

# OASIS: Object-based Analytics Storage for Intelligent SQL Query Offloading in Scientific Tabular Workloads

Soon Hwang<sup>1</sup>, Junhyeok Park<sup>1</sup>, Junghyun Ryu<sup>1</sup>, Seonghoon Ahn<sup>1</sup>, Jeoungahn Park<sup>2</sup>, Jeongjin Lee<sup>2</sup>

Soonyeal Yang<sup>2</sup>, Jungki Noh<sup>2</sup>, Woosuk Chung<sup>2</sup>, Hoshik Kim<sup>2</sup>, and Youngjae Kim<sup>1</sup>

<sup>1</sup>Sogang University, Seoul, Republic of Korea

<sup>2</sup>Memory Systems Research, SK hynix Inc.

**Abstract**—Computation-Enabled Object Storage (COS) systems, such as MinIO and Ceph, have recently emerged as promising storage solutions for post hoc, SQL-based analysis on large-scale datasets in High-Performance Computing (HPC) environments. By supporting object-granular layouts, COS facilitates column-oriented access and supports in-storage execution of data reduction operators, such as filters, close to where the data resides. Despite growing interest and adoption, existing COS systems exhibit several fundamental limitations that hinder their effectiveness. First, they impose rigid constraints on output data formats, limiting flexibility and interoperability. Second, they support offloading for only a narrow set of operators and expressions, restricting their applicability to more complex analytical tasks. Third—and perhaps most critically—they fail to incorporate design strategies that enable compute offloading optimized for the characteristics of deep storage hierarchies. To address these challenges, this paper proposes OASIS, a novel COS system that features: (i) flexible and interoperable output delivery through diverse formats, including columnar layouts such as Arrow; (ii) broad support for complex operators (e.g., aggregate, sort) and array-aware expressions, including element-wise predicates over array structures; and (iii) dynamic selection of optimal execution paths across internal storage layers, guided by operator characteristics and data movement costs. We implemented a prototype of OASIS and integrated it into the Spark analytics framework. Through extensive evaluation using real-world scientific queries from HPC workflows, OASIS achieves up to a 32.7% performance improvement over Spark configured with existing COS-based storage systems.

## I. INTRODUCTION

In modern scientific research, vast amounts of data are generated through simulations and experiments in domains such as Computational Fluid Dynamics (CFD), High-Energy Physics (HEP), and Particle-In-Cell (PIC) simulations. These datasets are typically structured in tabular formats with well-defined schemas, where each record—such as a CFD cell or particle—contains a consistent set of fields [1]–[4]. This tabular structure aligns well with the relational model, enabling researchers to perform post hoc analysis using SQL-style queries (§II-A) [5]. To support scalable analysis of such large datasets, distributed data processing frameworks—most notably Apache Spark [6]—have been widely adopted on High-Performance Computing (HPC) systems for scientific data analytics [7], [8].

Meanwhile, recent advances in HPC simulations and scientific instrumentation have further accelerated data growth, placing increasing pressure on analytics systems to keep pace [9]. For example, supercomputers such as Frontier [10]

at Oak Ridge National Laboratory (ORNL) enable massive simulations that produce structured outputs across thousands of timesteps. Likewise, facilities like the High-Luminosity Large Hadron Collider (HL-LHC) [11] substantially increase data acquisition rates through enhanced detector resolution and higher event frequency. As data volumes continue to grow, their rate of increase is rapidly outpacing improvements in I/O and network bandwidth, leading to two key challenges:

First, data movement has become a major bottleneck in HPC analytics, exacerbated not only by increasing data volumes but also by low-selectivity queries that focus on narrow regions of interest (§II-B). While processing power has advanced for large-scale post-hoc analysis, storage I/O has not kept pace, causing transfer delays, idle compute nodes, and reduced system efficiency [12]–[15]. Second, traditional POSIX-based file systems struggle with efficient data placement at scale [16]. Their flat byte-stream model lacks structural awareness, making it difficult to selectively place hot columns on fast storage. This often results in hot data being placed on slow storage or in fast-tier overuse, degrading I/O performance [17]. These limitations highlight the need for storage systems that minimize data movement and support column-aware layout (§II-C).

To address these challenges, Computation-Enabled Object Storage (COS) systems—such as MinIO [18], Ceph [19] with S3 Select [20], and Ceph with SkyhookDM support [21]—have emerged as promising solutions that augment object storage with lightweight computation at the storage layer (§II-D) [21]–[25]. These systems can perform *filter* and *project* operations near the data, mitigating data movement bottlenecks in HPC environments. COS also improves data placement by supporting column-level granularity through its object abstraction, enabling selective tiering based on access frequency. This addresses the data placement limitations of POSIX systems and aligns data layout with workload behavior.

To enable compute offloading to storage, COS systems are designed following two distinct architectural approaches: executing query operations at the interface layer (e.g., Ceph S3 Select [23], MinIO Select [22]), or embedding computation within the internal layers of the storage stack (e.g., SkyhookDM [21]). While these systems differ in their offloading mechanisms, they share common limitations that hinder support for complex, high-throughput analytics. These limitations stem from architectural constraints and limited expressiveness in query execution, and are summarized as follows (§III-B).

Y. Kim is the corresponding author.

- Inflexible Output Format Limits Optimization and Integration:** Existing COS systems return offloaded query results in fixed output formats such as CSV and JSON, which lack structural metadata and require additional parsing, or Apache Arrow [26], which offers efficient columnar access but is not universally supported by all analytics engines.
- Limited Offloading Capability for Various Operators and Array Semantics:** Most existing COS systems support only *filter* and *project* operations with scalar conditions, lacking essential advanced operators such as *aggregate* and array-based expressions common in HPC queries (§III-A). As a result, for queries involving such unsupported operators or array-based conditions, COS systems still need to transfer entire files or row groups to the compute layer, increasing data movement and slowing analysis.
- Excessive Inter-Storage Data Movement Due to Fixed Execution Layer:** Current COS systems execute queries at a single fixed layer, such as the gateway node that interfaces with compute clients, and thus fail to minimize internal data transfers. This inflexible design prevents early-stage data reduction in lower layers of the storage stack, such as storage array nodes, thus limiting the benefits of offloading.

To overcome the aforementioned limitations, we propose OASIS, a new COS system designed for analytical workflows in HPC environments (§IV-A). OASIS is composed of a frontend node (OASIS-FE) as a object interface gateway and multiple storage arrays (OASIS-A), interconnected via high-speed NVMe-over-Fabrics (NVMe-oF) over RDMA (§IV-B).

OASIS implements the following key features: **First**, OASIS generates both intermediate and final results of offloaded query execution in the Arrow columnar format, within storage nodes and during transmission to compute nodes. This design reduces serialization overhead and enables efficient data exchange across system layers. Final outputs can also be serialized in CSV or JSON format for compatibility with legacy tools. **Second**, OASIS provides advanced operators such as aggregation, sorting, and array-level computations, including element-wise arithmetic and conditional evaluation. These operations are executed directly by the storage-layer Query Executor, which is implemented using DuckDB [27] due to its lightweight design, embeddability, and support for complex SQL semantics that align well with OASIS’s core requirements (§IV-E). **Third**, OASIS deploys Query Executors on both the OASIS-FE and the OASIS-A. It then employs a Local Optimizer that analyzes the offloaded query plan and applies the Storage-side Query Plan Offloading and Decomposition Algorithm (SODA) to partition the plan into subplans for distributed execution across the OASIS-FE and OASIS-A nodes (§IV-F).

Based on operator characteristics, SODA selects between two decomposition strategies (§IV-G): (1) Coefficient-Aware Decomposition (CAD) and (2) Structure-Aware Placement (SAP). CAD applies to queries with scalar comparisons or simple scalar computations, where data movement can be reasonably estimated using precomputed statistics. It identifies a partitioning point that minimizes total data movement based

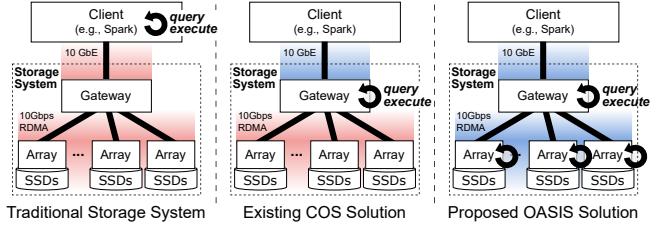


Fig. 1: Comparison of traditional storage systems, existing COS solutions, and the OASIS system for data analytics.

on estimated input-output ratios. In contrast, SAP targets queries for which data movement is difficult to estimate by statistics, such as those with array-level conditions or computations. SAP executes these operators close to the data and employs a lazy strategy. Specifically, it measures intermediate result sizes at runtime and forwards results to upper layers only when they fit within available internal bandwidth.

Figure 1 compares the storage architecture of OASIS with that of existing COS solutions. While existing COS systems reduce storage-to-compute data transfers by executing queries at the gateway layer within the storage system, they still incur substantial internal data movement across layers. In contrast, OASIS minimizes both inter-layer and storage-to-compute data transfers by initiating early-stage query processing in lower storage layers, guided by data movement cost.

We implemented a prototype of OASIS by building key components on SPDK v23.09 and DuckDB v1.3.0, where DuckDB serves as the in-storage Query Executor deployed across OASIS-FE and OASIS-A. We integrated OASIS with Apache Spark 4.0.0 to support query offloading. Our evaluation was conducted on a multi-node Spark cluster and an RDMA-connected OASIS setup, comprising 36–112 core servers with up to 386 GB of memory.

