Decentralized Actor Scheduling and Reference-based Storage in Xorbits: a Native Scalable Data Science Engine

Weizheng Lu
Renmin University of China
luweizheng@ruc.edu.cn

Chao Hui
Shandong University
chaohui@mail.sdu.edu.cn

Yunhai Wang
Renmin University of China
cloudseawang@gmail.com

Feng Zhang
Renmin University of China
fengzhang@ruc.edu.cn

Yueguo Chen
Renmin University of China
chenyueguo@ruc.edu.cn

Bao Liu
Xorbits Inc.
liubao@xprobe.io

Chengjie Li
Xorbits Inc.
lichengjie@xprobe.io

Zhaoxin Wu
Xorbits Inc.
wuzhaoxin@xprobe.io

Xuye Qin
Xorbits Inc.
qinxuye@xprobe.io

ABSTRACT

Data science pipelines consist of data preprocessing and transformation, and a typical pipeline comprises a series of operators, such as DataFrame filtering and `groupby`. As practitioners seek tools to handle larger-scale data while maintaining APIs compatible with popular single-machine libraries (e.g., pandas), scaling such a pipeline requires efficient distribution of decomposed tasks across the cluster and fine-grained, key-level intermediate storage management, two challenges that existing systems have not effectively addressed. Motivated by the requirements of scaling diverse data science applications, we present the design and implementation of Xorbits, a native scalable data science engine built on our decentralized actor model, Xoscar. Our actor model can eliminate dependency on a global scheduler and enable fast actor task scheduling. We also provide reference-based distributed storage with unified access across heterogeneous memory resources. Our evaluation demonstrates that Xorbits achieves up to 3.22× speedup on 3 machine learning pipelines and 22 data analysis workloads compared to state-of-the-art solutions. Xorbits is available on PyPI with nearly 1k daily downloads and has been successfully deployed in production environments.

PVLDB Reference Format:
Weizheng Lu, Chao Hui, Yunhai Wang, Feng Zhang, Yueguo Chen, Bao Liu, Chengjie Li, Zhaoxin Wu, and Xuye Qin. Decentralized Actor Scheduling and Reference-based Storage in Xorbits: a Native Scalable Data Science Engine. PVLDB, 18(9): 2955 - 2963, 2025.
doi:10.14778/3746405.3746420

PVLDB Artifact Availability:
The source code, data, and/or other artifacts have been made available at https://github.com/xorbitsai/xorbits.

Yueguo Chen (BRAIN, RUC) is the corresponding author.

Proceedings of the VLDB Endowment, Vol. 18, No. 9 ISSN 2150-8097.
doi:10.14778/3746405.3746420

1 INTRODUCTION

Data science pipelines typically involve data loading, preprocessing, transformations, and analysis [33, 43]. When scaling these workloads, practitioners prefer toolkits with APIs similar to single-node pandas [21, 27] while handling larger datasets [34, 37]. One common implementation approach to scalable data science (SDS) is partitioning large datasets into chunks and building a computational task graph, where each graph node executes using single-node libraries like pandas [34, 35]. Although this paradigm sounds straightforward, scaling diverse data science operations while maintaining compatible APIs presents significant challenges. Consider a data science pipeline that reads a CSV file and performs `groupby` and `merge` operations. In a distributed environment, data of different keys is distributed across multiple workers. Suppose the `key1` data should flow from worker 1 to worker 2, then to worker 3 for other `merge` operations. Worker 1 should deallocate the original `key1` memory space when `key1` data is no longer needed. This pipeline highlights two critical requirements: 1) efficient scheduling tasks from the computational graph across workers and 2) fine-grained, key-level data management within the distributed environment.

SDS requires addressing two critical challenges: efficiently distributing task graphs across workers for concurrent execution and enabling fast key-level data access to both local and remote data for operations like `groupby`. Existing SDS systems fall short in addressing these challenges. 1) **Centralized Scheduling**: Systems such as Dask [35] and Modin [34] on Ray [30] rely on a centralized scheduler to coordinate all task executions. This scheduler processes task submissions and data movement requests sequentially, which can easily become a system bottleneck [3, 31]. 2) **Coarse-grained Data Management**: Ray, originally designed for reinforcement learning [24], offer users the object-level data API rather than key-level. This design is misaligned with data science operations like `groupby` that require shuffling specific keys between workers. Consequently, Modin on Ray demonstrates poor efficiency in large datasets [6, 10].

Rather than building upon existing distributed execution engines like Ray or Dask, we design Xorbits from the ground up as an SDS-native engine driven by data science workload requirements. Xorbits leverages dynamic tiling [25] to build balanced computational task graphs, efficiently schedules partitioned tasks to appropriate

computing nodes, and then executes using single-machine tools. This ground-up design allows Xorbits to directly address the fundamental limitations of existing systems through two key innovations. First, we build Xorbits SDS engine with the actor model [19] that we call Xoscar. Xoscar uses IP addresses to create or reference actors. This eliminates the need for a centralized global scheduler, enabling efficient actor creation and execution through direct addressing. For task scheduling, Xorbits employs a nearest common successor enhanced breadth-first search (BFS) algorithm to assign tasks from the computational graph to appropriate workers, aiming to preserve data locality. Second, we implement a reference-based distributed data management system. We design a unified, filesystem-like I/O interface, abstracting data management through file handles instead of direct data objects, providing unified access across diverse storage backends like main memory, GPUs, and disks. Xorbits is a production-ready toolkit and can act as a distributed and drop-in replacement for widely used libraries such as pandas, NumPy, and cuDF, among others.

We evaluate the performance of Xorbits and Xoscar with various types of workloads, including machine learning preprocessing pipelines [1, 4, 16], DataFrame data analysis [15], and actor task scheduling. Compared to other fastest solutions, Xorbits achieves a substantial 3.22× average speedup on 3 machine learning pipelines and 22 data analysis workloads. In task scheduling tests, Xoscar achieves a 8× speedup over Ray.

To summarize, this paper makes the following contributions:

- We design our Xorbits SDS-native engine to meet the requirements of data science workloads. It is built upon the decentralized Xoscar actor model.
- We present the design of efficient task scheduling with reference-based distributed storage, enabling data locality and fine-grained data management across heterogeneous memory resources.
- We evaluate Xorbits and Xoscar against existing systems like PySpark, Dask, Ray, and Modin on various benchmarks with different hardware platforms. We demonstrate significant speedups over these systems in machine learning pipelines, data analysis, and actor concurrent execution.

