM	Incremental Inference
Q4	Spam Detection	TPCx-AI	NB	
Q5	Antigen-Nanobody Binding Prediction	NbBench	DNN	Heterogeneous Inference
Q6	Price Prediction	TPCx-AI	RNN	
Q7	Sentiment Classification	SemBench	LLM	

Since DuckDB uses push-based vectorized execution, the operator must return the state `NEED_MORE_INPUT` to the query engine, prompting it to push the next batch.

6 EXPERIMENTAL STUDIES

We first describe the experimental setting in Section 6.1. Then, we evaluate the efficiency of IMLane in Section 6.2 and Section 6.3. In addition, we provide further discussion in Section 6.4.

6.1 Experimental Setting

In the following, we describe our experimental settings.

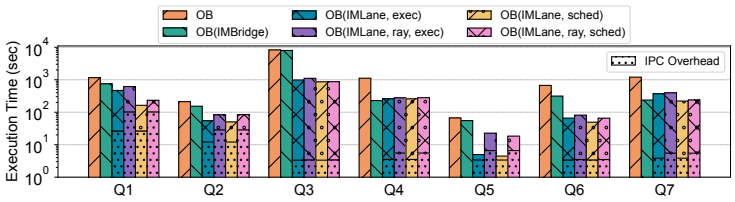
Hardware. We deploy database engines and IMLane on a server node with an Intel(R) Xeon(R) Gold 6240R CPU @ 2.40GHz (48 physical cores), 128 GB DDR4 RAM, a 24 TB RAID controller, and an NVIDIA Tesla V100 GPU. For the remote environment, we use four server nodes with Intel(R) Xeon(R) Gold 6248R CPU @ 3.00GHz (48 physical cores), 256 GB DDR4 RAM, and one NVIDIA RTX A6000 GPU (48 GB GDDR6) per node.

Software. The software stack comprises Ubuntu 20.04 running in Docker 26.1.4, Python 3.11, OceanBase Paetica 4.3, DuckDB 0.10.1, PySpark 3.5.7, Ray 2.49.2, and vLLM 0.19.1.

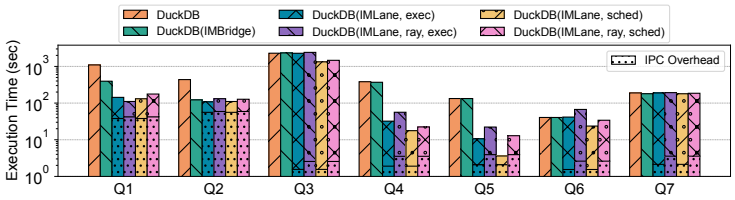
Datasets. We use five datasets in the experiment. We apply two publicly available datasets, Expedia and Flights, widely adopted in prior works [51, 52, 101], and three benchmark datasets, NbBench [103], TPCx-AI [6], and SemBench [35] to evaluate the optimization effects and usability in real-world applications.

Queries. Table 1 shows the queries and AI functions generated by a data agent backed by codex [49] to address the specific user questions in the three scenarios outlined in Section 4.1.

- *Bulk Data Inference (Q1-Q2)*: The queries and AI functions follow the data analysis semantics in Raven [52] and IMBridge [101], using Random Forest (RF) and Gradient Boosting Decision Tree (GBDT) as the ML models. Each dataset is scaled to 10 GB (20M-40M rows) after `JOIN` by replicating the original dataset.
- *Incremental Inference (Q3-Q4)*: The queries and AI functions for this scenario are generated from TPCx-AI [6], using SVM and NB as the ML models. To match this scenario, Q3 and Q4 are executed on incremental tables with 1M rows, representing data collected in ten days, following MaR [12] and FEBench [106].
- *Heterogeneous Inference (Q5-Q7)*: Q5 uses a GPU-accelerated DNN model locally for the binding prediction task from NbBench [103]. Q6 deploys an RNN model remotely with Ray Serve [73] on remote CPU workers. Q7 uses the sentiment classification

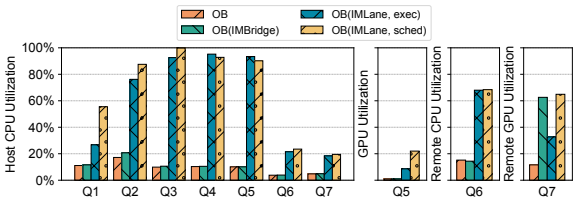


(a) OceanBase

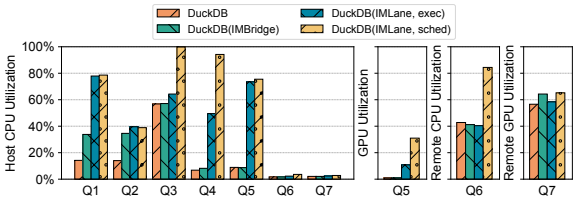


(b) DuckDB

Figure 12: End-to-end execution time, data transfer overhead, and Ray backend on OceanBase and DuckDB



(a) OceanBase



(b) DuckDB

Figure 13: Resource utilization on OceanBase and DuckDB, where IMLane improves resource utilization

query 3 from the SemBench Movie reviews dataset, deploying Qwen3 1.7B [71] on remote GPU servers with vLLM [88]. All three queries use a 1M-row data table by default.