We evaluated OASIS using a diverse set of real-world analytical queries from representative workflows, including CFD and HEP domains. Evaluation results show that OASIS achieves up to 70.59% speedup over the traditional storage system and up to 32.7% speed up compared to existing COS-based setups. Furthermore, the SODA demonstrated its effectiveness in minimizing query execution time by optimally partitioning the query plan for complex queries composed of multiple operators (e.g., Q1 in Table IV).

In summary, our key contributions are as follows:

- We collect and analyze real-world tabular queries from HPC workflows in domains such as CFD, HEP, and PIC, with a focus on identifying operator patterns and structural traits that impact data movement and offloading opportunities.
- We design and implement OASIS, a novel object-based, computation-enabled analytical storage system that supports columnar formats, advanced operators with array-level expressions, and dataflow-aware query path optimization across hierarchical internal storage layers.
- We propose SODA, a query decomposition mechanism that partitions full offloaded query plans into subplans for hierarchical internal storage layers of OASIS-FE and OASIS-A.

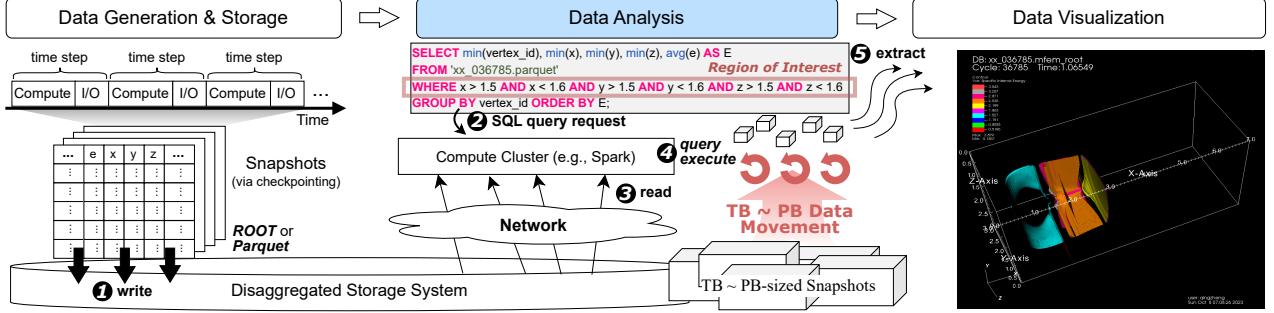


Fig. 2: A scientific analytics workflow, illustrating a post-hoc analysis pipeline (1 - 5) based on the disaggregated storage infrastructure. The example SQL query and visualization results are from the open-source Laghos 3D Mesh Dataset [28].

It applies CAD for scalar-centric workloads and SAP for array-centric workloads to enable efficient offloading.

## II. BACKGROUND AND RELATED WORK

### A. Post-Hoc Analysis Workflow in Scientific Applications

Figure 2 presents a representative post-hoc analysis workflow widely adopted in modern scientific applications, including those in Computational Fluid Dynamics (CFD), High-Energy Physics (HEP), and Particle-In-Cell (PIC) simulations. This workflow typically consists of four sequential stages: *data generation*, *data storage*, *data analysis*, and *data visualization*.

1) **Data Generation:** Modern scientific applications generate data by computing physical quantities (e.g., position, velocity, and energy) of individual particles within defined simulation domains, such as observation regions or grid cells. In observation-driven applications, such as particle collision experiments, data is collected in real-time through specialized detectors and instrumentation systems.

2) **Data Storage:** The generated data is periodically persisted via checkpointing (1) as schema-consistent records, where identical attributes are recorded per timestep or event. Hierarchical formats such as HDF5 [29] and ROOT [30] are widely used to organize this data into nested groups and datasets. Due to their repetitive structure, such records are often convertible into tabular form. Recently, columnar formats like Parquet [31] has gained popularity for analytical workloads, leading to its adoption as a native storage format [32], [33] and motivating the conversion of HDF5 and ROOT data for improved compatibility and scalability [34].

3) **Data Analysis:** For analyzing tabular data, distributed data processing engines such as Apache Spark [6] and Relational Database Management Systems (RDBMS) are commonly used. In Spark-based environments, users can perform efficient analysis on specific Regions of Interest (ROI) using either SQL queries or high-level API such as the Spark DataFrame API [8], [35], [36]. Upon receiving an analysis request (2), the system loads relevant data from storage into compute nodes (3), where queries are executed to extract scientifically meaningful subsets (4).

4) **Data Visualization:** These extracted subsets are subsequently leveraged for downstream analytical tasks, including

visualizing simulation results (5), comparing with experimental observations, and validating the accuracy and reliability of the simulation outcomes.

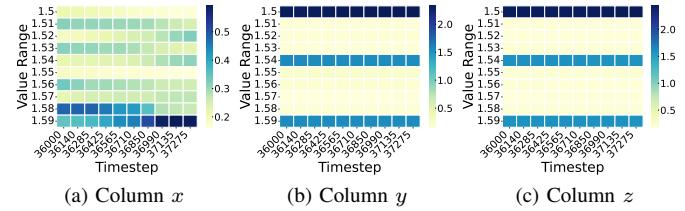


Fig. 3: Heatmaps showing distribution rates (%) of filtered  $x$ ,  $y$ , and  $z$  values across timesteps in the Laghos dataset [28].

### B. Low-Selectivity Queries in Scientific Workflows

Prior studies have shown that, in many scientific workflows, queries often require only a small subset of data despite scanning the entire dataset [37]–[39]. To further understand this, we analyzed real-world queries from the CFD domain using the Laghos 3D Mesh Dataset [40] and representative query patterns from Los Alamos National Laboratory (LANL) (refer to the example SQL query in Figure 2).

Figure 3 visualizes the distribution density of records in the Laghos dataset that satisfy the filter condition  $1.5 < x, y, z < 1.6$ . The horizontal axis denotes simulation timestep IDs, and the vertical axis divides the 1.500–1.600 value range into ten uniform bins of width 0.01. Subfigures (a)–(c) respectively illustrate the density distributions for  $x$ ,  $y$ , and  $z$  dimensions (columns). Most bins exhibit zero density, and even the maximum bin value does not exceed 2%. Given that the query requires all three coordinates to fall within the specified ROI range, only overlapping non-zero bins contribute to the result. The resulting compound selectivity is as low as  $1.91 \times 10^{-4}\%$ , indicating an extremely sparse subset of relevant records. This high sparsity reflects a common pattern in domain-specific workflows, where scientists often focus on localized regions or rare events within large-scale simulation outputs [41].

### C. Well-Known Challenges of Data Movement and Storage

Scientific workflows produce massive datasets that are persisted to storage and repeatedly moved between storage and compute nodes for post-hoc analysis. While distributed

engines such as Spark support scalable analysis, their performance is constrained by access to remote data. Specifically, modern HPC storage systems face two key limitations that hinder scalable analysis: *data movement* and *data placement*.

**Challenge#1—Massive Data Movement.** While compute performance in HPC systems has improved rapidly, I/O bandwidth between storage and compute nodes has not scaled accordingly. As a result, data movement from storage to compute nodes dominates analysis runtimes [41], even when queries access only a small subset of the data. This imbalance is exacerbated by growing simulation output sizes and increasing demands for faster turnaround, leading to higher latency, energy consumption, and infrastructure costs [9].

**Challenge#2—Placement-Access Frequency Mismatch.** Scientific datasets in columnar formats often exhibit skewed access patterns, where certain columns are accessed far more frequently than others [16], [17]. Nevertheless, most HPC storage systems rely on POSIX-based flat file abstractions that lack semantic awareness of column-level locality. This limitation is especially problematic in tiered storage hierarchies comprising HDDs and SSDs, where all data is managed uniformly regardless of access frequency [16]. This mismatch leads to full-file reads from high-latency storage and underutilization of high-performance devices such as NVMe SSDs, degrading bandwidth efficiency and overall system performance.

#### D. The Advent of Computation-Enabled Object Storage

**Computational Storage for Data Movement Reduction:** To mitigate the data transfer bottleneck, recent efforts have investigated computational storage, which enables basic operations such as *filter* and *project* to be offloaded to the storage layer [9], [41], [42]. This approach reduces the volume of data transferred to compute nodes, thereby accelerating analysis and alleviating pressure on network and I/O subsystems. It is particularly effective in low-selectivity scenarios where only a small portion of the dataset satisfies the query predicates.

**Improving Data Placement with Object Storage:** To overcome the limitations of uniform data placement, the HPC community is increasingly adopting object storage [43]. Unlike POSIX-based file systems, object storage manages data as discrete objects enriched with metadata, allowing more granular control over storage policies. This facilitates adaptive tiering, where frequently accessed columns can be placed on fast NVMe SSDs while rarely accessed data resides on high-capacity HDDs. Thus, AWS S3 [44]-compatible object storage systems such as Ceph [19] and MinIO [18] are already in wide use at large-scale research sites such as Conseil Européen pour la Recherche Nucléaire (CERN) and Institute of High Energy Physics (IHEP) [17], [45]. Furthermore, commercial offerings including IBM COS and NetApp StorageGRID which further accelerate this trend [46], [47].

**Computation-Enabled Object Storage:** To address the dual challenges of excessive data movement and suboptimal data placement, recent research has introduced Computation-Enabled Object Storage (COS) systems [21]–[23]. These systems integrate the flexible, metadata-rich structure of object

storage with computation capabilities, enhancing both query efficiency and storage utilization.

