Integrating Distributed SQL Query Engines with Object-Based Computational Storage

Junghyun Ryu
Sogang University
Seoul, Republic of Korea
jhryu@sogang.ac.kr

Soon Hwang
Sogang University
Seoul, Republic of Korea
soonhw@sogang.ac.kr

Junhyeok Park
Sogang University
Seoul, Republic of Korea
junttang@sogang.ac.kr

Seonghoon Ahn
Sogang University
Seoul, Republic of Korea
ok10p@sogang.ac.kr

JeoungAhn Park
Memory Systems Research
SK hynix Inc.
jungahn.park@sk.com

Jeongjin Lee
Memory Systems Research
SK hynix Inc.
jeongjin0.lee@sk.com

Jinna Yang
Memory Systems Research
SK hynix Inc.
jinna.yang@sk.com

Soonyeal Yang
Memory Systems Research
SK hynix Inc.
soonyeal.yang@sk.com

Jungki Noh
Memory Systems Research
SK hynix Inc.
jungki.noh@sk.com

Qing Zheng
Los Alamos National Laboratory
Los Alamos, NM, USA
qzheng@lanl.gov

Woosuk Chung
Memory Systems Research
SK hynix Inc.
woosuk.chung@sk.com

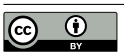
Hoshik Kim
Memory Systems Research
SK hynix Inc.
hoshik.kim@sk.com

Youngjae Kim*
Sogang University
Seoul, Republic of Korea
youkim@sogang.ac.kr

Abstract

Existing object storage systems like AWS S3 and MinIO offer only limited in-storage compute capabilities, typically restricted to simple SQL WHERE-clause filtering. Consequently, high-impact operators such as aggregation and top-N are still executed entirely at the compute layer. Recent advances in Object-based Computational Storage (OCS) enable these complex operators to run natively within storage, creating opportunities for substantial reductions in data movement and query time. To demonstrate these benefits in distributed SQL engines, we used Presto as a case study and developed the Presto-OCS connector, which analyzes execution plans to identify pushdown-eligible operators and offloads them to OCS for efficient in-storage execution. Evaluations with real-world HPC analytics queries and the TPC-H benchmark show that our approach achieves up to 4.07× speedup and 99% data movement reduction compared to filter-only pushdown. When combined with compression techniques, our approach delivers 1.39× speedup over compressed filter-only pushdown, demonstrating that advanced query pushdown complements existing optimizations.

*Y. Kim is the corresponding author.



SC Workshops '25, St Louis, MO, USA

ACM ISBN 979-8-4007-1871-7/2025/11
https://doi.org/10.1145/3731599.3767371

CCS Concepts

• **Information systems** → **Information storage systems**; **Data management systems**; **Storage architectures**.

Keywords

Computational Storage, Object Storage, SQL Query Engines, Big Data Analytics

ACM Reference Format:
Junghyun Ryu, Soon Hwang, Junhyeok Park, Seonghoon Ahn, JeoungAhn Park, Jeongjin Lee, Jinna Yang, Soonyeal Yang, Jungki Noh, Qing Zheng, Woosuk Chung, Hoshik Kim, and Youngjae Kim. 2025. Integrating Distributed SQL Query Engines with Object-Based Computational Storage. In *Workshops of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC Workshops '25), November 16–21, 2025, St Louis, MO, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3731599.3767371

1 Introduction

Disaggregated architectures have become dominant in both cloud and High-Performance Computing (HPC) environments, emerging as the standard for large-scale data analytics systems by separating the compute and storage layers [2, 33, 43]. While this separation enables independent scalability and simplifies system management, it also creates a critical performance challenge: the network bottleneck [15, 36, 49]. In practice, entire files are often transferred across the network even when queries access only a small fraction of the

data, making excessive data movement a major contributor to query latency and overall execution time [20, 37, 51]. This inefficiency has emerged as one of the most pressing obstacles to scaling modern analytics systems.

To address this growing bottleneck, modern object storage platforms such as Amazon S3 [3] and MinIO [34] have introduced query pushdown techniques that offload part of the SQL processing to the storage layer. By executing simple operators such as WHERE-clause filtering or column projections closer to the data, these systems (e.g., S3 Select [40] and MinIO Select [35]) can reduce the volume of data transferred over the network. As a result, they have become widely adopted backends for cloud-based and HPC OLAP systems, valued for their scalability, cost-effectiveness, and compatibility with disaggregated architectures.