2 BACKGROUND AND MOTIVATION

2.1 Background & Related Works

Scalable Data Science. One common approach for scalable data science [26, 45] that maintains API compatibility is the divide-and-conquer strategy, which partitions large datasets into smaller chunks, builds computational task graphs, and processes each chunk independently using single-machine tools. Popular libraries such as Dask [35] and Modin [34] utilize this method. While this approach sounds straightforward, developing a production-ready SDS system should address two fundamental challenges: 1) **Computational Graph**: building load-balanced computational task graphs to prevent data skewing and 2) **Distributed Execution Engine**: scheduling and executing tasks while efficiently managing intermediate data of computational graphs and preserving data locality. To address the first challenge, state-of-the-art solutions collect accurate runtime statistics to dynamically partition and re-optimize the computational graph, resulting in more balanced chunks [25, 41, 42]. This paper focuses on the latter challenge, where implementation complexity stems from the diverse range of data science operations, from embarrassingly parallel operators (like filtering DataFrame columns) to shuffle-intensive operations (like `groupby`).

Distributed Programming Model. In distributed computing, three main programming models exist: bulk synchronous parallel (BSP) [38], asynchronous many-task (AMT) [13, 18, 22], and actor [11, 19]. MPI [39], widely used in high-performance computing, implements the BSP model but suffers from inefficiencies due to synchronization barriers, especially with imbalanced workloads and heterogeneous computing environments [14, 46]. The AMT model eliminates strict synchronization barriers, but requires efficient task scheduling and data management to prevent performance degradation from load imbalance [29, 32, 40]. In contrast, the actor model provides a distinct approach where each actor maintains internal state and communicates exclusively through asynchronous messages, differing from both BSP's explicit barriers and AMT's limited state management during execution. Task scheduling can be categorized into two types: centralized and decentralized. In centralized approaches, all decision-making is handled by a single scheduler, and task assignments are made sequentially. This serial nature becomes a bottleneck as task complexity increases [17, 40]. In contrast, decentralized approaches offer better scalability by allowing parallel decision-making based on partial information, thereby eliminating single points of failure [28].

Existing Systems. Table 1 compares existing systems along with their underlying distributed runtimes and programming models. Dask's dask.distributed module employs a centralized scheduler that coordinates workers and making all decisions, including task states management and task graph updates. The scheduler can easily become the bottleneck [3, 31]. Modin on Ray uses Ray as its distributed runtime, which supports both tasks and actors but faces centralized scheduling challenges. Ray maintains a global control service for metadata and resource allocation, requiring frequent metadata synchronization and creating overhead in large clusters [8, 12]. While Ray's distributed object store offers simple APIs (`ray.put` and `ray.get`) that abstract data management, it lacks fine-grained interfaces for users. Spark has developed its own execution engine. PySpark connects Python processes to the JVM-based runtime through language bindings, introducing serialization overhead with Python user-defined functions (UDFs) [2, 23], limiting its integration with the broader Python data science ecosystem. Speed and scalability are discussed in detail in Section 6. In terms of API compatibility, Modin offers the best coverage. Other systems may fail in less common long-tail scenarios. Dask and PySpark do not support certain column-wise data decomposition operators [34], whereas Xorbits does. Regarding fault tolerance, Spark is the most mature. Dask, Modin, and Xorbits employ simpler mechanisms: Dask supports basic recomputation, Modin relies on Ray's fault tolerance, and Xorbits leverages orchestration platforms such as Kubernetes.

2.2 Empirical Study & Observation

To validate the limitations of existing systems in scheduling and data management, we conduct an empirical study. We use four computing workers, each with 64GB of memory, to perform the `groupby.agg` operation on a 22GB Parquet dataset. The detailed

Table 1: Comparison of SDS Engines. H: High, M: Medium, L: Low.

	Dask	Modin Ray	PySpark	Xorbits
Speed	M	M	H	H
Scalability	M	L	H	H
Python Native	✓	✓		✓
API Compatibility	L	H	L	M
Fault Tolerance	M	M	H	M
GPU	✓		plugin	✓
Distributed Engine	dask distributed	Ray	Spark Runtime	Xoscar
Programming Model	AMT	AMT Actor	AMT	Actor

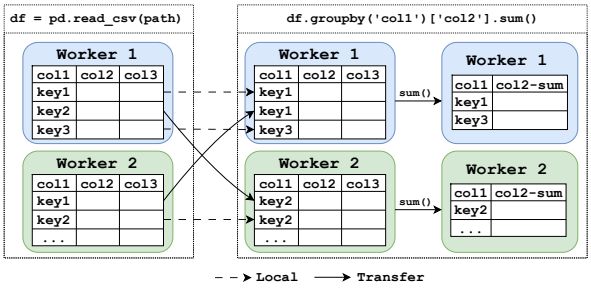


Figure 1: A simple data science pipeline in a distributed environment: loading data followed by a column-wise `groupby`. Note that this diagram provides a simplified illustration and does not show the complete MapReduce workflow.

experimental setup, including hardware specifications and existing system baselines, is detailed in Section 6.1. Figure 1 illustrates this shuffle-intensive DataFrame operation, where part of the data should be transferred while others remain local. In real-world scenarios, workloads exhibit much more complex data placement patterns and larger computational graphs than depicted in this figure. We monitor memory utilization, which is displayed in Figure 2, where each line represents the memory utilization of one worker. We plot memory instead of CPU usage, as memory is often the primary bottleneck, and memory patterns offer valuable insights into scheduling efficiency and data distribution. In this experiment, Xorbits is slightly slower than PySpark but 3.65× times faster than Dask, while Modin on Ray hangs and cannot complete it successfully. Through the empirical study and analysis of these systems' architecture, we reveal the following limitations:

- **Inefficient Task Scheduling**. Section 2.1 discusses the bottlenecks of centralized scheduling in Dask and Modin. The results of the empirical study also confirm this. In Figure 2, the memory usage of the four workers reveals the scheduling patterns. The four memory lines of Xorits overlap, indicating parallel execution. In contrast, Dask exhibits significant idle time between workers, referred to as *bubble* overhead [20], indicating inefficient task scheduling across multiple workers.
- **Coarse-grained Storage Management**. Ray's distributed object store is well-suited for learning tasks; however, it is not optimal for DataFrame scenarios. Take the `groupby` in Figure 1 as an example. A Ray worker will first put data locally. If the data needs to be transferred to other workers, Ray creates redundant copies of the data object on multiple workers because it does not give users fine-grained control over data transfer or garbage collection of unused data. This limitation is evident in our empirical results in Figure 2, where Modin on Ray exhibits severe memory imbalance: one worker consumes nearly 100% memory while others remain underutilized, data skew towards a single worker, and the workload cannot execute successfully. This issue has been repeatedly raised in the Modin community due to poor memory spill management [6, 10].