There are two main design approaches. The **first** extends the object storage interface to support lightweight offloaded computation. Systems such as MinIO Select [22] and Ceph S3 Select [23] enable SQL-like filter queries to be executed directly at the object interface layer, reducing data transfer by pushing down simple operations. The **second** approach embeds computation deeper into the storage stack. A representative example is SkyhookDM [21], which integrates data processing capabilities into Ceph OSD layer using Apache Arrow [26]. While SkyhookDM supports basic operations such as *filter* and *aggregate*, it also allows users to define custom processing logic within the storage backend. By enabling more expressive query offloading at the storage layer, this integration establishes storage-side computation as a foundational building block for scalable and efficient scientific data analysis.

### III. MOTIVATION

While COS solutions have gained attention for supporting query processing at the storage layer, their alignment with the query characteristics of real-world HPC workloads remains largely limited. The effectiveness of query offloading varies significantly depending on the types of operators involved [48], [49]. For instance, *filter* operations, which can substantially reduce data volume, are well-suited for execution at the storage layer. In contrast, high-complexity operations such as *join* incur greater computational overhead and offer limited reductions in data transfer, making them less suitable for offloading. Nevertheless, there has been little quantitative analysis of the query patterns employed in HPC workloads that process large-scale tabular data, such as those in the CFD, PIC, and HEP domains. Consequently, empirical evidence remains insufficient for evaluating the applicability and limitations of storage-based query offloading in these contexts.

#### A. Query Characteristics of Tabular HPC Workloads

To quantitatively analyze the query patterns observed in HPC workloads of CFD, HEP, and PIC domains, we collected real-world queries from publicly available analytical workloads released by institutions such as CERN, LANL, and ORNL [34], [40], [50]–[52]. All collected queries operate on tabular scientific datasets. We performed a quantitative structural analysis based on the types of operators used and the ways in which they are composed.

For this analysis, we classified the queries into four categories based on their operator composition:

- **Filter:** Queries that include only simple *filter* predicates.
- **Filter+Agg/Sort:** Queries that combine *filter* with *aggregate* or *sort* operations.
- **Project:** Queries that include only *project* operation
- **Join:** Queries that involve complex operators such as *join*.

Table I summarizes the clustering results of queries across each scientific domain. In the table, **Predicate** indicates whether the data fields used in operator predicates are

TABLE I: Characteristics of representative queries found in CFD, HEP, and PIC scientific domains.

Predicate	Type	Filter	Filter+Agg/Sort	Project	Join
Scalar	Cmp.	18	2	0	0
	Arith.	2	3	9	0
Array	Cmp.	3	0	0	0
	Arith.	10	1	7	0
User-Defined Func		0	0	2	0
No Predicate		0	0	9	0
<b>Total</b>		33	6	27	0

scalar or array-based, while **Type** distinguishes between arithmetic (Arith.) and comparison (Comp.) operations. Among all queries in the table, **Filter** appeared in 33 cases, **Filter+Agg/Sort** in 6 cases, and **Project** in 27 cases. Notably, there was not a single instance of a complex query involving a *join* operation across multiple columns. The **Filter** and **Filter+Agg/Sort** applied predicates exclusively to individual column values, while **Project** either selected specific columns or generated new ones through column-wise computation. These results suggest that the majority of queries operate on a narrow set of columns, without engaging in row-wise processing over entire records. An in-depth analysis further reveals that all queries explicitly reference only the required columns, with non-essential fields excluded from computation.

Publicly available HPC queries primarily consist of simple operations centered around *filter* or *project*. This aligns with the typical usage patterns of COS systems and suggests that SQL offloading techniques such as filter pushdown can reduce data transfer and improve analytic performance. Meanwhile, as shown in the **Array** section of Table I, complex operator conditions frequently involve computations that go beyond simple scalar comparisons. For example, expressions such as *Muon\_charge[0] != Muon\_charge[1]* involve element-wise comparisons or computations within array-structured columns. This analysis yields the following three key observations.

**Observation 1.** Most queries follow a column-based analytics pattern, selectively accessing only the required columns without scanning entire rows.

**Observation 2.** The core operations are concentrated on *filter*, *project*, *aggregate*, and *sort*, with no occurrence of relational operations such as *join*.

**Observation 3.** Predicate conditions frequently go beyond simple scalar range comparisons, often involving computations between elements within array-structured columns.

These three observations clearly articulate the core requirements for designing next-generation COS storage systems. Specifically, such systems should (i) support columnar I/O and processing, (ii) enable efficient in-storage execution of core relational operators such as *filter*, *project*, *aggregate*, and *sort*, and (iii) support predicate evaluation involving element-wise computation over array-valued columns. Satisfying these requirements is critical to fully realizing the performance advantages of compute offloading in HPC analytical workloads.

### B. Limitations of Existing COS Systems

#### Limitation#1–Limited Output Formats for Columnar Semantics and Compatibility:

MinIO Select and Ceph S3 Select support filtering at the storage layer but serialize the results in a row-oriented format. This necessitates reconstructing the data into a columnar layout, during which column statistics and structural metadata are lost. As a consequence, downstream analytical engines are unable to apply optimizations such as filter skipping or projection pruning, potentially resulting in redundant reprocessing along the full query path. SkyhookDM returns query results exclusively in the Arrow IPC format. While this preserves columnar locality, the fixed output format reduces interoperability. Engines that do not provide native support for Arrow, such as Presto [53], require an additional conversion layer, which complicates integration.

#### Limitation#2–Restricted Support for SQL Operators and Array Expressions:

Current COS solutions exhibit limited operator support and lack the ability to express array-level expressions, even though HPC queries are often simple and structurally well-defined. MinIO Select and Ceph S3 Select support only simple filtering conditions, basic projection, and regular expression matching, but do not support more advanced operators such as *aggregate*, *sort* or predicate evaluation over array elements [54]. These limitations prevent the storage layer from processing queries involving array-based predicates and *aggregate* or *sort*, both of which are frequently observed in real HPC workloads. The absence of these features requires entire files or row groups to be transferred to compute nodes, resulting in significant inefficiencies.

SkyhookDM leverages Apache Arrow’s compute kernels to offload *filter* operations, including array element indexing and arithmetic conditions. Although it supports extensibility via user-defined kernels for operations not natively available in Arrow, integrating such kernels requires recompiling the Skyhook-specific libraries within the Arrow ecosystem. This requirement limits flexibility and imposes additional overhead on users. More critically, *aggregate* and *sort* are not natively supported, and the *project* operator handles only direct column selection. As a result, projections involving computed expressions must be executed on the compute node. Consequently, analytical queries with advanced operators or complex array logic cannot be flexibly offloaded, reducing the overall effectiveness of storage-side computation.

#### Limitation#3–Excessive Inter-Layer Data Movement Due to Fixed Execution Layer:

Existing COS systems typically support computation at only a single logical layer, which leads to structural inefficiencies. For example, systems like MinIO Select or Ceph S3 Select perform operations at the S3 interface level, requiring data to be collected from storage nodes to a gateway node that handles client S3 requests and storage backend coordination. This architecture incurs data movement bottlenecks between storage backend and the gateway. Even systems such as SkyhookDM, which execute computation at the Object Storage Daemon (OSD) level, are restricted to single-layer execution. This limitation prevents the effective

use of emerging lower-tier compute resources, such as DPU-based Just a Bunch of Flash (JBOF) arrays [55], [56], whose computational capabilities remain largely underutilized.

In hierarchical storage environments, data movement occurs not only across networks but also between internal layers. Fixing computation at a single layer prevents early-stage reduction and results in excessive inter-layer data transfers. To alleviate this, computations should be initiated at lower layers closer to the data, to progressively reduce volume before reaching upper layers.

#### IV. DESIGN OF OASIS

##### A. Design Principle

To address the aforementioned limitations, we design a new COS architecture that preserves the benefits of columnar layout and in-storage computation, while introducing complex operators with array semantics and enabling flexible operator decomposition across execution tiers within the storage stack. Guided by these goals, we present the core Design Principles (DP) that shape the architecture of OASIS.

- DP#1: Column-Oriented Output Format Support for HPC Analytics.** OASIS should support both Arrow and CSV outputs to enable efficient columnar processing while preserving compatibility with applications that lack native Arrow support, such as Presto.
- DP#2: Storage-Level Query Execution with Advanced Operator and Array Support.** As we analyzed in §III-A, HPC queries often involve *aggregate* and *sort* operations over simulation units, and array-level expressions on physical quantities. OASIS must support these operators and array semantics for effective and practical in-storage execution.
- DP#3: Storage-Aware Query Path Optimization.** OASIS should reduce internal data movement by generating hierarchical query execution plans that place high data movement cost operations closer to storage and minimize interconnect and network traffic.

##### B. Overview

The proposed OASIS system consists of a OASIS frontend (OASIS-FE) and multiple storage array (OASIS-A) servers. The OASIS-FE mainly comprises the following components (§IV-C): the *S3 Gateway*, the *Local Optimizer*, the *Metadata Manager*, the *Query Executor and Result Handler*, and the *NVMe-over-Fabrics (NVMe-oF) Initiator module*. Each OASIS-A is connected to the OASIS-FE via NVMe-oF, enabling high-throughput I/O. The OASIS-A consists of the following components (§IV-D): the *NVMe-oF Target*, the *Storage Manager*, and the *Query Executor and Result Handler*.