6.2 Efficiency of Effective Parallel Execution

In this section, we study the efficiency of effective parallel AI function execution on OceanBase and DuckDB with sixteen host CPUs.

6.2.1 End-to-End Performance. We first study end-to-end performance of process-level parallel AI function execution, denoted as (IMLane, exec), and compare them against IMBridge [101], which improves thread parallelism via batching described in Section 3.1.

OceanBase. Figure 12(a) presents the experimental results on OceanBase (abbreviated as OB), which explores thread-level parallel execution. Specifically, OB(IMLane, exec) delivers a substantial performance improvement, running on average 5.47x faster than OB. Although OB(IMBridge) mitigates the ineffective thread-level execution, it only runs 2.03x faster than OB. This is because IMBridge achieves the performance improvement by reducing runtime switch overhead of Python invocation, shown in Q4, and utilizing the internal parallelism of the underlying ML frameworks within to realize the parallel speedup, shown in Q1, Q2, Q5, Q6, and Q7. It remains constrained by the GIL in CPython and fails to gain the speedup, especially for AI functions with lots of pure-Python preprocessing operations, like depicted in Q3. In contrast, OB(IMLane, exec) can parallelize the whole AI function for all experimental queries by leveraging the process parallelism to avoid the GIL limitations.

DuckDB. Figure 12(b) shows the experimental results for DuckDB. DuckDB(IMLane, exec) achieves a speedup of 3.33x on average over DuckDB, while DuckDB(IMBridge) is only 1.4x on average faster than DuckDB. This speedup is even lower than that in OB since DuckDB adopts a larger batch size (2048) than OceanBase (256), thus leading to less optimization space for desirable batching. Instead, DuckDB(IMLane, exec) exhibits significant performance gains for all experimental workloads, except for Q3, Q6 and Q7, without the impact of batch size. For Q3, its performance in DuckDB(IMLane, exec) is still constrained by the load imbalance incurred by the coupled scheduling. For Q6 and Q7, the Python UDF in DuckDB is able to release the GIL during the remote inference while that in OceanBase holds the GIL for the entire execution of the AI function.

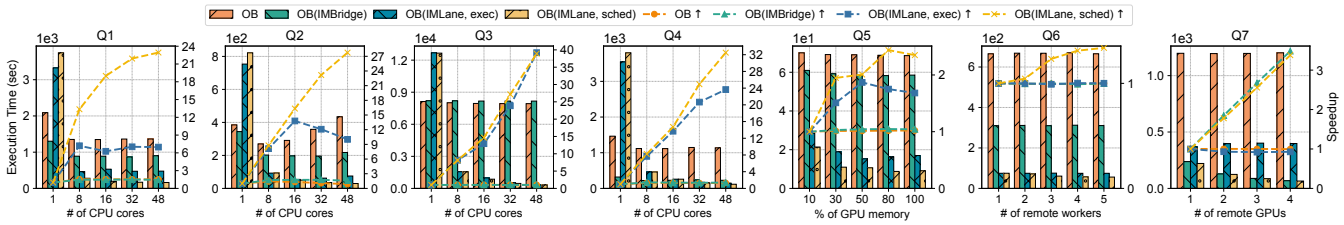
6.2.2 Overhead of Data Transfer. We further measure the extra overhead of data transfer introduced by the process-level parallel execution, shown at the bottom of bars depicted in Figure 12(a) and Figure 12(b). In particular, the extra overhead of IPC in IMLane only accounts for 14.8% and 15.5% of the execution time on average in OB(IMLane, exec) and DuckDB(IMLane, exec), respectively. Also, compared to the speedup after using the process parallelism, our shared memory-based data transfer has low overhead and is amortized by its significant performance benefits. To sum up, the process-level parallel execution of IMLane is able to empower the database engine with effective parallel AI function execution without introducing much overhead of data transfer between processes.

6.3 Efficiency of Resource-matched Scheduling

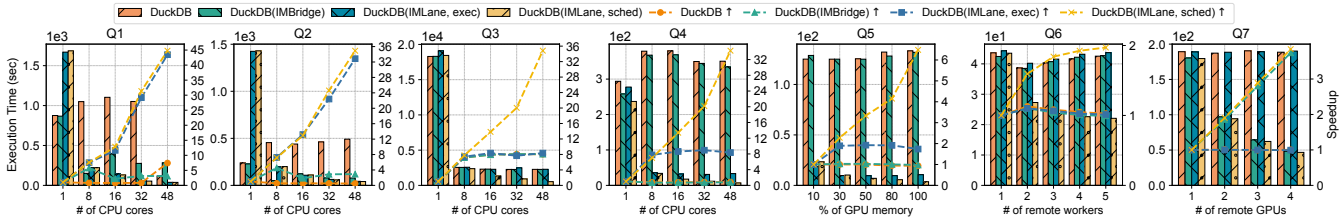
In this section, we study the efficiency of resource-matched AI function scheduling on OceanBase and DuckDB on the basis of the above process-level parallel execution.