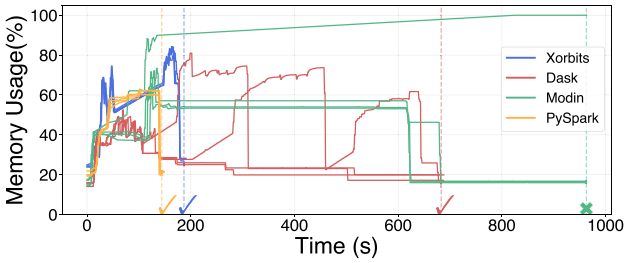


Figure 2: Memory usage during distributed DataFrame `groupby` on four workers using Xorbits, Dask, PySpark, and Modin on Ray. Each line represents one worker's memory usage over time.

3 SYSTEM OVERVIEW

3.1 Design Principle

Driven by the requirements of SDS, we believe that designing an SDS system should follow the following principles.

- **Lightweight and Flexible Task Scheduling**. To avoid the global scheduler becoming a system bottleneck, the scheduler and task dispatching should be lightweight and flexible.
- **Locality-aware and Fine-grained Control of Intermediate Data**. To optimize I/O-intensive operations, intermediate data management requires locality-aware, fine-grained control over data placement, movement, reference, and de-reference.

To this end, we develop **Xorbits, a toolkit that offers competitive performance while keeping APIs compatible with original local libraries.**

3.2 System Design

Figure 3 illustrates Xorbits' two-tier architecture. The underlying Xoscar is a decentralized, lightweight actor model without a global scheduler, allowing for the concurrent execution of computational tasks. Built upon Xoscar, we develop Xorbits, an SDS engine that can efficiently scale data science workloads, including DataFrame processing, array operations, etc. Since the actor layer handles both computation and communication, it should not be tightly bound to a global scheduler. While distributed systems often rely on a central scheduler, it can easily become a bottleneck. To mitigate this, we embed a lightweight scheduler within the SDS engine, minimizing centralized control. The engine defines two roles: a supervisor and

multiple workers. The supervisor delegates responsibilities like memory and task graph management to workers. Workers execute physical plans and manage local storage, with actors operating autonomously and without frequent synchronization.

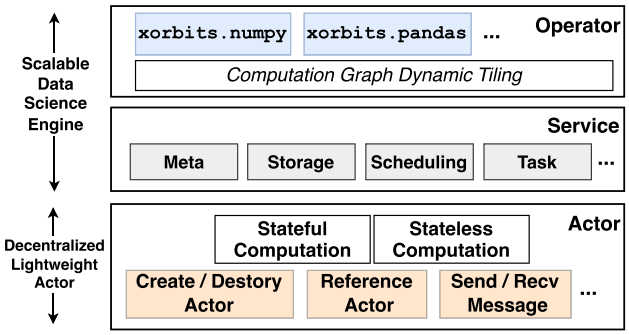


Figure 3: Xorbits' system architecture: divided into the decentralized actor layer and the scalable data science engine layer.

Our system introduces two key innovations:

- **IP-based Decentralized Actor** (Section 4) that alleviates the bottleneck of centralized scheduling. Xorbits supports direct task scheduling to specific computing nodes using IP-based addressing, allowing actors on different nodes to execute asynchronously and concurrently, and operations on worker actors do not need to inform the central supervisor.
- **Reference-based Distributed Storage** (Section 5) that enables fine-grained, key-level data access across both local and remote workers. When data is not in use, it is managed as a lightweight reference and only accessed or transferred when needed. First, this enables full-lifecycle, key-level data management with no redundant data copies. Second, it supports heterogeneous computing by allowing SDS workloads to scale on various storage media such as GPUs and disks.

4 LIGHTWEIGHT ACTOR SCHEDULING

In this section, we first introduce Xoscar, our actor model for task scheduling and execution, including its usage patterns and design principles. We then describe how physical execution plans are distributed across workers in the Xorbits SDS engine.

4.1 Xoscar Usage & Implementation

Before delving into the internal design, we will first explain how to use Xoscar, ensuring the reader can grasp our design philosophy.

Xoscar Usage. Listing 1 showcases the API usage. Before creating and using actors, users must start an actor pool which binds to a specific IP address and Xoscar generates a unique address (`external_address`) in the format of *ip:port*. To define an actor, users can inherit from `xoscar.Actor` and customizes their own methods (e.g., `estimate`). Xoscar supports both stateless and stateful computation. When actor users inherit from the `xoscar.Actor` class, they can create class instance attributes as needed (such as NumPy arrays or pandas DataFrames) to enable stateful computation. When creating and referencing an actor, users locate the actor by using the `external_address`.

Decentralized Actor. The example in Listing 1 illustrates that Xoscar 1) does not emphasize the concept of computing resources and 2) utilizes addresses for managing actor pools and actors. In a Xoscar cluster, there is no centralized scheduler. Centralized schedulers can be developed by higher-level applications (e.g., the SDS engine) tailored to specific workloads (e.g., scalable DataFrame). This approach eliminates the need for querying a global control service, as done in Ray. Consequently, Xoscar can rapidly create, reference, and use actors.

Listing 1: Quickstart example of Xoscar.

```
import xoscar as xo

pool = await xo.create_actor_pool(
    address="192.168.1.1", n_process=4
    )
class PiActor(xo.Actor):
    def estimate(self, n):
        # code to estimate pi
actor_ref = await xo.create_actor(
        PiActor, address=pool.external_address, ...
    )
actor_ref.estimate(100)
```

Lightweight Actor. Xoscar implements core functionalities of actor programming, such as creating, referencing, and message passing. Xoscar adheres to the principle of "users manage what they use", delegating fine-grained management tasks, such as resource management and task scheduling, to actor users (i.e., the SDS engine). Figure 4 illustrates the architecture of Xoscar, with two Xoscar workers on two computing nodes.

Actor Pool. In our implementation, Xoscar initiates a `MainActorPool` on each Worker node, which subsequently creates `SubActorPools` based on the user's required degree of parallelism. The `MainActorPool` is responsible for managing and coordinating all `SubActorPools`, while the `SubActorPools` are where actors are executed. Each actor pool operates as a separate process capable of hosting multiple actors, meaning multiple actors can exist within a single process. Applications built on Xoscar are responsible for managing parallelism. For instance, based on the application requirements, the SDS engine allocates one SubActorPool per CPU core or GPU device.

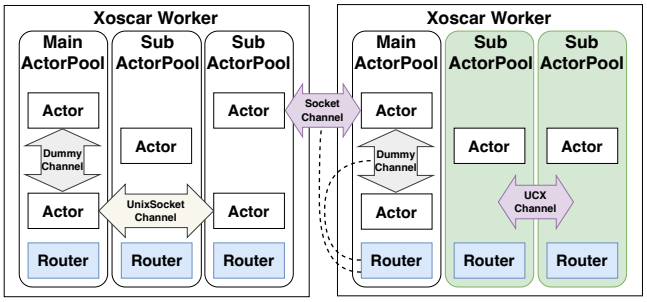


Figure 4: Xoscar architecture.