However, the scope of such pushdown remains narrow, as current object storage systems lack support for diverse query pushdown capabilities. Consequently, more complex queries including aggregation (GROUP BY) or top-N (ORDER BY + LIMIT) operators must still be executed on the client side, leaving the fundamental data movement challenge only partially addressed. Thus, the narrow scope of query pushdown in current object storage systems highlights the need for a more powerful solution.

Recognizing this limitation, SK hynix recently proposed an enhanced design called Object-based Computational Storage (OCS) [23], which extends the boundaries of in-storage query processing. Unlike S3 Select or MinIO Select, which are restricted to simple filtering and column projection, OCS embeds an SQL query execution engine directly inside the storage system. This in-place processing enables a much broader set of operators, including aggregation (GROUP BY) and top-N (ORDER BY + LIMIT), to be executed close to the data, substantially reducing network traffic and query latency. To achieve interoperability and efficiency, OCS adopts Substrait Intermediate Representation (IR) [45] for standardized query plan exchange and Apache Arrow [9] for lightweight, high-performance columnar result serialization. With Substrait ensuring compatibility across query engines and Arrow providing efficient data transfer, their design opens up new opportunities to bridge the long-standing gap between disaggregated storage and analytics engines, addressing the fundamental bottleneck of data movement that prior approaches could not overcome.

Despite OCS's ability to pushdown complex queries, there has been no attempt to integrate it with distributed SQL query engines such as Presto [41], creating a critical gap that prevents its widespread adoption. Presto is one of the most widely used distributed SQL query engines for interactive analytics across heterogeneous data sources, with a modular architecture that allows pluggable storage connectors via its Service Provider Interface (SPI) [39]. Today, object storage systems are typically accessed in Presto through the Hive connector [38], which provides a unified API across S3-compatible backends. While this abstraction simplifies integration and ensures compatibility, it also imposes a fundamental limitation: pushdown is restricted to basic filters and column projections, leaving operators such as aggregation (GROUP BY) and top-N (ORDER BY + LIMIT) entirely unoffloaded.

This limitation exposes the central problem: without a dedicated connector, OCS's advanced pushdown capabilities cannot be leveraged within Presto's execution framework. Addressing this gap introduces several key challenges:

- **Limited accessibility of OCS features:** Although OCS supports complex operations such as aggregation and top-N, these capabilities remain inaccessible through the Hive connector, which is bound to the limited S3 Select API.
- **Preserving Presto's modularity:** The integration must adhere to Presto's design philosophy of modularity, ensuring that OCS-specific optimizations do not compromise compatibility with existing query pipelines.
- **Query translation overhead:** The connector must efficiently translate Presto's internal query representations into Substrait IR, which OCS requires for execution.

To address these challenges, this paper proposes the design of a Presto-OCS Connector that fully exploits the advanced pushdown capabilities of OCS while preserving Presto's modular architecture. Our approach builds on Presto's Connector SPI framework, which is designed to support storage-specific optimizations without breaking compatibility with the standard query execution pipeline. In this design, the Presto-OCS connector extends Presto's local optimizer, which is a component that allows storage connectors to apply their own optimizations after the main query planning phase. This extension enables the connector to identify pushdown-eligible operators during query optimization, determining which operators can be efficiently executed at the storage layer rather than in the compute layer. By doing so, we demonstrate that advanced query pushdown can be seamlessly integrated into Presto's modular architecture. These operators are then translated into Substrait IR and dispatched to OCS via gRPC [19], enabling in-storage execution of complex queries that conventional object storage systems cannot handle. We implement our design on Presto version 0.286, a widely used distributed SQL query engine, and integrate the proposed OCS connector into its execution framework.

Evaluation on HPC and business OLAP workloads demonstrates the significant benefits of the proposed Presto-OCS connector. For the Laghos dataset [29], a fluid dynamics simulation from Los Alamos National Laboratory, our approach achieves a 2.25× speedup (from 1,015 to 450 seconds) with a 99.99% reduction in data movement (from 5.1GB to 0.5MB) compared to filter-only pushdown. The Deep Water Impact dataset [22] shows a 1.32× speedup (from 441 to 335 seconds) with 99.98% less data transferred (from 5.37GB to 1MB). For TPC-H Query 1, we observe a 4.07× speedup (from 9 to 2.21 seconds) with a 99.7% reduction in data movement (from 192MB to 0.5MB). Additionally, experiments with various compression algorithms reveal that our approach achieves 1.36×–1.39× speedup over compressed filter-only pushdown across all compression methods (Snappy: 1.37×, GZip: 1.39×, Zstd: 1.36×), confirming that advanced operator pushdown and compression are complementary techniques that together maximize performance in computational storage systems.