6.3.1 End-to-End Performance. We first study the end-to-end performance of decoupled AI function scheduling, denoted as (IMLane, sched), and compare them against (IMLane, exec), which couples the scheduling of AI functions with that of database engines.

OceanBase. As shown in Figure 12(a), OB(IMLane, sched) delivers a further 1.37x improvement on average over OB(IMLane, exec). In particular, for bulk data inference Q1, OB(IMLane, sched) still achieves 1.09x speedup over OB(IMLane, exec). Since OceanBase is a disk-based database with disk I/O operations, this performance gain of decoupled scheduling can also be attributed to the overlapping of disk I/O during table scanning and AI function execution. Besides, for Q3-Q7, OB(IMLane, exec) does not match with the resource requirements, i.e., e.g., CPUs, GPUs, and remote servers in these



(a) OceanBase



(b) DuckDB

Figure 14: Resource scalability study, showing that IMLane achieves improved scalability with increasing resources

scenarios, and is clearly not suitable for AI function scheduling, thus leading to 17.6% slower than OB(IMLane, sched) on average.

DuckDB. As shown in Figure 12(b), DuckDB(IMLane, sched) runs 1.51x faster on average over DuckDB(IMLane, exec). For Q1 and Q2, the performance gain from DuckDB(IMLane, sched) is relatively modest. This is because DuckDB is a memory-based database with less I/O overhead than OceanBase, so that it already saturates CPU compute resources throughout most of AI function execution, leaving little room for further scheduling optimizations. Additionally, DuckDB(IMLane, sched) brings substantial improvements for Q3–Q6, achieving a speedup of 1.71x in these user scenarios. Note that since DuckDB employs a coarser-grained data partition as scheduling units than OceanBase, it is more prone to load imbalance issues in incremental inference scenarios. Thus, DuckDB(IMLane, sched) with its batch-wise scheduling strategy shows superior performance gains over OB (IMLane, sched) for Q3 and Q4.

6.3.2 Resource Utilization Improvement. We measure the averaged utilization rate of compute resources during AI function execution to showcase the utilization improvement of our resource-matched scheduling, depicted in Figure 13(a) and Figure 13(b). First, the utilization rates of OceanBase, DuckDB and their IMBridge variants are generally low (less than 60%) due to the GIL issue. By introducing process-level execution, OB(IMLane, exec) and DuckDB(IMLane, exec) improve the utilization by 500.7% and 302.6% on average over OB and DuckDB, respectively. On top of this, OB(IMLane, sched) and DuckDB(IMLane, sched) improve the utilization further by 39.1% and 52.8% on average, respectively. Particularly, for Q5-Q7 with heterogeneous computation, the decoupled scheduling significantly improves the utilization rate for both host CPUs and heterogeneous resources. This verifies the effectiveness of asynchronous scheduling strategy for achieving the overlapped computation. In summary, these utilization improvements align with the observed reduction in end-to-end execution time.

6.4 Discussion

6.4.1 Resource Scalability Study. Next, we evaluate the scalability of IMLane by progressively increasing the primary bottleneck resource for each query, i.e., the number of CPU cores, GPU units split by GPU memory, remote workers, and remote GPUs in containers via Docker resource constraints, as shown in Figure 14(a) and Figure 14(b). The speedup is calculated by dividing the execution time with the minimum resources, i.e., the experiment result on the leftmost, by the execution time after increasing the resources. In particular, the scalability of OceanBase, DuckDB and their IMBridge variants are generally constrained for most queries, which is caused by the Python GIL issue. By introducing the process-level execution, OB(IMLane, exec) and DuckDB(IMLane, exec) allows effective parallel AI function execution bypassing the GIL issues. This optimization exhibits speedup enhancement as compute resources increase to some extent. However, the speedup of them finally stops increasing as more compute resources are added, especially for Q3-Q7. This occurs because the coupled scheduling ignores the compute resources needed by AI functions and fails to scale when additional resources are added. In comparison, OB(IMLane, sched) and DuckDB(IMLane, sched) leverage the decoupled scheduling to scale AI function independently, which maintains the speedup enhancement as the amount of resources grows.