Communication. To facilitate message passing between actors, we offer various communication channels. These channels establish

connections between two actor pools, and each pool contains a router that maintains routing paths between source and destination.

- **DummyChannel**. When two actors reside within the same actor pool, meaning inside a single process, we define the DummyChannel. Two actors communicate with each other within the same process by putting or getting messages from the `asyncio.Queue`, following the producer-consumer pattern.
- **UnixSocketChannel**. When two actors are in different actor pools of the same worker, i.e., in different processes within the same worker, we design the UnixSocketChannel. The UnixSocketChannel leverages Unix domain sockets, which provide efficient inter-process communication by reading from and writing to shared files, thus enabling direct data exchange without the overhead of network protocols.
- **SocketChannel**. For scenarios where two actors are on different workers, thus requiring inter-node communication, we develop the SocketChannel, which uses TCP sockets for message passing.
- **UCXChannel**. UCXChannel facilitates UCX's [36] Remote Direct Memory Access (RDMA) for GPU-to-GPU communication.

4.2 Data-Intensive I/O

To optimize data-intensive I/O, we implemented two enhancements: point-to-point transfers and "delay & batch" remote actor calls.

Point-to-point Transfer. Shuffle-intensive I/O (e.g., DataFrame `groupby` or `merge`) mainly involves point-to-point (P2P) data exchanges between actors. We implement a buffer-to-buffer `copy_to` interface that utilizes the UCXChannel for RDMA data transfer between actors.

Delay & Batch. In our practice, we observe that, due to the non-uniform data distribution in SDS workloads, many remote actor calls contain only small amounts of data. And one I/O operation per remote actor call is quite costly. Consequently, we develop a "delay & batch" mechanism that initially delays these I/O-based actor calls; they are logged by Xoscar and then batch executed after accumulating several such calls.

4.3 Locality-Aware Task Scheduling

Task scheduling maps tasks of the physical execution plan to available computing resources. Xorbits builds task scheduling service through Xoscar's address-based allocation mechanism. We use "band" as the basic unit of hardware resource, where a band represents either a NUMA (Non-Uniform Memory Access) node or a GPU device. Xorbits' scheduling service identifies available bands on each Xorbits worker and assigns unique addresses to each band. Band address follows the format of "ip:numa-0" or "ip:gpu-0", aligning with Xoscar's address-based design principle.

Locality-Aware Scheduling Algorithm. To ensure upstream and downstream nodes in the task graph are assigned to the same or nearby bands, we design a nearest common successor enhanced breadth-first search (BFS) algorithm, which is outlined in Algorithm 1. This algorithm selects tasks with the nearest common successors from the task graph and assigns these tasks to the same or nearby workers. This algorithm leverages data locality between connected nodes as there are dataflow connections between these connected nodes.

Algorithm 1 Nearest Common Successor Enhanced Breadth-First Search for Task Assignment

Require: Graph $G = (N, E)$, N: task graph nodes, E: edges
$SQ \leftarrow$ starting nodes without predecessors in G
$first \leftarrow SQ[0]$, $remaining \leftarrow SQ[1:]$
Sort $remaining$ by NCS_Height with $first$ (ascending)
$SQ \leftarrow [first] + remaining$
$visited[n] \leftarrow false$ for all $n \in N$, $visited[SQ[0]] \leftarrow true$
$Q \leftarrow$ empty queue
ENQUEUE(Q, DEQUEUE(SQ))
while Q is not empty **do**
    $u \leftarrow$ DEQUEUE(Q), **output** u
    for each successor v of u **do**
        if $visited[v] = false$ **then**
            $visited[v] \leftarrow true$, ENQUEUE($Q$, v)
    if Q is empty & SQ is not empty **then**
        ENQUEUE(Q, DEQUEUE(SQ))

function NCS_HEIGHT($n1$, $n2$)
    Return the height of the NCS of $n1$ and $n2$

5 REFERENCE-BASED DISTRIBUTED STORAGE

In this section, we show the storage service for Xorbits based on Xoscar. As illustrated in Figure 5a, we decouple the intermediate storage service of Xorbits into two parts: the implementation of an SDS operator and the underlying distributed storage service.

5.1 Implementing SDS Operators

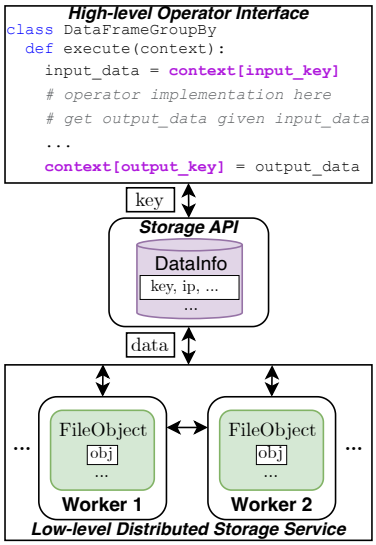
SDS comprises numerous APIs and operators, ranging from array to DataFrame. To facilitate the integration of various data science APIs and operators, we provide a `context` variable to access the metadata and data within the distributed environment.

Key-Value Context. Within the Xorbits' task graph, each graph node is assigned a unique key. Xorbits uses this key to reference the chunk data. Figure 5a provides a code snippet for the `groupby` operator. This code will be executed during runtime to compute the output of an operator based on its input data. If the predecessor's key is denoted as `input_key`, the output of the predecessor or the input of the current node can be retrieved by `context[input_key]`. With this key-value `context`, during operator development, developers can easily access the operator's intermediate data by referencing a key in a dictionary. This `context` acts like a proxy of distributed storage and abstracts away the complexity of underlying intermediate data management and physical memory operations.

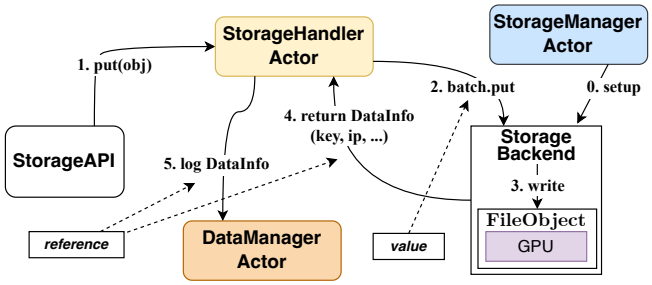
5.2 Reference-Based Distributed Storage

At the lower level, we overwrite the `context` dictionary, which is built upon our distributed storage service. The storage service is based on the Xoscar actor model and stores data across workers, providing data access service to the SDS operator through the storage API.