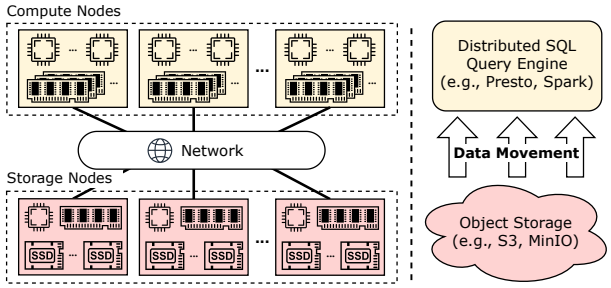


Figure 1: Overview of the disaggregated architecture for distributed SQL query processing with object storage

2 Background

2.1 Excessive Data Movement in Analytical Workloads

Modern large-scale data processing analytics systems are now increasingly built on disaggregated architectures that physically separate compute and storage nodes [2, 33, 41, 43]. This separation enables independent scalability and simplifies system maintenance but also incurs unavoidable network data movement during data analytics, as all data must be retrieved from remote storage. Figure 1 illustrates the disaggregated architecture, where compute nodes must retrieve all data from physically separated storage nodes through network connections. Compared to local I/O, remote access introduces higher latency and bandwidth constraints. This inefficiency is particularly pronounced in high-selectivity analytical workloads such as Online Analytical Processing (OLAP), where queries typically access only selected columns or a subset of rows. In high-selectivity workloads, only a small portion of the dataset is needed, yet entire files may still be transferred, making data movement the dominant bottleneck [33]. For instance, scientific data analytical workloads used in High-Performance Computing (HPC) typically access only a small fraction of the dataset, yet still incur significant overhead from transferring entire files [20, 32, 37, 51]. Large-scale commercial data analytics systems also follow this trend. For example, more than half of all queries in Google's analytical workloads return less than 1% of total data [33].

2.2 Object Storage Systems and Their Computation Support

Object storage is a storage architecture that manages data as discrete objects identified by globally unique IDs. Well-known object storage systems such as AWS S3 [3] and MinIO [34] adopt flat bucket-object namespaces and separate metadata from data. Unlike hierarchical file systems with directory structures, object storage eliminates POSIX constraints such as file locking, improving scalability and parallel data access. Additionally, object storage systems can store columns as independent objects, facilitating distributed placement across storage tiers based on data access patterns in tiered Storage Systems [42].

The stateless nature of object storage enables horizontal scaling without performance degradation, allowing multiple compute nodes to read data in parallel without metadata overhead. From a columnar processing perspective, object storage naturally aligns with column-oriented data formats like Parquet [7] and ORC [6], where each column chunk or row group can be stored as a separate object [42]. This enables selective column retrieval without reading entire datasets, significantly reducing I/O overhead for analytical queries that typically access only a subset of columns. The object-level granularity also facilitates efficient predicate pushdown and column pruning, as compute nodes can request only the specific column objects required for query execution. These features improve overall storage efficiency and I/O performance, which is why object storage has become the prevalent storage layer for data analytics workloads in cloud and has been increasingly adopted in HPC environments processing large-scale scientific datasets [14, 16].

To further reduce data movement when processing SQL-based analytics workloads on object storage systems, query pushdown techniques have emerged as an optimization strategy [15, 49]. Query pushdown offloads certain SQL operators directly to the storage layer, allowing data reduction to occur before network transfer. Operators such as filter, projection, aggregation and top-N are particularly suitable for pushdown, as they require modest computation but significantly reduce data volume [50]. Representative examples include S3 Select [40] and MinIO Select [35], which enable storage-side filtering through SELECT (column projection) and WHERE (filter) clauses for CSV, JSON, and Parquet formats. By executing these operators at the storage side, only filtered rows and columns are transmitted to compute nodes, reducing data movement. As shown in Figure 2 (a), traditional object storage systems execute all SQL operators at the compute node, requiring full dataset or column chunk transfer. Figure 2 (b) illustrates how S3 Select and MinIO Select reduce this overhead by performing filtering and column projection at the storage layer, transmitting only the processed results to compute nodes.