Figure 4 illustrates the end-to-end query offloading process in the OASIS system. ① The process begins when a client-side query engine submits an SQL query. ② The query is translated into an Intermediate Representation (IR), Substrait [57], which describes the operator-level execution plan (§IV-F). This IR plan is transmitted to the OASIS-FE via the OASIS-extended Pushdown (P/D) API integrated into the client-side query engine (§IV-H). ③ The S3 Gateway forwards the IR plan to

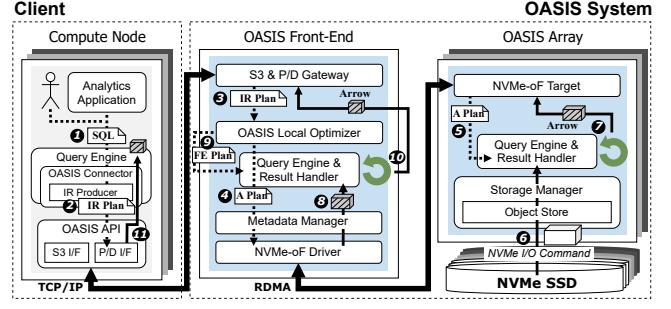


Fig. 4: Overall architecture of OASIS and a query processing flow. A query task in Substrait IR format is sent to OASIS via the P/D Interface (I/F) within the client-side query engine.

the Local Optimizer. ④ The Local Optimizer partitions the IR plan into a OASIS-FE plan and a OASIS-A plan. The OASIS-FE plan depends on the intermediate result of the OASIS-A plan, establishing an execution dependency. ⑤ The OASIS-A plan is sent to the corresponding OASIS-A for execution. ⑥ The target object is retrieved from local storage via the Storage Manager. ⑦ The OASIS-A query engine executes the assigned plan and returns the intermediate result to the OASIS-FE. ⑧ The OASIS-FE registers the result into its query engine as a temporary table. ⑨ The OASIS-FE executes the OASIS-FE plan, ⑩ producing and ⑪ returning the final result.

##### C. Core Components of OASIS-FE

1) **S3 Gateway:** This module acts as a network-facing entry point that bridges external S3 clients and the internal components of OASIS. It handles S3 protocol parsing and translates standard operations such as *PutObject* and *GetObject* into internal gRPC messages. These messages encapsulate request metadata and payloads, which are then forwarded to downstream modules for further processing, with responses returned in S3-compatible HTTP format.

2) **Local Optimizer:** This module analyzes and decomposes offloaded Substrait IR query plans using the Storage-side Query Plan Offloading and Decomposition Algorithm (SODA). It estimates the data transfer volume per operator to determine the optimal execution layer, either at the OASIS-FE or the OASIS-A. Then, the Local Optimizer generates subplans for the OASIS-FE and OASIS-A that account for both operator semantics and communication overhead, enabling efficient query execution while minimizing data movement.

3) **Metadata Manager:** This module manages execution-related metadata and performs logical-to-physical translation of requests received from the S3 Gateway. Upon receiving a gRPC message, it resolves the target object by mapping the S3 bucket name and object key to an internal Object Space ID and Object ID, maintained in its mapping table. It then generates the corresponding NVMe I/O commands and dispatches them to the appropriate OASIS-A for execution. When a new bucket is created, a corresponding OASIS-A is designated and allocates a unique Object Space ID for routing subsequent I/O requests. Additional metadata such as object size is included in each request.

To assist the Local Optimizer in operator-level query decomposition, the Metadata Manager collects lightweight statistical metadata during object ingestion. Specifically, when a *PutObject* operation occurs, a compact histogram is generated via sampling and stored locally on the OASIS-FE using the object key as an index. These histograms typically cover 0.5–5% of the object and capture column-level distributions for various data types such as double and int. The Local Optimizer later uses this metadata to estimate operator selectivity and output size, guiding plan partitioning and placement.

**4) Query Executor and Result Handler:** This module provides a lightweight, columnar-aware execution engine capable of running queries directly at the storage tier. It supports array-level expressions to efficiently evaluate scientific workloads involving nested or repeated structures. After execution, results are serialized into client-specified formats such as Arrow for high-throughput transfer or CSV/JSON for legacy compatibility. Further implementation details are described in §IV-E.

**5) NVMe-oF Initiator:** This component acts as a network interface that issues NVMe I/O commands to OASIS-A servers over NVMe-oF. The OASIS-FE converts object storage requests into NVMe commands and transmits them over a low-latency RDMA transport. Object data and metadata are encapsulated into a single extended NVMe command [58].

#### D. Core Components of OASIS-A

**1) NVMe-oF Target:** The NVMe-oF Target module receives NVMe commands from the OASIS-FE and performs object I/O operations accordingly. Each command encapsulates both object data and metadata, allowing the OASIS-A to directly access memory and execute read/write operations without separate metadata handling. By leveraging an extended block command format, this design enables single-command execution with minimal data copying and supports high-throughput memory-based parallel I/O.

**2) Storage Manager:** The Storage Manager is responsible for handling storage and retrieval requests issued from the OASIS-FE using an internal object store. It manages data at the blob level through a Blob Property Table (BPT), which maps Object Space IDs and Object IDs to physical offsets on the storage device. Each blob follows an OPEN–RUN–CLOSE lifecycle during I/O, where DMA buffers and I/O channels are initialized for asynchronous execution. Write-Ahead Logging (WAL) ensures metadata consistency, and a slice-based address space is employed to improve scalability and physical alignment. While full implementation details are beyond the scope of this paper, this layer adopts core design principles from conventional object storage systems and provides a robust and scalable backend for upper-layer query execution.

**3) Query Executor and Result Handler:** This component mirrors the functionality of its counterpart in the OASIS-FE, executing assigned sub-plans and returning intermediate results. To enable fast and efficient communication between the OASIS-A and the OASIS-FE, intermediate results are serialized using the Arrow format and streamed back to the OASIS-FE. Further details are described in §IV-E.

#### E. In-Storage Query Execution Engine and Result Transfer

The Query Executor and Result Handler are core components of OASIS, responsible for realizing **DP#1** and **DP#2** within the storage layer by enabling columnar query processing within the storage layer. Accordingly, the execution engine must satisfy three key requirements: (1) native support for columnar formats such as Arrow, (2) evaluation of expressions on individual elements within array-typed columns, and (3) a lightweight execution environment that operates reliably under constrained computational resources.

To fulfill these requirements, we adopt DuckDB [27] as the core execution engine within the storage layer. DuckDB is an open-source, embedded RDBMS optimized for Online Analytical Processing (OLAP) workloads, offering native support for columnar formats like Parquet and Arrow. It supports a wide range of SQL operators—including complex aggregations, sorts, and array-level functions—making it particularly effective for offloading diverse query fragments. Notably, DuckDB supports both in-memory processing and disk-based execution, allowing it to efficiently process large-scale columnar datasets ranging from tens to hundreds of gigabytes without requiring full in-memory loading. This makes it well-suited for lightweight, in-storage execution.

To enable execution across hierarchical layers while preserving the benefits of columnar processing for **DP#3**, the Query Executor serializes intermediate results and final outputs in compressed Arrow format for efficient transfer to the upper layer. For compatibility with legacy tools, final outputs can also be emitted in CSV or JSON format.

Arrow’s columnar layout and zero-copy semantics minimize serialization overhead and enable efficient downstream execution. DuckDB natively supports this Arrow-based execution pipeline through Arrow Database Connectivity (ADBC) [59], allowing query execution result tables to be serialized into Arrow and exported to external memory buffers. These buffers can then be transferred with minimal overhead, using RDMA for intra-system communication and gRPC for external delivery. Result Handlers are deployed at both the OASIS-FE and the OASIS-A, where they work in conjunction with their local query engines to forward intermediate or final result tables to the next processing layer.

#### F. Operator-Level Query Plan Optimizer and Decomposer

The Local Optimizer is responsible for analyzing the Substrait IR-based query plan received from the client and partitioning it across execution layers based on operator characteristics and data transfer cost. It consists of two core components. First, an optimization algorithm (§IV-G) identifies the optimal decomposition point for execution. Subsequently, the Substrait Decomposer utilizes this point to partition the original plan into two semantically equivalent subplans, thereby enabling efficient query execution across heterogeneous layers.

**Why Substrait IR?** OASIS adopts the Substrait IR [57] to enable fine-grained operator-level partitioning and modular execution across layers. Substrait is a cross-platform, language-

agnostic IR designed for interoperability between query engines and execution backends. Compared to SQL, it offers a more structured and expressive form, explicitly encoding operator types, input/output schemas, and expression trees. This structure allows the system to efficiently isolate and recompose subplans with minimal transformation overhead [60]. It also facilitates extensibility through its `extensionURI` mechanism, enabling the use of user-defined functions and non-standard operators with formal external definitions. When the client submits a SQL query, the system translates it into a Substrait IR-based query plan, embedding all necessary metadata such as schema and object references (§ IV-H).

**Substrait Decomposer.** The Substrait Decomposer divides the query plan into two parts based on the operator selected by the optimization algorithm as the decomposition point. The decomposition process begins by traversing the original relational tree to locate the target operator that serves as the decomposition point. Once identified, the subtree rooted at this operator is extracted to construct the OASIS-A subplan. In the original plan, the corresponding node is replaced with a new `read` operator, thereby forming the OASIS-FE subplan (Figure 5). Each resulting subplan is subsequently reconstructed

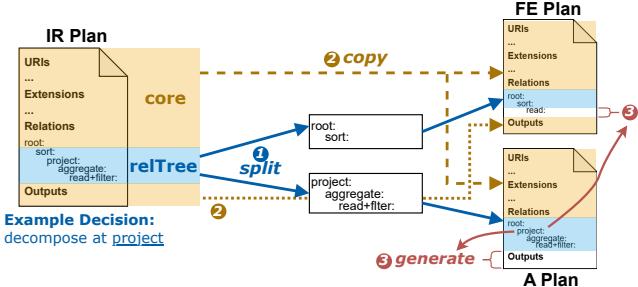


Fig. 5: Illustration of the mechanism by which the IR plan is decomposed into the OASIS-FE plan and the OASIS-A plan.

to comply with the Substrait specification. Specifically, the `extensionURI` and `extensions` sections from the original plan are selectively replicated to ensure that all custom operators and functions used in each subplan are correctly defined. The schema of the intermediate result produced by the OASIS-A plan is inferred by analyzing the output structure of the extracted subtree, including operators such as `aggregate`, `project`, and `read`. This analysis captures grouping keys, column names, and data types. To avoid naming conflicts, temporary column names are systematically generated using a unique alphabetical naming convention. Once the intermediate schema is finalized, it is applied consistently to both subplans. The OASIS-A plan is explicitly configured to emit this schema, while the OASIS-FE plan introduces a new `read` operator that declares the same schema as its input.