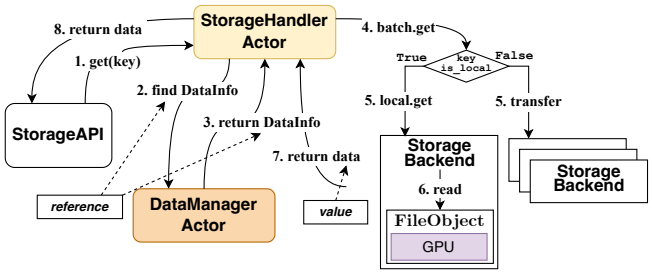
Unified File-Handle based Interface for Storage Management. To integrate heterogeneous storage resources, we design a unified file handle-based interface for storage management. Each



(a) Xorbits storage service: decoupling actual data from reference.



(b) Put data into the storage service.



(c) Get data from the storage service.

Figure 5: Xorbits storage: from operator implementation to distributed data access.

Xorbits worker creates and manages its own storage resource through `FileObject`, a filesystem-like interface that allows Xorbits to manipulate heterogeneous storage resources as if handling files. With the unified FileObject interface, we abstract various storage types as files and support pluggable backends, including shared memory, GPU, disk, and mmap [5]. Since traditional files are blocking, we introduce non-blocking I/O (e.g., asynchronous file writing or serialization) in Xoscar and Xorbits to support asynchronous access. To boost performance, we implement buffering for GPU and shared memory; for instance, the GPU backend offers filesystem-like interfaces leveraging NVIDIA's RMM [9] for memory management.

Decoupling of Actual Data from its Reference. Using the file handle-based storage management, we decouple physical data from its reference. Data is written to the worker's local storage backend, where each data chunk is assigned a unique key for SDS operator data access and referencing. We use the `DataInfo` data structure to manage data reference, and `DataInfo` is a metadata container that holds the data key, data size, offset, and the IP address of the worker where the data object is stored. Figure 5a illustrates the data-reference decoupling mechanism.

Fine-grained Intermediate Data Management. We provide fine-grained control over intermediate data management by implementing several services with actor programming. Figure 5b and Figure 5c demonstrate how to put and get a data object. After a user configures a specific storage backend, the `StorageManagerActor` is responsible for setting up, registering, and managing the backend. The `StorageAPI` sends requests to the `StorageHandlerActor`, which then delays and batches similar remote calls before putting or getting data into the storage backend. `DataInfo` is managed by the `DataManagerActor`, and subsequent I/O operations will use `DataInfo` for addressing, locating the worker first, and then getting data from the memory buffer based on the key and the offset. The get operation checks with `DataInfo` whether the data key is locally available, and if the data object resides on a remote node, fetches it through RDMA transfer. For garbage collection, the `DataManagerActor` keeps track of reference counts for all data objects and monitors their lifecycle within the distributed system. Once a data object is no longer referenced, the actor automatically schedules its deletion, thus efficiently reclaiming storage resources. Only the `DataManagerActor` is centralized, where workers query the `DataManagerActor` for `DataInfo` metadata to locate remote data chunks across the cluster.

6 EVALUATION

6.1 Experiment Setup

Benchmarks. We categorize our benchmark tests into four groups: machine learning (ML) pipelines, data analysis (DA), shuffle-intensive scalability tests, and actor micro-benchmarks. Table 2 summarizes the end-to-end workloads, including dataset sizes and computing resources (e.g., number of workers or GPUs). In these experiments, we use the pandas DataFrame API, for example, translating 22 TPC-H SQL queries into their DataFrame API equivalents.

Baselines. We compare Xorbits (0.8.2) and Xoscar (0.4.3) with Modin [34] (0.32.0) on Ray [30] (2.38.0), PySpark [44] (3.4.1), RAPIDS plugin for Spark [7] (24.04.1), Dask [35] (2024.8.2). Xorbits, Dask, and Modin leverage either pandas (2.2.3) or cuDF (24.8.1) as their underlying execution engine. Notably, all GPU-based PySpark executions use NVIDIA's RAPIDS GPU plugin, which intercepts Spark's

Table 2: Workloads to benchmark different systems.

Workload	Size	Format	Workers	Type
TPCx-AI [16] UC10 SF100	34GB	CSV	4	ML
NYC Taxi [1]	24GB	CSV	4	ML
Flight [4]	7GB	CSV	4	ML
TPC-H [15] SF10	4GB	Parquet	1	DA
TPC-H [15] SF100	36GB	Parquet	1 (8 GPUs)	DA
TPC-H [15] SF1000	358GB	Parquet	2 (16 GPUs)	DA

query execution plan to replace CPU operations with GPU equivalents. We use the default configuration without any tuning or performance optimization for all of these systems.

Hardware. We conduct experiments in two environments: a high-performance computing cluster and a personal laptop. These two environments are highly representative, as most data scientists start with exploratory data analysis locally and then scale up to clusters. The hardware specification is listed in Table 3.

Table 3: The hardware specification for experimental evaluation.

Env Setting	Device	CPU	Memory	GPU	# Nodes
Cluster	CPU	Intel E5-2650 v4 24 Cores	64GB	-	1-32
	GPU	Intel 8358 64 Cores	512GB	8 NVIDIA A800 80G SXM GPUs	1-2
Laptop	CPU	Apple M2	24GB	-	1

6.2 End-to-end Workload Performance

Machine Learning Pipelines. The machine learning pipeline tests are conducted on CPU workers, and the results are shown in Figure 6a. The NYC Taxi pipeline only consists of embarrassingly parallel operations (shuffle-free), where Xorbits outperformed the fastest Modin by 1.25×. On the Flight and TPCxAI UC10 workloads, which include `merge` and `groupby` operations, Xorbits surpasses PySpark with speedups of 1.69× and 1.08×, respectively.

Data Analysis on GPU Backend. The GPU experiments are to test how Xorbits utilizes GPU memory and are conducted on the NVIDIA A800 GPU nodes. Figure 6b displays the relative time of different frameworks running on GPUs with TPC-H SF100 and SF1000. The findings underscore Xorbits' superior GPU memory resource efficiency and robust handling of various query workloads. For the TPC-H SF100 on 8 A800 GPUs, Xorbits demonstrates substantial speedups over Dask across various queries, achieving a peak speedup of 10.56× (Q9) and an average speedup of 1.92× (considering only successful queries). Against PySpark with the RAPIDS plugin, Xorbits achieves a maximum speedup of 8.65× (Q12) and an average speedup of 3.37× on successful queries. Moreover, Xorbits successfully completes all queries, while both Dask and PySpark encounter failures, highlighting Xorbits' reliability. TPC-H SF1000 benchmarks are conducted on 2 nodes with a total of 16 A800 GPUs. Among successful queries, Xorbits achieves an average speedup of 1.56× and a peak speedup of 2.06× (Q2) over Dask. Compared to PySpark with the RAPIDS plugin, Xorbits achieves an average speedup of 5.67× for the two successful queries.