However, their pushdown capabilities remain limited. AWS S3 and MinIO offer limited query pushdown methods which only support SELECT and WHERE clauses for storage-side filtering, with results returned in traditional row-oriented formats (CSV, JSON) that lack the columnar processing benefits of modern formats like Apache Arrow. Data-reducing operators such as aggregation and top-N, which can significantly reduce result sizes, must still be handled by compute nodes, missing substantial opportunities for minimizing data movement if executed at the storage layer. Moreover, S3 Select lacks support for double-precision floating-point values, making it unsuitable for scientific domains that require high numeric precision. MinIO Select, built on the S3 Select API, inherits the same limitations.

2.3 Towards Computational Object Storage

To overcome the mentioned limitations of conventional object storage, SK Hynix has introduced Object-based Computational Storage (OCS) [23]. OCS integrates an embedded SQL engine capable of executing a broad range of SQL operators such as column project (SELECT), expression project (SELECT), filter (WHERE), aggregation (GROUP BY), sort (ORDER BY), and limit/top-N (LIMIT, ORDER BY + LIMIT) directly at the storage level, with support for complex data types including double-precision floating-point numbers.

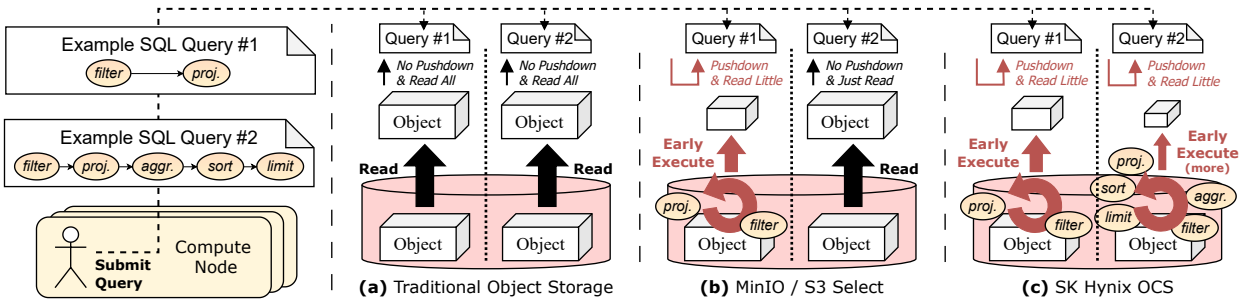


Figure 2: (a) In traditional object storage systems, all SQL operators are executed at the compute node, requiring the transfer of the entire row group regardless of query selectivity. (b) S3 Select and MinIO Select add limited compute capabilities (filtering and column projection) at the storage layer to reduce data movement by transferring only the processed results. (c) SK Hynix OCS enables early execution of a wider range of SQL operators—such as aggregation and top-N—directly within the storage layer, expanding the scope of pushdown.

As shown in Figure 2 (c), OCS significantly extends pushdown capabilities to include aggregation, sorting, and result limiting operators. Since operators such as aggregation and limiting have significant potential to achieve even greater data movement reduction compared to simple filtering at the storage layer, this capability extends in-storage processing that further minimizes data movement and accelerates query performance for analytical queries in HPC environments. To represent query plans, OCS adopts the Substrait Intermediate Representation (IR) [45], which promotes interoperability across query engines. For result transmission, Apache Arrow [9] is used as a columnar in-memory format optimized for analytical workloads.

2.4 Limitations of the Hive Connector for Query Pushdown

Distributed SQL engines such as Presto [41] and Spark [5] rely on the Hive connector [38] as the primary interface to object storage. The Hive connector has long served as the de facto standard for accessing S3-compatible object storage in distributed SQL engines. It provides a unified interface to diverse S3-compatible object storage backends and leverages the Hive Metastore [4] to deliver catalog services including schemas and tables required for query planning and optimization. In addition, the Hive connector exposes storage-supported pushdown APIs such as S3 Select, which has become standard for query pushdown and is adopted for filter and column project pushdown in object storage systems including AWS S3, MinIO. This ensures users can apply pushdown for common operators across diverse object storage systems.

However, query pushdown through the Hive connector remains limited to the standard pushdown API, which limits potential performance gains. Although modern storage systems such as OCS support more complex operators including aggregate and top-N for pushdown, the Hive connector exposes only basic operators and thus cannot exploit these extended capabilities. This implies that the OCS-specific connector is required to fully utilize the extended storage-specific capabilities.