This decomposition process forms a semantic bridge between subtree extraction and plan reconstruction. It preserves the logical continuity of data flow while enabling physical modularization of execution. Importantly, the resulting plans are not merely structurally separated but also logically ordered: since each operator in the original Substrait IR plan follows a

Directed Acyclic Graph (DAG) based on data dependencies, the OASIS-A plan must be executed first to produce the intermediate output, which is then consumed by the OASIS-FE plan. This strict dependency preserves the original query semantics while enabling efficient execution across disaggregated compute-storage layers.

#### G. Optimization Strategies for Query Plan Decomposition

The SODA employed in the Local Optimizer uses two strategies to split Substrait IR plans for HPC tabular queries. The first strategy, Coefficient-Aware Decomposition (CAD), estimates input-to-output ratios (coefficients) using pre-built histograms and selects a split point that minimizes data movement. It is suited for queries involving scalar-based conditions or simple computations. The second strategy, Structure-Aware Placement (SAP), applies when coefficients cannot be reliably estimated from statistics or histograms, such as in queries with array-level conditions or computations. SAP considers the physical data layout to place operators close to the data.

1) **Operator Classification:** It is critical to estimate the data movement introduced by each operator when decomposing a query plan into subplans for the OASIS-FE and OASIS-A. This cost primarily depends on the output size of the last operator executed on the OASIS-A, since its result is passed to the OASIS-FE when execution moves to the upper layer.

If the transition occurs after the  $i$ -th operator, the output of that operator becomes the input to the  $(i+1)$ -th and constitutes the intermediate data that must be transferred. The size of this intermediate result directly impacts the total data movement overhead. To quantify this, OASIS defines a per-operator input-to-output coefficient based on Substrait operator semantics. Using these coefficients, it estimates the input and output sizes of each operator from the initial input size. Operators are then classified into four categories based on their data transformation behavior, as summarized in Table II.

TABLE II: Classification of Substrait operators by type.

Type	Input-Output Relationship	Substrait Relations
Op 1	Single parent, 1:1	<code>read</code> , <code>sort</code>
Op 2	Single parent, 1: $x$ ( $x \leq 1$ )	<code>filter</code> , <code>project</code> , <code>aggregate</code>
Op 3	Single parent, 1: $x$ ( $x > 1$ )	<code>expand</code>
Op 4	Dual parent, 1: $x$ ( $x > 0$ )	<code>join</code> , <code>set</code>

As shown in §III-A, all operators involved in HPC tabular queries fall into either the Op-1 or Op-2 categories. Operators in Op-3 and Op-4 (e.g., `join`) are thus excluded from coefficient-based cost estimation. Op-1 operators have identical input and output sizes, yielding fixed coefficients. In contrast, Op-2 operators produce variable output sizes depending on filter selectivity or the number of projected columns, resulting in dynamic coefficients. To estimate this variability, OASIS constructs offline histograms at data ingestion time. These histograms capture column value distributions and are later used to estimate filter selectivity and projection effects, allowing accurate output size estimation for Op-2 operators.

Crucially, coefficient estimation is not only used to predict output sizes but also plays a central role in modeling total data movement. Starting from the input size of the initial *read* operator, OASIS performs chained inference across the operator tree, applying estimated coefficients to compute the input and output sizes of subsequent operators. By combining operator classification, histogram-based estimation, and coefficient inference, OASIS builds a cost model focused on data transfer. This enables the system to identify the optimal query plan split point between the OASIS-FE and OASIS-A, minimizing internal data movement and maximizing offloading efficiency.

2) **Coefficient-Aware Decomposition:** CAD is a strategy designed for query plans involving schemas with scalar-based conditions or computations, where output size can be predicted using coefficient estimation. CAD sequentially infers the input and output sizes of all operators based on their input-output coefficient and the initial input size. To determine the optimal split point, we make the following assumption: *Query plans are split under the assumption of one-way data transfer from the lower layer (OASIS-A) to the upper layer (OASIS-FE), without return traffic.* This assumption avoids unnecessary costs from round-trips and ensures that once sufficient data reduction occurs at the lower layer, intermediate results can be efficiently transferred and processed at the upper layer.

CAD determines the optimal split point through three sequential steps: (1) Estimate operator-specific input-to-output coefficients using prebuilt histograms; (2) Propagate input and output size estimates across the operator tree based on these coefficients and the initial input size; (3) Select a split point based on two criteria: (a) if a semantic boundary requiring centralized processing (e.g., global *sort*) is encountered before further data reduction is possible, the plan is split at that point; (b) If maximal data reduction is achieved, execution continues on the OASIS-A until a boundary appears, avoiding unnecessary memory transitions and operator materialization in the upper layer.

A representative boundary is the *sort* operator, which requires global ordering and must be merged at the upper layer after partial processing. In contrast, operators like *aggregate* can be safely offloaded, as their commutative and associative properties enable partial aggregation at the lower layer and finalization at the upper layer. Functions like *MEDIAN*, however, rely on global ordering and cannot be decomposed into partial forms.

3) **Structure-Aware Placement:** SAP is a decomposition strategy designed for query plans involving array-level conditions or computations, typically found in schemas with nested structures such as *List* or *Array*. In such cases, coefficient estimation using Parquet statistics or prebuilt histograms is infeasible, as these are collected at the column level and do not capture intra-array value distributions. Because the output size of such operations depends on runtime evaluations over individual array elements, the CAD strategy is not applicable.

To address this, SAP mandates that any condition or expression directly referencing array elements be evaluated at the OASIS-A level. For example, a predicate such as  $a[i] + a[j] < 0$ ,

which depends on the runtime values of individual array elements, cannot be statically predicted and must therefore be executed at the data-resident layer. SAP proceeds in three steps: (1) Analyze the query plan to detect array-aware predicates that reference internal items; (2) Enforce the evaluation of such predicates at the OASIS-A to ensure locality; (3) Evaluate the resulting data size at runtime, and apply a lazy execution strategy that transmits results to the OASIS-FE only when the output remains within acceptable transfer limits or when the boundary requiring centralized processing is encountered.

By statically determining the plan split while dynamically evaluating result transfer sizes, SAP enables effective offloading even when coefficient-based estimation is infeasible. This approach supports fine-grained filtering of nested structures near the data, reducing unnecessary transfers and improving overall processing efficiency.

#### H. Client Integration via IR Producer and Pushdown API

Client-side integration with OASIS is enabled via a custom connector, illustrated here using Spark as a representative example. The connector consists of two main components (see Figure 4): (1) an **IR Producer** that translates the SQL query into a Substrait IR, and (2) a **P/D API** that transmits the IR plan to the OASIS-FE via gRPC for pushdown execution. Users can access data via the standard `.read.format("...")` interface without modifying their existing Spark applications. Final query results are serialized in Arrow format and returned to the client, where they can be deserialized directly into Spark DataFrames through the Arrow source interface for further analysis or visualization. This design enables drop-in compatibility with existing Spark pipelines while leveraging the flexibility of the Substrait IR to support integration with other query engines in the future.

## V. EVALUATION

### A. Experimental Setup

**Implementation:** We implemented a prototype of OASIS by building the OASIS-A using SPDK [61] v23.09, extending its BDEV layer to incorporate the Storage Manager and Result Handler. The in-storage Query Executor is built on DuckDB [27] v1.3.0. On the OASIS-FE, we employ Versity Gateway

TABLE III: Details of the hardware specifications used to configure the OASIS system and the Spark cluster.

System	Component	Specification
Spark Cluster	Driver	CPU: Intel® Xeon® Gold 6226R (3.9 GHz max) Cores: 64 cores Memory: 386 GB DDR4
	Executor	CPU: Intel® Xeon® Gold 6330 (3.1 GHz max) Cores: 112 cores Memory: 128 GB DDR4
OASIS	OASIS-FE	CPU: Intel® Xeon® Silver 4410Y (3.9 GHz max) Cores: 48 cores Memory: 64 GB DDR4
	OASIS-A	CPU: Intel® Xeon® Silver 4410Y (2.0 GHz max) Cores: 16 cores Memory: 64 GB DDR4 Storage: 1 TB NVMe SSD + 512 GB SATA SSD

[62] v2.49.2 for S3 compatibility and implement the Metadata Manager, Result Handler, and Local Optimizer in a C++ backend server. Communication between the OASIS-FE and OASIS-As is handled via the NVMe-oF initiator in the Linux kernel v5.15.0.

**OASIS and Analytics Cluster Setup:** To configure OASIS, we used a 48-core, 64 GB memory server as the OASIS-FE, and a server with identical specifications but limited to 16 cores as the OASIS-A and equipped 1 NVMe SSD.

We deployed a Spark cluster with Spark 4.0.0 consisting of a 64-core, 386 GB memory server as the Spark driver and two 112-core, 128 GB memory servers as Spark executor nodes. The server specifications for the OASIS-FE, OASIS-A, and the Spark cluster are listed in Table III. The OASIS-FE and OASIS-A are connected via a 10GbE RDMA network for SPDK, while OASIS communicates with the Spark cluster over 10GbE Ethernet network.