Data Analysis on Disk Backend. To assess Xorbits' ability to leverage disk storage, we run TPC-H SF100 with *mmap* storage backend on a 24GB memory MacBook laptop. Table 4 presents the number of successful queries out of the total 22 and the corresponding speedups achieved by Xorbits compared to other systems. The results reveal that other systems fail primarily due to inefficient memory-to-disk management. Specifically, pandas is an in-memory-only engine; Modin on Ray frequently encounters spilling-related deadlocks caused by its coarse-grained object store [6, 10]; and Dask and PySpark, despite supporting disk spilling by default, suffer performance bottlenecks on queries involving intensive I/O and frequent shuffles. Xorbits' success and performance advantages stem from its effective reference-based storage design.

Table 4: Successful TPC-H SF100 Query Executions and Xorbits Speedup on a 24GB MacBook.

SDS Engine	pandas	Dask	Modin on Ray	PySpark	Xorbits
# of Success	5/22	14/22	1/22	17/22	22/22
Speedup	4.32	1.26	6.72	4.32	/

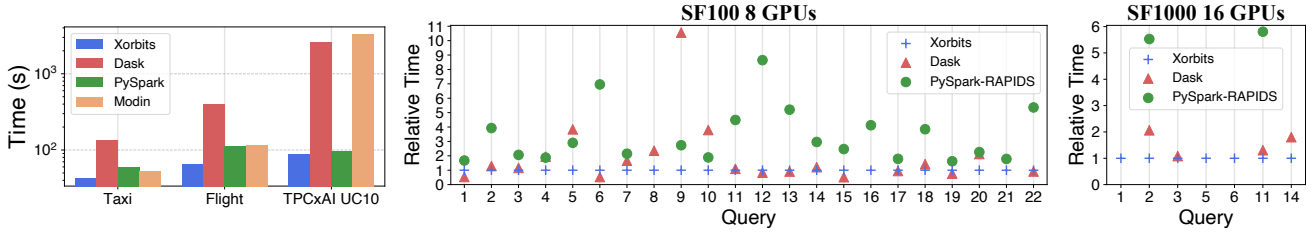
Compare with pandas. To evaluate any potential overhead introduced by the Xorbits engine running atop pandas, we conduct a comparative performance analysis between Xorbits and native pandas using TPC-H SF10 on the laptop. Xorbits completes the benchmark in 1394s compared to pandas' 1773s, achieving a 1.27× speedup. While Xorbits shows some overhead for lightweight queries Q2 and Q22 (which pandas completes in approximately 10 seconds), Xorbits demonstrates significant acceleration for complex operations, particularly Q9 which has five `merge` and one `groupby` and achieves a 3.59× speedup.

6.3 Scaling Data-Intensive Operations

To evaluate the ability of Xorbits' SDS engine to achieve high-performance data shuffling, we test DataFrame `groupby` and `merge`, two data-intensive operators, on both CPUs and GPUs. We synthesize these datasets with various cases, with each case featuring different amounts of shuffle data, such as the unique ratio for `groupby` and the match ratio for `merge`. Figure 8 presents the results. In CPU-based tests, Xorbits slightly underperforms PySpark in three out of four cases and shows notable advantages over Dask across all four. Modin completes only one task, constrained by memory management limitations. Xorbits shows a considerable advantage over both PySpark and Dask in GPU-based tests. Specifically, Xorbits achieves a maximum speedup of 1.57× and an average speedup of 1.30× compared to Dask. The relative speedup compared to PySpark is even greater, with a peak of 7.64× and an average of 5.34× (based on successful cases only).

6.4 Performance Impact Breakdown

To isolate and quantify the specific benefits of our two key innovations, we design three micro-benchmarks. The scheduling task



(a) ML pipeline

(b) TPC-H on GPUs

Figure 6: End-to-end workloads performance.

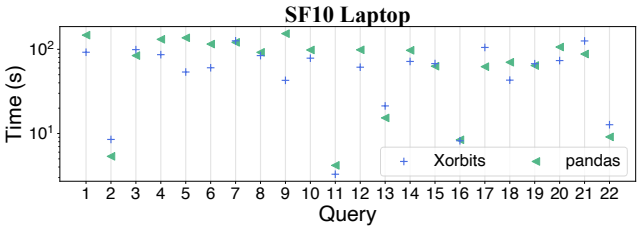


Figure 7: Compare Xorbits with underlying pandas to test potential overhead introduced by Xorbits.

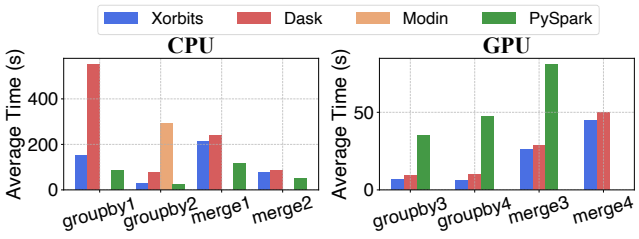


Figure 8: Experiment on two shuffle-intensive DataFrame operators: `groupby` and `merge` on CPU (Left) and GPU (Right).

evaluates the time needed to schedule an actor before execution. Estimating π demonstrates the efficiency of concurrent actor execution, and key-level storage experiment simulates a key-level data access scenario, where part of the data resides on local actors and the rest on remote actors. Figure 9 presents the results. Compared to "Estimate π," which represents a complete workload, the "Scheduling Overhead" benchmark isolates the scheduling time from actual task execution. By comparing the time saved in scheduling with the total execution time improvement, we estimate that our scheduling method contributes 40–50% to overall performance. The "Key-level Storage" benchmark shows approximately a 35% improvement.

6.5 Strength & Weakness

Xorbits outperforms Dask and Modin due to its more efficient scheduling and data management. While Spark SQL offers strong performance (Figure 2 and left graph of Figure 8), using Python UDFs and the overhead of Python-to-JVM translation in PySpark introduce extra overhead, leading to lower performance for the PySpark pandas API than Xorbits (Figure 6a). Additionally, Spark is memory-based

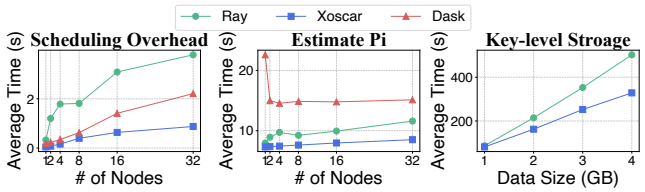


Figure 9: Micro-benchmarks: scheduling actors (Left), approximating π (Middle), and key-level data storage (Right).

and performs well when sufficient memory is available, but it may hang when memory is insufficient (Table 4).