3 Design of Presto-OCS Connector

We design and implement Presto-OCS connector, which extends Presto's Connector Service Provider Interface (SPI) [39] to enable advanced operator pushdown into object storage. At a high level, (i) the connector intercepts query operators during optimization, (ii) detects pushdown-eligible operators, and (iii) translates them into Substrait, an emerging standard for cross-system query plan representation, for in-storage execution.

3.1 Presto's Connector-Based Architecture

Presto is a distributed ANSI SQL-compatible query engine originally developed by Meta to provide a unified SQL interface over heterogeneous data sources, following an "SQL-on-Anything" architecture [41]. Built on a disaggregated architecture that separates the compute engine from storage, Presto supports a flexible connector-based interface, enabling customized access mechanisms for various storage systems such as AWS S3 [3], HDFS [8], and others. This flexible execution model enables direct SQL querying over diverse backends and has led to widespread adoption by large-scale enterprises such as Meta and Uber [30]. More recently, there have been efforts to apply Presto as a data analytics application in HPC environments [11, 48].

3.2 Presto's Query Planning Workflow

Presto adopts a coordinator-worker architecture, where the coordinator is responsible for planning and scheduling query execution across workers, and the workers execute the assigned query fragments. To process an SQL query, the coordinator performs the following planning steps, as illustrated in Figure 3:

(1) **SQL Parsing:** The input query is parsed into an abstract syntax tree (AST).
(2) **Analysis and Logical Plan Construction:** The AST is semantically analyzed (e.g., type checking and schema resolution) and converted into a logical plan tree composed of nodes such as *TableScanNode*, *FilterNode*, and *AggregationNode*, each representing an operator in the query pipeline.
(3) **Logical Optimization (Global Optimizer):** Rule-based transformations (e.g., join reordering and projection pruning) are applied.

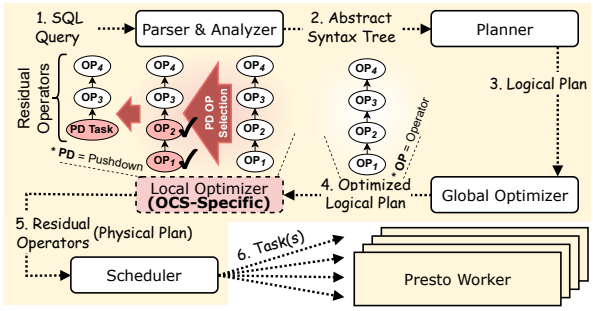


Figure 3: Query planning sequence in Presto's coordinator.

(4) **Connector-Specific Optimization (Local Optimizer):** Through the *ConnectorPlanOptimizer* interface, each connector can inspect and rewrite plan nodes based on backend-specific logic. Our system detects pushdown-eligible operators at this phase.
(5) **Physical Planning:** The logical plan is translated into a physical plan, including execution strategies and parallelism configuration.
(6) **Split Generation and Scheduling:** The *TableScan* operator is partitioned into splits for parallel execution across available workers. Split generation considers data distribution, worker availability, and parallelism configuration to optimize workload balance. The scheduler distributes these splits to workers and also assigns subsequent tasks in the execution pipeline, ensuring each worker knows its complete sequence of operators for processing the query.

3.3 Unlocking the Potential of Computational Object Storage

Integrating OCS into Presto requires preserving Presto's modular architecture. As mentioned in Section 2.4, simply replacing the storage backend with OCS via the Hive connector is insufficient, since existing connectors in distributed SQL engines are primarily designed as I/O adapters with limited pushdown functionality. The challenge, therefore, is to design a connector that exploits OCS-specific pushdown features while operating within Presto's execution pipeline. This requires careful coordination between Presto's planning and execution phases, with the connector identifying pushdown-eligible operators during planning and correctly executing them at storage during runtime. To fully utilize OCS's computational capabilities, we developed a separate connector that extends beyond simple I/O translation to directly expose near-data processing capabilities. This design decouples OCS access from the Hive connector, preserving the standard Hive interface for conventional object stores while providing an optimized path for OCS with advanced query pushdown support.