6.4.2 Performance of Ray Backend Executor. Furthermore, we study the IMLane's optimization effect after replacing the default backend executor with the Ray [74] executor, denoted as ray. As depicted in Figure 12(a) and Figure 12(b), OB(IMLane, ray, exec) achieves 3.74x average speedup over OB, and OB(IMLane, ray, sched) delivers further average 1.35x speedup over it. DuckDB(IMLane, ray, exec) achieves average 2.58x speedup over DuckDB, and DuckDB(IMLane, ray, sched) is further average 1.37x faster than it. In general, combining IMLane and Ray achieves similar effects for effective parallel execution and resource-matched scheduling, but is 61.6% on average slower in execution time than using the default executor. This

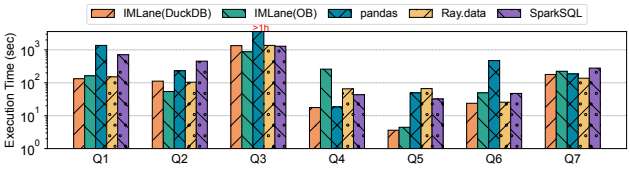


Figure 15: Performance comparison with pandas, SparkSQL, and Ray.data, where IMLane achieves comparable results

is largely due to the extra data transfer and invocation overhead for calling the Ray runtime than our default executors.

6.4.3 Comparison with Other Solutions. We compare IMLane with three alternative systems including pandas [102], SparkSQL [62], and Ray.data [75] that represent Python-based AI function execution paradigms. These alternative systems execute outside the database engine and pull data from OceanBase first. See Section 7 for detailed discussion and comparison. As shown in Figure 15, IMLane(OB) runs 2.65x faster than pandas and IMLane(DuckDB) is 4.02x faster than pandas, as pandas suffers from both high data transfer overhead and limited single-threaded execution. IMLane(OB) runs 1.87x faster than SparkSQL and 1.19x faster than Ray.data. Even better, IMLane(DuckDB) is 2.82x faster than SparkSQL and 1.8x faster than Ray.data. We argue that the performance advantage gained in IMLane primarily stems from two aspects. First, although SparkSQL and Ray.data also adopt process parallelism during execution, they require loading data from the database with additional data transfer overhead, not to mention using socket-based IPC, especially for bulk data inference Q1 and Q2. Second, the scheduling of AI functions in SparkSQL and Ray.data does not fully match the resource requirements in various user scenarios either, especially for heterogeneous inference Q5 and Q6. In general, IMLane delivers AI function execution performance on par with SparkSQL and Ray.data by addressing common bottlenecks in parallelization and scheduling, while also avoiding moving data out of the database.

6.4.4 Efficiency on LLM-based AI Functions. We further discuss IMLane on LLM-based AI functions using Q7. The performance of (IMLane, sched) is close to IMBridge and does not substantially outperform (IMLane, exec), with only about 1.05x improvement on DuckDB. This is because LLM inference is particularly compute-intensive and typically applies continuous batching optimizations. Using the default batch size or IMBridge-controlled batching already saturates GPU resources in our current hardware setting, leaving little room for scheduling gains including computation overlap. Nevertheless, if compute resources are sufficiently abundant, our scheduling may still play a role in improving performance.

7 RELATED WORK

Alternative AI Function Execution Systems. Commercial products like BigQuery [26], Snowflake [27], Apache Doris [2], and Databricks [25] provide built-in AI function execution within their engines, with tightly coupled runtimes and limited flexibility for other databases. In addition to these commercial systems, there are alternative open-source systems for executing Python AI functions on data stored in databases. Pandas pulls the entire table from the database into Python and applies the AI function in memory, an approach widely used by data agents like ReAcTable [102]. SparkSQL [62] executes AI functions as Python UDFs on Spark, leveraging data parallelism. Ray.data [75] builds on top of Ray [74] to provide scalable data processing based on task parallelism. These systems represent a spectrum from single-threaded execution to large-scale parallel execution. However, they execute AI functions outside the database engine, incurring extra overhead of data movement and lacking resource-aware scheduling. In contrast, IMLane is designed as a composable and extensible framework that can be integrated into different database engines with only minor modifications to their core code, and specifically addresses bottlenecks including ineffective parallel execution and resource-mismatched scheduling.
Optimizations of AI Function Execution. Prior work has explored various optimizations for AI function execution in databases. Raven [52] co-optimizes AI functions with relational operators via model pruning and inference runtime selection. Smart [23] generates SQL predicates from model structures to filter tuples before inference. LingoDB [29] and SDQL [82] use compilation to jointly optimize operators and AI functions. EvaDB [94] caches AI function results across queries. Aero [31] automates AI function scaling and predicate optimization. MASQ [50], Craftsman [51], and SmartLite [39] translate AI functions to SQL for in-database inference. Containerized UDF execution [78] eases AI function deployment. LEADS [99] optimizes with memory sharing and state caching. PEPS [58] applies model graph optimization and heuristic rewriting. IMLane focuses on parallelizing and scheduling AI functions, which is orthogonal and complementary to existing efforts.
Composable Database System. There is a growing shift toward disassembling monolithic database systems into composable and reusable components. ZetaSQL [20], CoreSQL [7], and libpgquery [38] provide unified SQL parsers and analyzers. Calcite [5], Orca [83], and Substrait [85] offer modular query planners and optimizers. Velox [55, 76], DataFusion [34], and BOSS [42] deliver unified execution engines with extensible components. Vineyard [98] and XDBC [19] introduce pluggable frameworks for efficient data transfer. Existing databases, e.g., DuckDB and PostgreSQL, also provide extension mechanisms [32] to facilitate adding new features into the database system. Following this trend, IMLane is a composable framework for efficient AI function execution in database engines.