6.6 Summary of Findings & Insights

Our work on Xorbits has revealed several insights about rethinking and building SDS systems from scratch. First, general-purpose distributed engines like Dask and Ray suffer from centralized scheduling or coarse-grained object management, limiting performance and scalability. Second, we observe a gap between API compatibility and efficient execution: while PySpark offers strong performance, it does not integrate seamlessly with the native Python ecosystem. Third, achieving SDS on distributed heterogeneous computing hardware remains challenging.

7 CONCLUSION

Xorbits efficiently scales data science workloads by leveraging heterogeneous memory resources through two key innovations: a decentralized actor model, Xoscar, that uses IP addresses for scheduling and referencing actors, and a distributed storage system that abstracts memory resources as files, manipulating handles rather than data objects. Experiments demonstrate that this architecture delivers performance advantages over competing SDS toolkits.

ACKNOWLEDGMENT

This work is supported by the grants of the National Key R&D Program of China under Grant 2022ZD0160805, NSFC (62132017, 62272466, U2436209 and U24A20233), the Shandong Provincial Natural Science Foundation (No.ZQ2022JQ32), the Beijing Natural Science Foundation (L247027), big data and responsible AI for national governance (BRAIN, RUC), the Public Computing Cloud of RUC, the Fundamental Research Funds for the Central Universities, and the Research Funds of Renmin University of China (24XNKJ22).

REFERENCES

[1] 2014. *2014 Yellow Taxi Trip Data.* https://catalog.data.gov/dataset/2014-yellow-taxi-trip-data Accessed: 2024-11-22.

[2] 2022. *PySpark vs Scala Spark vs Spark SQL - Which one is performance efficient? Are UDFs still bad?* https://community.databricks.com/t5/data-engineering/pyspark-udf-is-taking-long-to-process/td-p/7794 Accessed: 2025-03-15.

[3] 2024. *Actors.* https://distributed.dask.org/en/stable/actors.html Accessed: 2024-10-22.

[4] 2024. *Flight Status Prediction.* https://www.kaggle.com/datasets/robikscube/flight-delay-dataset-20182022/data Accessed: 2024-11-22.

[5] 2024. *mmap — Memory-mapped file support.* https://docs.python.org/3.12/library/mmap.html Accessed: 2024-10-26.

[6] 2024. *modin with ray engine hang.* https://github.com/modin-project/modin/issues/7349 Accessed: 2025-01-20.

[7] 2024. *RAPIDS Accelerator For Apache Spark.* https://github.com/NVIDIA/spark-rapids Accessed: 2024-11-22.

[8] 2024. *Ray v2 Architecture.* https://docs.google.com/document/d/1tBw9A4j62ruI5omIJbMxly-la5w4q_TjyJgJL_jN2fI Accessed: 2024-11-22.

[9] 2024. *RMM: RAPIDS Memory Manager.* https://github.com/rapidsai/rmm Accessed: 2024-10-28.

[10] 2024. *suggestions on handling out of memory matrix operation on large dataset.* https://github.com/modin-project/modin/issues/6677 Accessed: 2024-11-02.

[11] Gul Agha. 1986. *Actors: A Model of Concurrent Computation in Distributed Systems.* MIT Press, Cambridge, MA, USA.

[12] Yadu Babuji, Anna Woodard, Zhuozhao Li, Daniel S. Katz, Ben Clifford, Rohan Kumar, Lukasz Lacinski, Ryan Chard, Justin M. Wozniak, Ian Foster, Michael Wilde, and Kyle Chard. 2019. Parsl: Pervasive Parallel Programming in Python. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*. 25–36.

[13] Michael Bauer, Sean Treichler, Elliott Slaughter, and Alex Aiken. 2012. Legion: Expressing Locality and Independence with Logical Regions. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[14] Rob H. Bisseling. 2004. *Parallel Scientific Computation: A Structured Approach Using BSP and MPI.* Oxford University Press, Inc.

[15] Peter Boncz, Thomas Neumann, and Orri Erling. 2013. TPC-H Analyzed: Hidden Messages and Lessons Learned from an Influential Benchmark. In *Performance Characterization and Benchmarking*. 61–76.

[16] 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. *Proceedings of the VLDB Endowment* 16, 12 (2023), 3649–3661.

[17] Yakup Budanaz, Mario Wille, and Michael Bader. 2022. Asynchronous Workload Balancing through Persistent Work-Stealing and Offloading for a Distributed Actor Model Library. In *2022 IEEE/ACM Parallel Applications Workshop: Alternatives To MPI+X (PAW-ATM)*. 39–51.

[18] Minyang Han and Khuzaima Daudjee. 2015. Giraph Unchained: Barrierless Asynchronous Parallel Execution in Pregel-like Graph Processing Systems. *Proceedings of VLDB Endowment* 8, 9 (2015), 950–961.

[19] Carl Hewitt, Peter Boehler Bishop, and Richard Steiger. 1973. A Universal Modular ACTOR Formalism for Artificial Intelligence. In *Proceedings of the 3rd International Joint Conference on Artificial Intelligence*. 235–245.

[20] Yanping Huang, Youlong Cheng, Ankur Bapna, Orhan Firat, Dehao Chen, Mia Chen, HyoukJoong Lee, Jiquan Ngiam, Quoc V Le, Yonghui Wu, and zhifeng Chen. 2019. GPipe: Efficient Training of Giant Neural Networks Using Pipeline Parallelism. In *Advances in Neural Information Processing Systems*, Vol. 32. 103–112.

[21] Alekh Jindal, K. Venkatesh Emani, Maureen Daum, Olga Poppe, Brandon Haynes, Anna Pavlenko, Ayushi Gupta, Karthik Ramachandra, Carlo Curino, Andreas Müller, Wentao Wu, and Hiren Patel. 2021. Magpie: Python at Speed and Scale Using Cloud Backends. In *11th Conference on Innovative Data Systems Research*.

[22] Laxmikant V. Kale and Sanjeev Krishnan. 1993. CHARM++: A Portable Concurrent Object Oriented System Based on C++. In *Proceedings of the Eighth Annual Conference on Object-Oriented Programming Systems, Languages, and Applications*. 91–108.