**Workload:** For evaluation, we selected three real-world scientific workloads: (1) Laghos [28] 3D Mesh Simulation, which models shock hydrodynamics in a Lagrangian framework, (2) DeepWater-Impact Simulation [63], which simulates the interaction of water with rigid bodies in marine environments; and (3) CMS Open Data [64], which consists of high-energy physics collision event records collected at the CERN LHC.

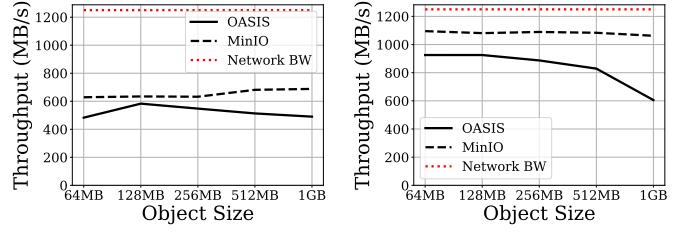
The Laghos dataset [40] is 20GB in size. The DeepWater-Impact workload comprises two datasets [50] of 13GB and 30GB, respectively, which differ in simulation resolution. The CMS dataset contains 12GB of data. All datasets were converted to the Parquet format for analysis and are publicly available.

Table IV shows the queries using these datasets. Q1 computes the average energy per vertex within a specified spatial region using the Laghos dataset. Q2 extracts fluid elements from the Deep Water Impact simulation based on the thresholds. Q3 analyzes the vertical extent of dynamic fluid activity over time in the high-resolution Deep Water dataset. Q4 identifies opposite-charge muon pairs with invariant mass between 60 and 120 GeV in CMS event data.

To evaluate the effectiveness of OASIS as a COS, we investigate the following research questions (RQs).

- **RQ#1:** Does OASIS provide object-level I/O performance for scientific datasets comparable to existing COSs?
- **RQ#2:** How effective is OASIS’s hierarchical operator execution strategy in improving query performance when evaluated under a uniform storage configuration?
- **RQ#3:** Does the Arrow-based output of OASIS offer performance advantages over traditional CSV formats in analytical workflows?
- **RQ#4:** How does selectivity affect the performance of OASIS in scientific query workloads?
- **RQ#5:** How effective is OASIS’s SODA strategy compared to other decomposition approaches in hierarchical execution?

**Comparison:** To answer these research questions, we conducted a series of experiments using the following four con-



(a) PUT throughput

(b) GET throughput

Fig. 6: Throughput comparison of OASIS and MinIO.

figurations.

- **Baseline:** Executes queries using standard Spark processing, where all computation is performed after retrieving data from storage.
- **Pred.:** Extends the Baseline by enabling predicate push-down to the storage layer.
- **OASIS:** Represents our proposed design that employs hierarchical execution across storage layers by SODA.
- **COS:** Emulates how OASIS would operate under the computation model of COSs by executing all operators at the OASIS-FE. This configuration ensures that the execution layer follows the same layer as in existing COS systems, allowing to isolate the effects of hierarchical execution without interference such as I/O overhead.

### B. Object-Level I/O Performance

To evaluate I/O performance for RQ#1, we compared the object-level PUT and GET throughput of OASIS against MinIO, using 16 threads and object sizes from 64MB to 1GB. This reflects typical scientific workloads, where data is written in parallel in 64–256MB blocks, such as Parquet row groups.

Figure 6 shows the results, with the 10Gbps network bandwidth used as the upper bound. In Figure 6(a), MinIO achieves up to 681.5MB/s, while OASIS peaks at 582.9MB/s but degrades with larger objects due to gRPC overhead from sending many small messages and lack of parallel buffer management. Neither system saturates the 10GbE link, mainly due to checksum generation and verification. OASIS is further limited by gRPC costs, including message fragmentation and lack of parallel buffer handling. Figure 6(b) shows similar trends for GET. MinIO sustains over 1,080MB/s, while OASIS drops to 605.6MB/s at 1GB due to the same bottlenecks. Overall, OASIS lags behind MinIO for large objects, but further optimization of buffer management and messaging is expected to close the gap and attain comparable performance.

### C. Effect of Hierarchical Execution on Query Performance

To evaluate the effectiveness of hierarchical execution in addressing RQ#2, we performed a series of experiments. OASIS leverages SODA for optimal execution planning, with a detailed analysis provided in § V-F.

1) *Queries involving Scalar-based Conditions:* We evaluated three queries (Q1-Q3) involving scalar-based conditions. Each query differs in form, as shown in Table IV. As illustrated in Figure 7, OASIS consistently achieves the lowest execution

TABLE IV: Realistic science discovery queries over datasets from scientific workloads.

Query	SQL Statement
Q1	SELECT min(vertex_id) AS VID, min(x) AS X, min(y) AS Y, min(z) AS Z, avg(e) AS E FROM parquet WHERE x > 1.5 AND x < 1.6 AND y > 1.5 AND y < 1.6 AND z > 1.5 AND z < 1.6 GROUP BY vertex_id ORDER BY E;
Q2	SELECT rowid, v03 FROM parquet WHERE v03 > 0.001 AND v03 < 0.999;
Q3	SELECT MAX((rowid % (500 * 500)) / 500) AS height, TIMESTEP FROM parquet WHERE v02 > 0.1 GROUP BY timestep;
Q4	SELECT MET_pt, sqrt(2 * Muon_pt[1] * Muon_pt[2] * (cosh(Muon_eta[1] - Muon_eta[2]) - cos(Muon_phi[1] - Muon_phi[2]))) AS Dimuon_mass FROM parquet WHERE nMuon = 2 AND Muon_charge[1] != Muon_charge[2] AND sqrt(2 * Muon_pt[1] * Muon_pt[2] * (cosh(Muon_eta[1] - Muon_eta[2]) - cos(Muon_phi[1] - Muon_phi[2]))) BETWEEN 60 AND 120;

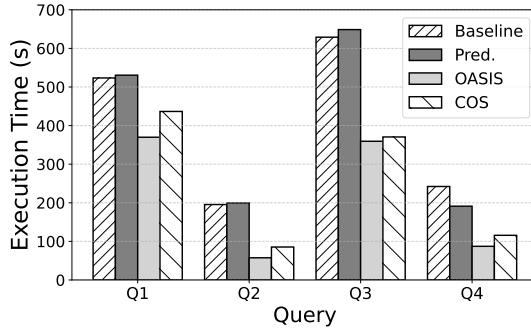


Fig. 7: Execution time comparison for the queries across four system configurations. Note that Q1–Q3 use scalar-based conditions, whereas Q4 involves an array-based condition.

time across both queries. In scalar-based query evaluation, we assume that **COS** supports all candidate operators, as operator support may vary across systems. This assumption allows us to isolate and validate the impact of hierarchical execution across different queries.

For Q1 and Q2, **OASIS** outperforms **COS** by 15.27% in Q1 and 32.7% in Q2 by minimizing internal data movement between the storage and compute layers, resulting in sharper performance gains. In Q3, **OASIS** continues to deliver the best performance, although the performance gap between **OASIS** and **COS** narrows. This is because Q3 is compute-intensive, and the performance benefit from reducing data movement becomes less prominent due to the compute capability gap between the **OASIS-A** and the **OASIS-FE**.

To validate the results presented in Figure 7, we measured both the inter-layer data traffic and the size of the result data transferred to the compute layer. Across all queries, a consistent trend emerges: both **COS** and **OASIS** substantially reduce the volume of result data compared to the **Baseline**. For instance, in Q2, the result size is reduced from 13.18GB in the **Baseline** to 52.89MB (Arrow IPC) in **OASIS** and 29.08 MB (CSV) in **COS**, with the smaller size in **COS** attributed to the higher compression ratio of CSV format. For inter-layer traffic, **COS** transfers the entire dataset from the **OASIS-A**, while **OASIS** performs early filtering and reduces inter-layer transfer to 53MB. This demonstrates that **OASIS** minimizes internal data movement, leading to faster execution for scalar queries with lightweight predicates. Across all workloads, **Pred** is slightly slower than the **Baseline**, primarily due to

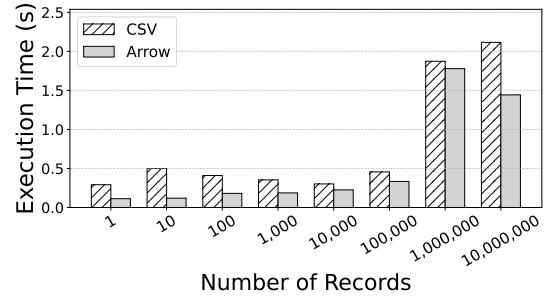


Fig. 8: Comparison of input parsing times of client-side Spark for CSV and Arrow formats across different record sizes.

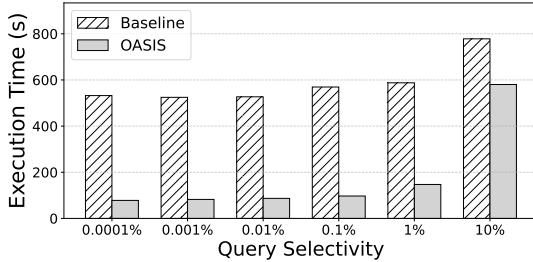
the overhead of scanning Parquet metadata, with no records being filtered out in the target datasets.