8 CONCLUSION

In this work, we present IMLane, a composable development framework designed to empower database engines with efficient AI function execution. In particular, IMLane introduces process-level AI function execution and decoupled AI function scheduling to achieve effective parallel execution and resource-matched scheduling, respectively. Our integration of IMLane into database engines including OceanBase and DuckDB yields significant performance improvements and consistently outperforms alternative solutions.

ACKNOWLEDGMENTS

This work was supported by the National Natural Science Foundation of China (No. 62272168), the ECNU-OceanBase Joint Research Lab, and the ECNU Academic Innovation Promotion Program for Excellent Doctoral Students (No. YBNLTS2026014).

REFERENCES

[1] Apache Arrow. 2026. The universal columnar format and multi-language toolbox for fast data interchange and in-memory analytics. https://arrow.apache.org/.

[2] Apache Doris. 2026. Apache Doris LLM Functions. https://doris.apache.org/docs/4.0/ai/llm-functions.

[3] AWS SageMaker. 2026. Deploy models for real-time inference. https://docs.amazonaws.cn/en_us/sagemaker/latest/dg/realtime-endpoints-deploy-models.html.

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

[5] Edmon Begoli, Jesús Camacho-Rodríguez, Julian Hyde, Michael J. Mior, and Daniel Lemire. 2018. Apache Calcite: A Foundational Framework for Optimized Query Processing Over Heterogeneous Data Sources. In *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. 221–230.

[6] 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.

[7] Biswapesh Chattopadhyay, Pedro Pedreira, Sameer Agarwal, Yutian Sun, Suketu Vakharia, Peng Li, Weiran Liu, and Sundaram Narayanan. 2023. Shared Foundations: Modernizing Meta's Data Lakehouse. In *13th Conference on Innovative Data Systems Research (CIDR)*.

[8] Zhoujun Cheng, Tianbao Xie, Peng Shi, Chengzu Li, Rahul Nadkarni, Yushi Hu, Caiming Xiong, Dragomir Radev, Mari Ostendorf, Luke Zettlemoyer, Noah A. Smith, and Tao Yu. 2023. Binding Language Models in Symbolic Languages. In *The Eleventh International Conference on Learning Representations (ICLR)*.

[9] ClickHouse. 2026. User Defined Function (UDF). https://clickhouse.com/docs/sql-reference/functions/udf.

[10] cloudpipe. 2026. Extended pickling support for Python objects. https://github.com/cloudpipe/cloudpickle.

[11] Dask. 2026. Dask. https://docs.dask.org.cn/en/stable/.

[12] Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Zoi Kaoudi, Tilmann Rabl, and Volker Markl. 2022. Materialization and Reuse Optimizations for Production Data Science Pipelines. In *Proceedings of the 2022 International Conference on Management of Data (SIGMOD)*. 1962–1976.

[13] DuckDB. 2026. DuckDB Documentation. https://duckdb.org/docs/stable/.

[14] DuckDB. 2026. From Waddle to Flying: Quickly expanding DuckDB's functionality with Scalar Python UDFs. https://duckdb.org/2023/07/07/python-udf.html.

[15] DuckDB. 2026. Tuning Worloads. https://duckdb.org/docs/stable/guides/performance/how_to_tune_workloads.html.

[16] Eric Snow. 2026. PEP 684 – A Per-Interpreter GIL. https://peps.python.org/pep-0684/.

[17] Yannis Foufoulas, Theoni Palaiologou, and Alkis Simitsis. 2025. The UDFBENCH Benchmark for General-Purpose UDF Queries. *Proc. VLDB Endow. (PVLDB)* 18, 9 (2025), 2804–2817.

[18] 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.

[19] Haralampos Gavriilidis, Kaustubh Beedkar, Matthias Boehm, and Volker Markl. 2025. Fast and Scalable Data Transfer Across Data Systems. *Proc. ACM Manag. Data (PACMMOD)* 3, 3, Article 157 (2025).

[20] Google. 2026. ZetaSQL - Analyzer Framework for SQL. https://github.com/google/zetasql.

[21] GraalPython. 2026. A high-performance embeddable Python 3 runtime for Java. https://www.graalvm.org/python/.

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

[23] Yunyan Guo, Guoliang Li, Ruilin Hu, and Yong Wang. 2025. In-database query optimization on SQL with ML predicates. *VLDB J.* 34, 1 (2025).

[24] Chuxuan Hu, Austin Peters, and Daniel Kang. 2024. LEAP: LLM-Powered End-to-End Automatic Library for Processing Social Science Queries on Unstructured Data. *Proc. VLDB Endow. (PVLDB)* 18, 2 (2024), 253–264.

[25] Inc. Databricks. 2026. Databricks MLflow and AI Functions. https://docs.databricks.com/aws/en/machine-learning.

[26] Inc. Google. 2026. Big Query ML. https://cloud.google.com/bigquery/docs/bqml-introduction.