[23] Lakshay Goel. 2023. *PySpark UDF is taking long to process.* https://community.databricks.com/t5/data-engineering/pyspark-udf-is-taking-long-to-process/td-p/7794 Accessed: 2025-03-15.

[24] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. 2018. RLlib: Abstractions for Distributed Reinforcement Learning. In *Proceedings of the 35th International Conference on Machine Learning*, Vol. 80. 3053–3062.

[25] Weizheng Lu, Kaisheng He, Xuye Qin, Chengjie Li, Zhong Wang, Tao Yuan, Xia Liao, Feng Zhang, Yueguo Chen, and Xiaoyong Du. 2024. Xorbits: Automating Operator Tiling for Distributed Data Science. In *2024 IEEE 40th International Conference on Data Engineering*. 5211–5223.

[26] Weizheng Lu, Jing Zhang, Ju Fan, Zihao Fu, Yueguo Chen, and Xiaoyong Du. 2025. Large language model for table processing: a survey. *Frontiers of Computer Science* 19, 2 (2025), 192350.

[27] Wes McKinney. 2011. Pandas: A Foundational Python Library for Data Analysis and Statistics. *Python for high performance and scientific computing* 14, 9 (2011), 1–9.

[28] Harshitha Menon and Laxmikant Kalé. 2013. A Distributed Dynamic Load Balancer for Iterative Applications. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. 1–11.

[29] Omri Mor, George Bosilca, and Marc Snir. 2023. Improving the Scaling of an Asynchronous Many-Task Runtime with a Lightweight Communication Engine. In *Proceedings of the 52nd International Conference on Parallel Processing*. 153–162.

[30] 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*. 561–577.

[31] Angelo Mozzillo, Luca Zecchini, Luca Gagliardelli, Adeel Aslam, Sonia Bergamaschi, and Giovanni Simonini. 2025. Evaluation of Dataframe Libraries for Data Preparation on a Single Machine. In *Proceedings 28th International Conference on Extending Database Technology*. 337–349.

[32] Niranda Perera, Arup Kumar Sarker, Kaiying Shan, Alex Fetea, Supun Kamburugamuve, Thejaka Amila Kanewala, Chathura Widanage, Mills Staylor, Tianle Zhong, Vibhatha Abeykoon, Gregor Von Laszewski, and Geoffrey Fox. 2024. Supercharging Distributed Computing Environments for High-Performance Data Engineering. *Frontiers in High Performance Computing* 2 (2024), 1384619.

[33] Devin Petersohn, Stephen Macke, Doris Xin, William Ma, Doris Lee, Xiangxi Mo, Joseph E. Gonzalez, Joseph M. Hellerstein, Anthony D. Joseph, and Aditya Parameswaran. 2020. Towards Scalable Dataframe Systems. *Proceedings of the VLDB Endowment* 13, 12 (2020), 2033–2046.

[34] Devin Petersohn, Dixin Tang, Rehan Durrani, Areg Melik-Adamyan, Joseph E. Gonzalez, Anthony D. Joseph, and Aditya G. Parameswaran. 2021. Flexible Rule-Based Decomposition and Metadata Independence in Modin: A Parallel Dataframe System. *Proceedings of the VLDB Endowment* 15, 3 (2021), 739–751.

[35] Matthew Rocklin. 2015. Dask: Parallel Computation with Blocked Algorithms and Task Scheduling. In *Python in Science Conference*. 126–132.

[36] Pavel Shamis, Manjunath Gorentla Venkata, M. Graham Lopez, Matthew B. Baker, Oscar Hernandez, Yossi Itigin, Mike Dubman, Gilad Shainer, Richard L. Graham, Liran Liss, Yiftah Shahar, Sreeram Potluri, Davide Rossetti, Donald Becker, Duncan Poole, Christopher Lamb, Sameer Kumar, Craig Stunkel, George Bosilca, and Aurelien Bouteiller. 2015. UCX: An Open Source Framework for HPC Network Apis and Beyond. In *2015 IEEE 23rd Annual Symposium on High-Performance Interconnects*. 40–43.

[37] Phanwadee Sinthong and Michael J. Carey. 2021. PolyFrame: A Retargetable Query-Based Approach to Scaling Dataframes. *Proceedings of the VLDB Endowment* 14, 11 (2021), 2296–2304.

[38] Leslie G. Valiant. 1990. A Bridging Model for Parallel Computation. *Communications of The ACM* 33, 8 (1990), 103–111.

[39] David W Walker and Jack J Dongarra. 1996. MPI: A Standard Message Passing Interface. *Supercomputer* 12 (1996), 56–68.

[40] Ke Wang, Kan Qiao, Iman Sadooghi, Xiaobing Zhou, Tonglin Li, Michael Lang, and Ioan Raicu. 2016. Load-Balanced and Locality-Aware Scheduling for Data-Intensive Workloads at Extreme Scales. *Concurrency and Computation: Practice and Experience* 28, 1 (2016), 70–94.

[41] Reynold S. Xin, Josh Rosen, Matei Zaharia, Michael J. Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and Rich Analytics at Scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. 13–24.

[42] Maryann Xue, Steven Chen, Andy Lam, Yuanjian Li, Yingyi Bu, Herman Van Hovell, Yunxiao Ma, Xiao Li, Sameer Paranjpye, Abhishek Somani, Bart Samwel, Vuk Ercegovac, Sriram Krishnamurthy, Reynold Xin, Wenchen Fan, Mostafa Mokhtar, Jiexing Li, Amit Shukla, Matei Zaharia, Ziqi Liu, Rk Korlapati, Alexander Behm, and Michalis Petropoulos. 2024. Adaptive and Robust Query Execution for Lakehouses at Scale. *Proceedings of the VLDB Endowment* 17, 12 (2024), 3947–3959.

[43] Cong Yan and Yeye He. 2020. Auto-Suggest: Learning-to-Recommend Data Preparation Steps Using Data Science Notebooks. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*. 1539–1554.

[44] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: A Unified Engine for Big Data Processing. *Communications of The ACM* 59, 11 (2016), 56–65.

[45] Feng Zhang, Chenyang Zhang, Jiawei Guan, Qiangjun Zhou, Kuangyu Chen, Xiao Zhang, Bingsheng He, Jidong Zhai, and Xiaoyong Du. 2025. Breaking the Edge: Enabling Efficient Neural Network Inference on Integrated Edge Devices. *IEEE Transactions on Cloud Computing* 13, 02 (2025), 694–710.

[46] Xing Zhao, Manos Papagelis, Aijun An, Bao Xin Chen, Junfeng Liu, and Yonggang Hu. 2019. Elastic Bulk Synchronous Parallel Model for Distributed Deep Learning. In *2019 IEEE International Conference on Data Mining*. 1504–1509.