3.4 Presto-OCS Connector

The Presto-OCS connector is implemented via Presto's SPI [39] without requiring any modification to the core execution pipeline. The connector focuses on the pushdown of data-reducing operators such as filter, aggregation, and top-N to minimize unnecessary data movement. To achieve this, it employs key components including

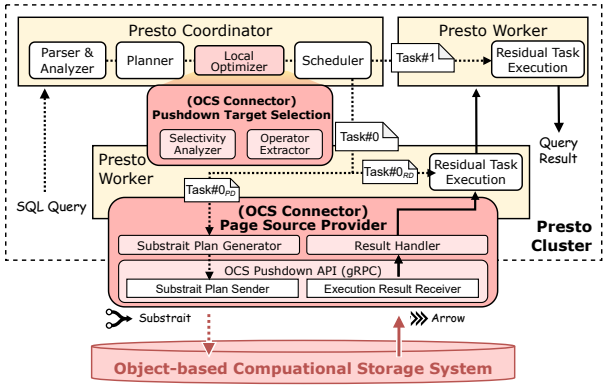


Figure 4: Execution flow of the Presto-OCS integrated architecture.

the *Selectivity Analyzer* for evaluating operator data reduction potential, the *Operator Extractor* for capturing pushdown candidates, and the *PageSourceProvider* for managing storage communication. These components work together through an optimized pipeline, translating operators into Substrait IR [45] and offloading them to OCS through gRPC [19] for low-latency execution. Figure 4 illustrates how these components interact within the system. The Presto cluster consists of a coordinator responsible for query planning and scheduling, and multiple workers that execute the distributed query fragments. The connector's pushdown logic operates at the Local Optimizer stage where pushdown decisions are made, with the identified operators later executed by OCS. The OCS system comprises a frontend node that accepts Substrait IR plans and storage nodes that process them, returning results in Arrow columnar format. By integrating OCS-specific pushdown logic through Presto's SPI and leveraging Substrait/Arrow standards, this design enables advanced in-storage computation that is not possible with the conventional Hive connector. When a user submits an SQL query, the overall execution proceeds as follows:

(1) **Pushdown Target Selection:** The coordinator constructs the logical plan from the AST and applies both global and connector-specific optimizations. During this phase, Presto-OCS Connector's *Selectivity Analyzer* traverses the logical plan, estimating operator selectivity using Hive metastore statistics and identifying high data-reduction operators. As part of this traversal, the *Operator Extractor* captures the identified operators along with their associated SQL conditions, including filter predicates (e.g., range boundaries, equality constraints), aggregation specifications (e.g., GROUP BY keys, aggregate functions), and sorting criteria (e.g., ORDER BY columns, LIMIT values). These extracted components are preserved in the connector's internal data structures for subsequent translation into storage-executable formats.
(2) **Task Scheduling:** The scheduler partitions the optimized plan into tasks and assigns them to workers. Workers assigned to initial tasks, those responsible for retrieving data from remote storage, are notified about the pushdown operators selected in the previous phase. This notification includes information about which operators will be executed at the storage layer and indicates that intermediate results, rather than raw data,

will be returned from OCS. The workers can then prepare their execution context accordingly, expecting pre-processed data in Arrow format instead of raw table scans, and adjusting their subsequent processing logic to handle these partially computed results.

(3) **Translating and Submitting Pushdown Plans to Storage:** When workers execute initial tasks for data retrieval, they invoke the *PageSourceProvider*, a Presto SPI interface responsible for sending requests to remote storage and processing the returned data. The OCS connector's *PageSourceProvider* reconstructs the pushdown target operators and their associated conditions into SQL statements, combining filter predicates, aggregation functions, and sorting criteria into a cohesive query. This reconstructed SQL is then translated into Substrait IR, which provides a standardized format for expressing the query plan across different systems. The Substrait IR encapsulates all necessary execution details, including operator semantics, data types, and function signatures, ensuring accurate reproduction of the intended operators within the OCS storage layer.

(4) **In-Storage Execution on OCS:** OCS receives and parses the Substrait plan through its gRPC endpoint, then executes the plan using its embedded SQL engine. The storage nodes perform the operators directly on the stored data, executing filters to eliminate unnecessary rows, applying aggregations to reduce data volume, and performing sorts or limits as specified. Upon completion, OCS serializes the results into Apache Arrow columnar format, which provides an efficient binary representation for analytical workloads, and returns them to the requesting worker through the established gRPC connection.

(5) **Result Reception and Post-Processing:** The OCS Connector's *PageSourceProvider* receives the Arrow-formatted intermediate results from OCS and deserializes them into Presto's internal page format for further processing. These deserialized pages are then fed to the worker's execution pipeline as if they were standard table scan results, maintaining compatibility with Presto's existing operator implementations. The worker proceeds to execute any residual operators that were excluded from pushdown to storage, such as complex joins across multiple tables, window functions requiring global context, or user-defined functions incompatible with storage-side execution. These residual operators process the pre-filtered and pre-aggregated data from OCS, benefiting from the reduced data volume while preserving full SQL semantics. Once all residual operators complete their execution, the worker streams the final results directly to the client.