[27] Inc. Snowflake. 2026. Snowflake Cortex AI Functions. https://docs.snowflake.com/en/user-guide/snowflake-cortex/aisql.

[28] Zhipeng Jia and Emmett Witchel. 2021. Nightcore: efficient and scalable serverless computing for latency-sensitive, interactive microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 152–166.

[29] 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.

[30] Jython. 2026. Jython. https://www.jython.org/.

[31] Gaurav Tarlok Kakkar, Jiashen Cao, Aubhro Sengupta, Joy Arulraj, and Hyesoon Kim. 2025. Aero: Adaptive Query Processing of ML Queries. *Proc. ACM Manag. Data (PACMMOD)* 3, 3, Article 174 (2025).

[32] Abigale Kim, Marco Slot, David G. Andersen, and Andrew Pavlo. 2025. Anarchy in the Database: A Survey and Evaluation of Database Management System Extensibility. *Proc. VLDB Endow. (PVLDB)* 18, 6 (2025), 1962–1976.

[33] 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)*.

[34] Andrew Lamb, Yijie Shen, Daniël Heres, Jayjeet Chakraborty, Mehmet Ozan Kabak, Liang-Chi Hsieh, and Chao Sun. 2024. Apache Arrow DataFusion: A Fast, Embeddable, Modular Analytic Query Engine. In *Companion of the 2024 International Conference on Management of Data (SIGMOD)*. 5–17.

[35] Jiale Lao, Andreas Zimmerer, Olga Ovcharenko, Tianji Cong, Matthew Russo, Gerardo Vitagliano, Michael Cochez, Fatma Özcan, Gautam Gupta, Thibaud Hottelier, H. V. Jagadish, Kris Kissel, Sebastian Schelter, Andreas Kipf, and Immanuel Trummer. 2026. SemBench: A Benchmark for Semantic Query Processing Engines. arXiv:2511.01716 [cs.DB]

[36] 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.

[37] Edo Liberty, Zohar Karnin, Bing Xiang, Laurence Rouesnel, Baris Coskun, Ramesh Nallapati, Julio Delgado, Amir Sadoughi, Yury Astashonok, Piali Das, Can Balioglu, Saswata Chakravarty, Madhav Jha, Philip Gautier, David Arpin, Tim Januschowski, Valentin Flunkert, Yuyang Wang, Jan Gasthaus, Lorenzo Stella, Syama Rangapuram, David Salinas, Sebastian Schelter, and Alex Smola. 2020. Elastic Machine Learning Algorithms in Amazon SageMaker. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 731–737.

[38] libpgquery. 2026. C library for accessing the PostgreSQL parser outside of the server. https://github.com/pganalyze/libpg_query.

[39] 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.

[40] Kuan Lu, Zhihui Yang, Sai Wu, Ruichen Xia, Dongxiang Zhang, and Gang Chen. 2025. Adda: Towards Efficient in-Database Feature Generation via LLM-based Agents. *Proc. ACM Manag. Data (PACMMOD)* 3, 3, Article 125 (2025).

[41] 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).

[42] Hubert Mohr-Daurat, Xuan Sun, and Holger Pirk. 2023. BOSS - An Architecture for Database Kernel Composition. *Proc. VLDB Endow. (PVLDB)* 17, 4 (2023), 877–890.

[43] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. 2018. Ray: a distributed framework for emerging AI applications. In *Proceedings of the 13th USENIX Conference on Operating Systems Design and Implementation (OSDI)*. 561–577.

[44] Thomas Neumann. 2011. Efficiently compiling efficient query plans for modern hardware. *Proc. VLDB Endow. (PVLDB)* 4, 9 (2011), 539–550.

[45] NVIDIA. 2026. NVIDIA Triton Inference Server. https://docs.nvidia.com/deeplearning/triton-inference-server/user-guide/docs/index.html.

[46] OceanBase. 2026. Introduction to parallel execution. https://en.oceanbase.com/docs/common-oceanbase-database-10000000001973808.

[47] OceanBase. 2026. SSTables. https://en.oceanbase.com/docs/common-oceanbase-database-10000000001973722.

[48] ONNX Runtime. 2026. Welcome to ONNX Runtime. https://onnxruntime.ai/docs/.

[49] OpenAI. 2026. Lightweight coding agent that runs in your terminal. https://github.com/openai/codex.

[50] 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.

[51] Qingfeng Pan, Jiahe Zhi, Chenyang Zhang, Chen Xu, Zhao Zhang, Anita Shao, Guanglei Bao, Qiu Cui, Xiaowei Chen, and Aoying Zhou. 2025. Machine Learning Inference Pipeline Execution Using Pure SQL Based on Operator Fusion. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 3397–3410.

[52] 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.

[53] Misun Park, Ketan Bhardwaj, and Ada Gavrilovska. 2023. Pocket: ML Serving from the Edge. In *Proceedings of the Eighteenth European Conference on Computer Systems (EuroSys)*. 46–62.