2) *Queries involving Array-based Conditions:* Queries Q1–Q3 use scalar-based conditions, whereas Q4 involves an array-based condition. In this section, we evaluated Q4, derived from the HEP benchmark [65], to assess the array-aware processing capabilities of **OASIS**. Q4 involves array-based *filter* and *project* operations. **COS**, modeled after vanilla SkyhookDM, supports only array-based *filter* and simple column-level *project*.

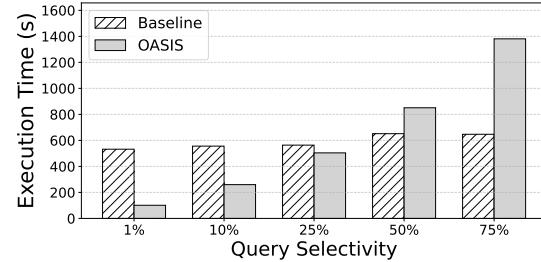
Figure 7 illustrates the Q4 execution time of four configurations. **OASIS** completed the query in 87.203 seconds, 24.6% faster than **COS** and 64.0% faster than the **Baseline**, by offloading both the *filter* and *project* with array-based conditions to the storage layer. In contrast, **COS** offloads only the initial *filter* and must transfer intermediate results to the compute node for array-based *project*, even though it can execute subsequent *filter* operations. This failure to offload *project* results in additional data movement. Predicate Push-down evaluates only the scalar predicate in storage, showing better performance than **Baseline**. These results highlight that **OASIS**'s array-aware offloading reduces data movement and accelerates query execution.

#### D. Impact of Output Format on System Efficiency

Figure 8 compares the data loading performance of Arrow and CSV formats when ingesting the output of Q1 into the Spark cluster for RQ#3. Arrow consistently outperforms CSV across all record sizes in terms of load time. The observed decrease in execution time from 1,000,000 to 10,000,000 records is due to Spark increasing the number of partitions from 112 to 1,000, thereby enhancing parallelism and improving



(a) Q1 with aggregation, selectivity: 0.0001%–10%.



(b) Q1 without aggregation, selectivity: 1%–75%.

Fig. 9: Execution time comparison between the baseline and OASIS for the Q1 query under varying selectivity.

overall data ingestion throughput. Since Arrow enables more efficient in-memory loading compared to CSV regardless of data size, these results suggest that Arrow is advantageous not only as a final output format but also as an intermediate data representation during multi-layer query execution.

#### E. Performance Behavior of OASIS Across Diverse Selectivity

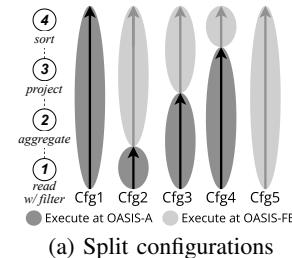
Figure 9 (a) presents the performance comparison between OASIS and the **Baseline** as the selectivity of Q1 varies. In contrast, (b) shows the results for a modified version of Q1 where the Group By (aggregation) operator is removed, with selectivity similarly adjusted.

In (a), OASIS consistently outperforms the **Baseline** even as selectivity increases. This is because the Group By operation limits the number of output rows based on the number of aggregation groups, preventing the output size from growing rapidly even as input data increases. In fact, for Q1, the maximum achievable selectivity was approximately 13%, constrained by the nature of the Group By. This suggests that aggregation operations impose a natural upper bound on output size, making them well-suited for query offloading.

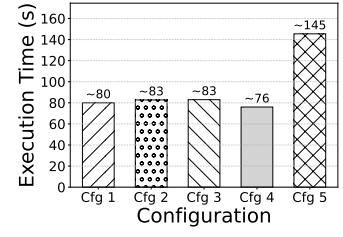
On the other hand, (b) explores the case where the Group By operation is removed, allowing selectivity to increase up to approximately 75%. In this setting, the **Baseline** begins to outperform OASIS when selectivity exceeds around 25%. This indicates that when heavy operations such as sorting follow the filtering step, traditional cluster-based processing may become more efficient than storage-side offloading as the amount of data grows. These results highlight the need for dynamic offloading decisions based on both the query’s operator characteristics and its selectivity.

#### F. Effectiveness of SODA Decomposition

Figure 10 illustrates the evaluation of various split strategies to assess the effectiveness of SODA. Among the queries, Q1 contains the largest number of operators, while the others are relatively simple and do not undergo plan splitting. Therefore, Q1 is selected as the representative case to validate the behavior of SODA. Q1’s query plan consists of four sequential stages: (1) *read w/ filter*, (2) *aggregate*, (3) *project*, and (4) *sort* (Figure 10(a)). We evaluate five different configurations within OASIS, where the Substrait Decomposer statically distributes operators between the OASIS-FE and OASIS-A without applying SODA. Among all configurations, SODA



(a) Split configurations



(b) Execution time comparison

Fig. 10: (a) Description of decomposition configurations and (b) execution time comparison across five configurations. Each bar uses a distinct hatch pattern for visual distinction.

selected cfg4, which offloads *read w/ filter*, *aggregate*, and *project* to the OASIS-A, while executing only *sort* at the OASIS-FE. This configuration achieved the best runtime of 76 seconds, yielding a 45% reduction compared to the OASIS-FE-only setup that logically corresponds to the computation model of conventional COS systems. Configurations that offloaded only *filter* or *filter* with *aggregate* showed runtimes around 83 seconds, as the reduced data volume did not fully offset the remaining compute overhead at the OASIS-FE.

In our experiments, SODA introduces minimal overhead, with an average of just 126ms for selectivity estimation and 1,810ms for Substrait-based plan decomposition.

These results demonstrate that pushing low-cost, high-reduction operators closer to data, while reserving compute-heavy ones like *sort* for the OASIS-FE, yields better performance. SODA can be further improved by incorporating operator-level compute cost into its decision model.

## VI. CONCLUSION

In this work, we presented OASIS, a computation-enabled object storage (COS) system designed for high-throughput scientific analytics workloads. OASIS overcomes key limitations of existing COS systems by enabling fine-grained operator offloading, supporting complex array-aware expressions, and dynamically optimizing query execution across storage layers. Leveraging Substrait-based plan decomposition and dynamic execution path optimization, OASIS identifies optimal split points to minimize data movement while utilizing in-storage compute. Real-world HPC query evaluations show that OASIS not only reduces execution time but also significantly improves resource efficiency across the storage stack.

## ACKNOWLEDGMENTS

This work was supported by SK hynix Inc.