4 Implementation Details

We implemented the Presto-OCS connector on Presto version 0.286 with OpenJDK 11.0.22, using Apache Hive 3.0.0 as the metadata catalog. The implementation leverages Presto's stable SPI interfaces to ensure compatibility while adding pushdown capabilities through custom optimizer and page source provider components.

Local Optimizer: To identify pushdown opportunities, the optimizer analyzes query plans via Presto's logical plan tree structure and performs a bottom-up traversal using the *ConnectorPlanOptimizer* interface. For each node, it evaluates pushdown eligibility based on expression complexity, expected input size, and data reduction ratio.

The selection process leverages Hive metastore statistics including min/max values for range filter selectivity, Number of Distinct Values (NDV) for estimating aggregation cardinality, and row count for computing reduction ratios. For filter operators, the optimizer assumes a normal distribution of values between the column's min/max boundaries and estimates the proportion of rows falling within the query's range predicate. For aggregation operators, the optimizer estimates output cardinality as row_count/NDV of the GROUP BY column(s), where aggregations with low NDV are prioritized for pushdown as they produce minimal output. For top-N operators, selectivity calculation is straightforward as the LIMIT clause explicitly specifies the output row count, which can be directly compared against the total row count. The pushdown decision is based on comparing the estimated selectivity and computational complexity against user-configurable thresholds. Operators with selectivity above the threshold and acceptable computational overhead are marked as pushdown candidates. However, this approach has limitations. The assumptions of normal distributions may not hold for skewed data distributions. Adapting to diverse data distributions dynamically and determining optimal thresholds for different workload characteristics remain important areas for future work.

Selected operators are recorded in the connector's table metadata structure along with their dependency relationships and execution order constraints. The corresponding PlanNodes are merged into a modified TableScan operator, which encapsulates the pushdown operators under the assumption that OCS will execute these operators and return only the processed results.

Page Source Provider: During query execution, Presto invokes the connector's *PageSourceProvider* to retrieve data from storage. Our OCS-specific implementation generates Substrait IR for the operators stored in table metadata. The translation process extracts pushdown operators and reconstructs them into SQL statements, combining filters with predicates, aggregations with grouping keys and functions, and sorts with ordering criteria. These SQL statements are then translated into Substrait IR through complex mappings: SQL clauses become Substrait relations, expressions are transformed with proper type casting, and Presto's function signatures map to Substrait's standardized namespace. Type normalization handles differences in null handling, decimal precision, and timestamp representations between systems. The completed Substrait plan is serialized using Protocol Buffers and transmitted to OCS via gRPC. OCS executes the query using its embedded SQL engine and returns Arrow columnar results, which the connector deserializes into Presto's page format with necessary type conversions.

Pushdown Monitoring and Auxiliary Components: The connector implements monitoring via Presto's *EventListener* interface to collect runtime statistics, including operator execution times, data volumes, and pushdown success rates. The collected metrics are stored in a pushdown history component that maintains a sliding window of recent executions to identify patterns and inform future optimization decisions. Additional components handle column metadata, table properties, schema synchronization, and split management for parallel execution, ensuring reliable operation.

Table 1: Hardware specifications

Compute Node Specifications	
CPU	Intel(R) Xeon(R) Gold 6226R (64 cores, 2.9 GHz max)
Memory	384 GB DDR4
Storage	1 TB NVMe SSD
Frontend Node Specifications	
CPU	Intel® Xeon® Silver 4410Y (48 cores, 3.9 GHz max)
Memory	64 GB DDR4
Storage	1 TB NVMe SSD
Storage Node Specifications	
CPU	Intel® Xeon® Silver 4410Y (16 cores, 2.0 GHz max)
Memory	64 GB DDR4
Storage	1 TB NVMe SSD + 512 GB SATA SSD