[54] Jignesh M. Patel, Harshad Deshmukh, Jianqiao Zhu, Navneet Potti, Zuyu Zhang, Marc Spehlmann, Hakan Memisoglu, and Saket Saurabh. 2018. Quickstep: a data platform based on the scaling-up approach. *Proc. VLDB Endow. (PVLDB)* 11, 6 (2018), 663–676.

[55] 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.

[56] Pedro Pedreira, Orri Erling, Konstantinos Karanasos, Scott Schneider, Wes McKinney, Satya R Valluri, Mohamed Zait, and Jacques Nadeau. 2023. The Composable Data Management System Manifesto. *Proc. VLDB Endow. (PVLDB)* 16, 10 (2023), 2679–2685.

[57] Pedro Pedreira, Deepak Majeti, and Orri Erling. 2024. Composable Data Management: An Execution Overview. *Proc. VLDB Endow. (PVLDB)* 17, 12 (2024), 4249–4252.

[58] Yuchen Peng, Zhongle Xie, Ke Chen, Gang Chen, and Lidan Shou. 2025. Towards Automatic and Efficient Prediction Query Processing in Analytical Database . In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 2253–2266.

[59] PostgreSQL. 2026. Parallel Query. https://www.postgresql.org/docs/current/parallel-query.html.

[60] PostgreSQL. 2026. PL/Python — Python Procedural Language. https://www.postgresql.org/docs/current/plpython.html.

[61] Py4J. 2026. Py4J - A Bridge between Python and Java. https://www.py4j.org/.

[62] PySpark. 2026. PySpark Overview. https://spark.apache.org/docs/latest/api/python/index.html.

[63] Python. 2026. Can subinterpreters import extension packages? https://discuss.python.org/t/can-subinterpreters-import-extension-packages/55770.

[64] Python. 2026. global interpreter lock. https://docs.python.org/3/glossary.html#term-global-interpreter-lock.

[65] Python. 2026. multiprocessing — Process-based parallelism. https://docs.python.org/zh-cn/3/library/multiprocessing.html.

[66] Python. 2026. PEP 684: A Per-Interpreter GIL Discussion. https://discuss.python.org/t/pep-684-a-per-interpreter-gil/19583.

[67] Python. 2026. pickle — Python object serialization. https://docs.python.org/3/library/pickle.html.

[68] Python. 2026. Python experimental support for free threading. https://docs.python.org/3/howto/free-threading-python.html.

[69] Python. 2026. Thread State and the Global Interpreter Lock. https://docs.python.org/3/c-api/init.html#thread-state-and-the-global-interpreter-lock.

[70] Yiming Qiao and Huanchen Zhang. 2025. Data Chunk Compaction in Vectorized Execution. *Proc. ACM Manag. Data (PACMMOD)* 3, 1, Article 26 (2025).

[71] Qwen Team, Alibaba Group. 2026. Qwen/Qwen3-1.7B. https://huggingface.co/Qwen/Qwen3-1.7B.

[72] 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.

[73] Ray. 2026. Ray Serve: Scalable and Programmable Serving. https://docs.ray.io/en/latest/serve/index.html.

[74] Ray. 2026. What's Ray Core? https://docs.ray.io/en/latest/ray-core/walkthrough.html.

[75] Ray Data. 2026. Ray Data: Scalable Datasets for ML. https://docs.ray.io/en/latest/data/data.html.

[76] Laith Sakka, Pedro Pedreira, Orri Erling, Masha Basmanova, Kevin Wilfong, Wei He, Xiaoxuan Meng, Krishna Pai, and Bikramjeet Vig. 2024. Simple (yet Efficient) Function Authoring for Vectorized Engines. *Proc. VLDB Endow. (PVLDB)* 17, 12 (2024), 4187–4199.

[77] Sam Gross. 2026. PEP 703 – Making the Global Interpreter Lock Optional in CPython. https://peps.python.org/pep-0703/.

[78] Karla Saur, Tara Mirmira, Konstantinos Karanasos, and Jesús Camacho-Rodríguez. 2022. Containerized execution of UDFs: an experimental evaluation. *Proc. VLDB Endow. (PVLDB)* 15, 11 (2022), 3158–3171.

[79] 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.

[80] seekdb. 2026. seekdb: The AI-Native Search Database. https://www.seekdb.ai/.

[81] 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.

[82] 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).

[83] Mohamed A. Soliman, Lyublena Antova, Venkatesh Raghavan, Amr El-Helw, Zhongxian Gu, Entong Shen, George C. Caragea, Carlos Garcia-Alvarado, Foyzur Rahman, Michalis Petropoulos, Florian Waas, Sivaramakrishnan Narayanan, Konstantinos Krikellas, and Rhonda Baldwin. 2014. Orca: a modular query optimizer architecture for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data (SIGMOD)*. 337–348.

[84] Spring AI Alibaba. 2026. Spring AI Alibaba DataAgent. https://github.com/spring-ai-alibaba/DataAgent.