## REFERENCES

- [1] J. Blomer, "A quantitative review of data formats for hep analyses," in *Journal of Physics: Conference Series*, vol. 1085, p. 032020, IOP Publishing, 2018.
- [2] J. Gray, D. T. Liu, M. Nieto-Santisteban, A. Szalay, D. J. DeWitt, and G. Heber, "Scientific data management in the coming decade," *Acm Sigmod Record*, vol. 34, no. 4, pp. 34–41, 2005.
- [3] J. Liu, C. Maltzahn, M. L. Curry, and C. Ulmer, "Processing particle data flows with smartnics," in *2022 IEEE High Performance Extreme Computing Conference (HPEC)*, pp. 1–8, IEEE, 2022.
- [4] A. Da Ronch, M. Ghoreyshi, D. Vallespin, K. Badcock, Z. Mengmeng, J. Opplestrup, and A. Rizzi, "A framework for constrained control allocation using cfd-based tabular data," in *49th AIAA Aerospace Sciences Meeting including the New Horizons Forum and Aerospace Exposition*, p. 925, 2011.
- [5] H.-T. Chiu, J. Chou, V. Vishwanath, and K. Wu, "In-memory query system for scientific databaseis," in *2015 IEEE 21st International Conference on Parallel and Distributed Systems (ICPADS)*, pp. 362–371, IEEE, 2015.
- [6] Apache Foundation, "Apache Spark: Unified engine for large-scale data analytics," 2024.
- [7] S. Blanas, K. Wu, S. Byna, B. Dong, and A. Shoshani, "Parallel data analysis directly on scientific file formats," in *2014 ACM SIGMOD International Conference on Management of Data, SIGMOD '14*, (New York, NY, USA), p. 385–396, Association for Computing Machinery, 2014.
- [8] J. Gu, S. Klasky, N. Podhorszki, J. Qiang, and K. Wu, "Querying large scientific data sets with adaptable IO system ADIOS," in *Supercomputing Frontiers* (R. Yokota and W. Wu, eds.), (Cham), pp. 51–69, Springer International Publishing, 2018.
- [9] Gary Grider, "LANL's Journey Toward Computational Storage." <https://snia.org/educational-library/lanls-journey-toward-computational-storage-2024>, 2024. Presented at the SNIA Storage Developer Conference (SDC), 2024.
- [10] O. R. N. Laboratory, "ORNL Frontier." <https://www.olcf.ornl.gov/frontier/>, 2022.
- [11] CERN, "High-luminosity large hadron collider (hl-lhc)." <https://hilumilhc.web.cern.ch/>, 2025. Accessed: 2025-05-12.
- [12] H. Xing, S. Floratos, S. Blanas, S. Byna, M. Prabhat, K. Wu, and P. Brown, "Arraybridge: Interweaving declarative array processing in scidb with imperative hdf5-based programs," in *2018 IEEE 34th International Conference on Data Engineering (ICDE)*, pp. 977–988, IEEE, 2018.
- [13] A. Giannakou, D. Hazen, B. Enders, L. Ramakrishnan, and N. J. Wright, "Understanding data movement patterns in hpc: A nerse case study," in *SC24: International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–17, 2024.
- [14] H. Liang, Z. Zhang, C. Hu, Y. Gong, and D. Cheng, "A survey on spatio-temporal big data analytics ecosystem: Resource management, processing platform, and applications," *IEEE Transactions on Big Data*, 2023.
- [15] P. Cicotti, S. Oral, G. Kestor, R. Gioiosa, S. Strande, M. Taufer, J. H. Rogers, H. Abbasi, J. Hill, and L. Carrington, *Data Movement in Data-Intensive High Performance Computing*, pp. 31–59. Cham: Springer International Publishing, 2016.
- [16] H. Li, "Status and planning of high energy physics data storage system." Presentation at the 2023 International Workshop on the High Energy Circular Electron Positron Collider (CEPC), October 2023. Institute of High Energy Physics (IHEP), Chinese Academy of Sciences.
- [17] N. Smith, B. Jayatilaka, D. Mason, O. Gutsche, A. Peisker, R. Illingworth, and C. Jones, "A ceph s3 object data store for hep," 2023.
- [18] MinIO, Inc., "Minio: High performance object storage." <https://min.io/>.
- [19] S. A. Weil, S. A. Brandt, E. L. Miller, D. D. E. Long, and C. Maltzahn, "Ceph: A scalable, high-performance distributed file system," in *7th Symposium on Operating Systems Design and Implementation, OSDI '06*, (USA), p. 307–320, USENIX Association, 2006.
- [20] Randall Hunt, "S3 Select and Glacier Select – Retrieving Subsets of Objects," Nov. 29, 2017.
- [21] J. LeFevre and C. Maltzahn, "Skyhookdm: Data processing in ceph with programmable storage," *USENIX login;*, vol. 45, no. 2, 2020.
- [22] MinIO, Inc., "mc sql command — run sql queries on object data." <https://min.io/docs/minio/linux/reference/minio-mc/mc-sql.html>. Accessed: 2025-05-12.
- [23] Ceph Project, "S3 select support in rados gateway." <https://docs.ceph.com/en/latest/radosgw/s3select/>. Accessed: 2025-05-12.
- [24] H. Sim, Y. Kim, S. S. Vazhkudai, D. Tiwari, A. Anwar, A. R. Butt, and L. Ramakrishnan, "AnalyzeThis: an analysis workflow-aware storage system," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2015.
- [25] H. Sim, Y. Kim, S. S. Vazhkudai, G. R. Vallée, S.-H. Lim, and A. R. Butt, "Tagit: an integrated indexing and search service for file systems," in *International Conference for High Performance Computing, Networking, Storage and Analysis*, pp. 1–12, 2017.
- [26] Apache, "Apache Arrow: A cross-language development platform for in-memory analytics." <https://arrow.apache.org/>, 2024.
- [27] D. Foundation, "DuckDB." <https://duckdb.org/>, 2024.
- [28] L. A. N. Laboratory, "OCS Laghos Sample Dataset." <https://github.com/laml-ocs/laghos-sample-dataset>, 2024.
- [29] S. Sehrish, J. Kowalkowski, and M. Paterno, "Spark and hpc for high energy physics data analyses," in *2017 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pp. 1048–1057, IEEE, 2017.
- [30] J. Liu, E. Racah, Q. Koziol, R. S. Canon, A. Gittens, L. Gerhardt, S. Byna, and M. F. Ringenburg, "H5spark: bridging the i/o gap between spark and scientific data formats on hpc systems," *Cray user group*, 2016.
- [31] Apache, "Parquet." <https://parquet.apache.org/>, 2024.
- [32] G. Gavalian, "High-performance data format for scientific data storage and analysis," *arXiv preprint arXiv:2501.07666*, 2025.
- [33] J. Planas, F. Delalondre, and F. Schürmann, "Accelerating data analysis in simulation neuroscience with big data technologies," in *Computational Science–ICCS 2018: 18th International Conference, Wuxi, China, June 11–13, 2018, Proceedings, Part I 18*, pp. 363–377, Springer, 2018.
- [34] L. Canali, "Apache spark for high energy physics," 2024. Benchmark notebooks demonstrating the use of Apache Spark for High Energy Physics data analysis.
- [35] B. Dong, S. Byna, and K. Wu, "Parallel query evaluation as a scientific data service," in *2014 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 194–202, IEEE, 2014.
- [36] H. Asaadi, D. Khaldi, and B. Chapman, "A comparative survey of the hpc and big data paradigms: Analysis and experiments," in *2016 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 423–432, IEEE, 2016.
- [37] Q. Zheng, "Toward open object-based computational storage for analysis query pushdown." PDSW Work-in-Progress, 2023.
- [38] D. Manno, "Improving storage systems for simulation science with computational storage." Compute+Memory+Storage Summit, 2023.
- [39] Q. Zheng, "Kinetic campaign: Speeding scientific data analytics with computational storage drives." Presented at SDC, 2022.
- [40] L. A. N. L. O. C. S. (LANL-OCS), "Laghos sample dataset," 2024. Sample dataset generated by the Laghos simulation application for system prototyping and benchmarking.
- [41] Gary Grider, "Leveraging Computational Storage for Simulation Science Storage System Design." Presented at the SNIA Storage Developer Conference (SDC), 2023, 2023.
- [42] I. Park, Q. Zheng, D. Manno, S. Yang, J. Lee, D. Bonnie, B. Settlemyer, Y. Kim, W. Chung, and G. Grider, "Kv-csd: A hardware-accelerated key-value store for data-intensive applications," in *2023 IEEE International Conference on Cluster Computing (CLUSTER)*, pp. 132–144, 2023.
- [43] K. Duwe and M. Kuhn, "Using ceph's bluestore as object storage in hpc storage framework," CHEOPS '21, (New York, NY, USA), Association for Computing Machinery, 2021.
- [44] Amazon Web Services, "Amazon S3, Object storage built to retrieve any amount of data from anywhere." <https://aws.amazon.com/s3/>, 2024.
- [45] R. C. for Computational Science, "Fugaku aws s3 service guide," 2024. Documentation for utilizing AWS S3-compatible storage on the Fugaku supercomputer.
- [46] IBM Corporation, "Ibm storage ceph s3 object deep dive," May 2024. Technical white paper detailing IBM Storage Ceph's S3 object storage features and deployment strategies.
- [47] NetApp, "S3 and analytics: Taming your storage costs," November 2018. White Paper WP-7289.

- [48] A. Montana, Y. Xue, J. LeFevre, C. Maltzahn, J. Stuart, P. Kufeldt, and P. Alvaro, “A moveable beast: Partitioning data and compute for computational storage,” 2023.
- [49] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker, “Pushdowndb: Accelerating a dbms using s3 computation,” in *2020 IEEE 36th International Conference on Data Engineering (ICDE)*, pp. 1802–1805, 2020.
- [50] L. A. N. Laboratory, “Deep water impact dataset (la-ur-17-21595),” 2024. Sample dataset generated by converting an existing LANL deep water impact simulation dataset for system prototyping and benchmarking.
- [51] L. A. N. Laboratory, “C2 vpic sample dataset,” 2022. Sample dataset generated by running the open-source VPIC particle simulation code for local C2 development and testing.
- [52] A. Huebl, F. Poeschel, F. Koller, J. Gu, M. Bussmann, J.-L. Vay, and K. Wu, “openpmd-api: C++ & python api for scientific i/o with openpmd,” 2018. Version 0.17.0-dev.
- [53] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yigitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, “Presto: SQL on Everything,” in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019.
- [54] Amazon Web Services, “Querying data in place with amazon s3 select,” 2024. Amazon Simple Storage Service (S3) User Guide.
- [55] Supermicro, “Supermicro’s Petascale All-Flash GEN 5 JBOF Storage Solution with NVIDIA BlueField-3 DPU.” <https://www.supermicro.com/en/products/jbof>, 2024.
- [56] DDN, “DDN Appliance Combines AI Storage and NVIDIA BlueField-3 DPUs for Enhanced Full-Stack Data Center and Cloud Efficiency.” <https://www.ddn.com/press-releases/ddn-appliance-combines-ai-storage-and-nvidia-bluefield-bf-3-dpus/>, 2024.
- [57] S. Project, “Substrait.” <https://substrait.io/>, 2024.
- [58] NVM Express Inc., “NVM Express Specification.” <https://nvmeexpress.org/developers/nvme-specification/>, 2011.
- [59] P. Holanda, “DuckDB ADBC – Zero-Copy Data Transfer via Arrow Database Connectivity,” Aug. 2023. Accessed June 12, 2025.
- [60] P. Pedreira, O. Erling, M. Basmanova, K. Wilfong, L. Sakka, K. Pai, W. He, and B. Chattopadhyay, “Velox: meta’s unified execution engine,” *VLDB Endowment*, vol. 15, p. 3372–3384, Aug 2022.
- [61] Z. Yang, J. R. Harris, B. Walker, D. Verkamp, C.-p. Liu, C. Chang, G. Cao, J. Stern, V. Verma, and L. E. Paul, “SPDK: A development kit to build high performance storage applications,” in *IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.
- [62] Versity, “Versity Gateway.” <https://www.versity.com/products/versitygw/>, 2024.
- [63] J. M. P. F. J. S. K. C. T. G. R. G. D. H. R. G. D. A. T. L. Turton, “Visualization and analysis of threats from asteroid ocean impacts,” *Technical report*, 2016.
- [64] CMS Collaboration, “SingleMu primary dataset in AOD format from Run 2012B (22 Jan 2013 re-reconstruction).” CERN Open Data Portal, 2017. Dataset record 6021.
- [65] D. Graur, I. Müller, M. Proffitt, G. Fourny, G. T. Watts, and G. Alonso, “Evaluating query languages and systems for high-energy physics data,” *Proc. VLDB Endow.*, vol. 15, p. 154–168, Oct. 2021.