[85] Substrait. 2026. Substrait: Cross-Language Serialization for Relational Algebra. https://substrait.io/.

[86] Zhaoyan Sun, Jiayi Wang, Xinyang Zhao, Jiachi Wang, and Guoliang Li. 2025. Data Agent: A Holistic Architecture for Orchestrating Data+AI Ecosystems. *CoRR* abs/2507.01599 (2025).

[87] D. Vandevoorde, N.M. Josuttis, and D. Gregor. 2017. *C++ Templates: The Complete Guide*. Pearson Education.

[88] vLLM Contributors. 2026. vLLM: Easy, Fast, and Cheap LLM Serving for Everyone. https://vllm.ai.

[89] Wang Wei, Shaofeng Li, Dong Tian, Meng Yan, and Zhu Haojin. 2025. From Function Calls to MCPs for Securing AI Agent Systems: Architecture, Challenges and Countermeasures. *ZTE Communications* 23, 3 (2025), 27.

[90] Doris Xin, Hui Miao, Aditya Parameswaran, and Neoklis Polyzotis. 2021. Production Machine Learning Pipelines: Empirical Analysis and Optimization Opportunities. In *Proceedings of the 2021 International Conference on Management of Data (SIGMOD)*. 2639–2652.

[91] Boyan Xu, Yuyuan Cai, Shaobin Shi, Zhifeng Hao, and Ruichu Cai. 2025. Chat2DB: Chatting to the Database with Interactive Agent Assisted Language Models. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 4620–4623.

[92] Quanqing Xu, Wei Sun, Chuanhui Yang, Jinlong Liu, Ziyun Wei, Fusheng Han, Liang Wang, and Xiaowei Zhai. 2025. OceanBase Unitization: Building the Next Generation of Online Map Applications. In *2025 IEEE 41st International Conference on Data Engineering (ICDE)*. 4183–4196.

[93] Quanqing Xu, Chuanhui Yang, Feizhi Cai, Li Feng, Jiannan Ji, Zhifeng Yang, Guoping Wang, and Yi Zhang. 2026. PowerRAG: Building Smarter and More Accurate RAG Systems with OceanBase's Multi-modal Retrieval. In *Companion of the International Conference on Management of Data (SIGMOD Companion '26)*. 142–145.

[94] 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.

[95] Yicun Yang, Zhaoguo Wang, Yu Xia, Zhuoran Wei, Haoran Ding, Ruzica Piskac, Haibo Chen, and Jinyang Li. 2025. Automated Validating and Fixing of Text-to-SQL Translation with Execution Consistency. *Proc. ACM Manag. Data (PACMMOD)* 3, 3, Article 134 (2025).

[96] 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.

[97] 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.

[98] Wenyuan Yu, Tao He, Lei Wang, Ke Meng, Ye Cao, Diwen Zhu, Sanhong Li, and Jingren Zhou. 2023. Vineyard: Optimizing Data Sharing in Data-Intensive Analytics. *Proc. ACM Manag. Data (PACMMOD)* 1, 2, Article 200 (2023).

[99] Lingze Zeng, Naili Xing, Shaofeng Cai, Gang Chen, Beng Chin Ooi, Jian Pei, and Yuncheng Wu. 2024. Powering In-Database Dynamic Model Slicing for Structured Data Analytics. *Proc. VLDB Endow. (PVLDB)* 17, 13 (2024), 4813–4826.

[100] 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)*.

[101] 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 (PACMMOD)* (2025).

[102] Yunjia Zhang, Jordan Henkel, Avrilia Floratou, Joyce Cahoon, Shaleen Deep, and Jignesh M. Patel. 2024. ReAcTable: Enhancing ReAct for Table Question Answering. *Proc. VLDB Endow. (PVLDB)* 17, 8 (2024), 1981–1994.

[103] Yiming Zhang and Koji Tsuda. 2026. NbBench: Benchmarking Language Models for Comprehensive Nanobody Tasks.

[104] Niu Zhi and Dong Luming. 2026. Enhancing Code Quality with LLM in Software Static Analysis. *ZTE COMMUNICATIONS* 24, 1 (2026).

[105] Wei Zhou, Xuanhe Zhou, Qikang He, Guoliang Li, Bingsheng He, Quanqing Xu, and Fan Wu. 2026. Automating Database-Native Function Code Synthesis with LLMs. *Proc. ACM Manag. Data (PACMMOD)* 4, 3, Article 141 (2026).

[106] Xuanhe Zhou, Cheng Chen, Kunyi Li, Bingsheng He, Mian Lu, Qiaosheng Liu, Wei Huang, Guoliang Li, Zhao Zheng, and Yuqiang Chen. 2023. FEBench: A Benchmark for Real-Time Relational Data Feature Extraction. *Proc. VLDB Endow. (PVLDB)* (2023), 3597–3609.

[107] Xuanhe Zhou, Zhaoyan Sun, and Guoliang Li. 2024. DB-GPT: Large Language Model Meets Database. *Data Science and Engineering* 9, 1 (2024), 102–111.