5 Evaluation

5.1 Experimental Setup

Testbeds. Table 1 lists the hardware specifications of our experimental setup. The compute node runs a single-node Presto deployment, with the coordinator and one worker on the same machine. The Object-based Computational Storage (OCS) follows a hierarchical design comprising a frontend node and multiple backend storage nodes. The frontend exposes a unified endpoint to applications, parses incoming queries, and dispatches them to the appropriate storage node. Each storage node holds data and includes an embedded SQL engine to execute query plans locally. For our experiments, we used a single storage node. Note that we restricted the storage node to 16 cores at 2.0 GHz (compared to 48 cores at 3.9 GHz for the frontend) to emulate the resource-constrained environment typical of production storage nodes. All nodes are interconnected through a 10 GbE Ethernet network. All machines, including the compute and OCS nodes, run Ubuntu 22.04.4 LTS. Presto version 0.286 is used as the distributed SQL query engine, compiled with OpenJDK 11.0.22. Apache Hive 3.0.0 serves as the metadata catalog for schema and statistics.

Workloads. To evaluate the performance impact of leveraging OCS's extended pushdown capabilities in Presto, we employ scientific simulation datasets with their corresponding analytical queries used at Los Alamos National Laboratory (LANL), as well as a standard decision-support benchmark (TPC-H).

- **Laghos Dataset:** The *LAGrangian High-Order Solver (Laghos)* dataset is generated by a fluid dynamics simulation mini-application on unstructured meshes. It consists of 256 Parquet files, each containing 10 columns and 4,194,304 rows, totaling approximately 24 GB [29]. To evaluate pushdown performance of various operators, we modified the original LANL query by appending a LIMIT clause, thereby introducing a top-N operator into the plan.
- **Deep Water Dataset:** The *Deep Water Asteroid Impact* dataset simulates an asteroid strike in a deep-sea environment. It contains 64 Parquet files, each representing a snapshot at a different timestep, with 27 million rows and 4 columns per file, totaling 30 GB [22].
- **TPC-H Benchmark:** To evaluate performance in traditional Online Analytical Processing (OLAP) scenarios, we include Query 1 from the TPC-H benchmark [47], which represents aggregation-heavy workloads in decision-support systems.

Table 2 presents the SQL queries, their selectivity (ratio of result to input size), and Presto logical execution plans for each dataset.

5.2 Results

For the evaluation, we conducted experiments and analyses guided by four research questions.

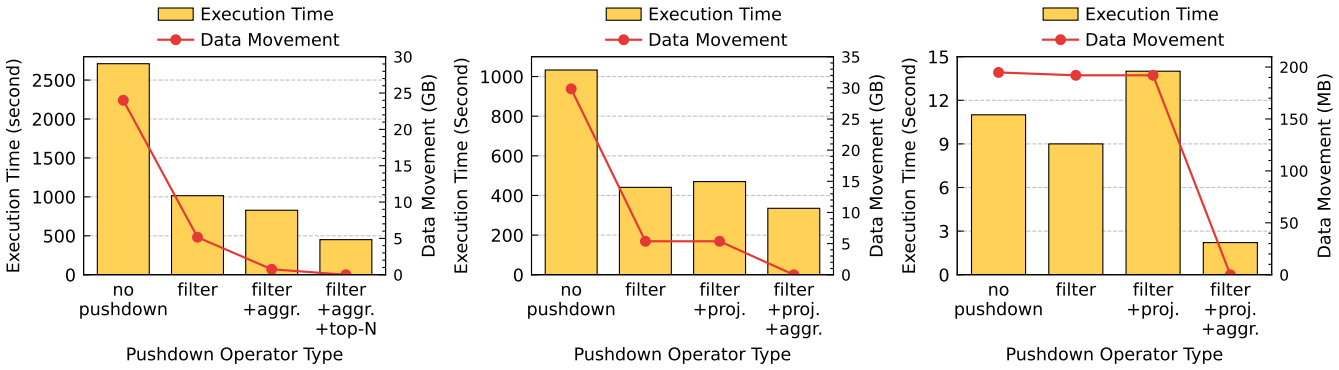
Q1. Does reducing data movement through pushdown improve query execution time?: Figure 5 compares query execution time and data movement across three workloads as we progressively enable query pushdown for different operator types, where the x-axis shows the cumulative set of operators selected for pushdown (following the execution order in Table 2) and the y-axis shows both execution time in seconds (bars) and data movement from OCS to Presto (red solid line) on separate scales, with data movement measured in GB for Laghos and Deep Water Impact dataset, and MB for TPC-H Q1. Figure 5 (a) presents performance results for the *Laghos* dataset. Since each operator in this query reduces its output size relative to its input, additional operator pushdown consistently decreases both data movement and execution time. Data movement decreases from 24GB without pushdown to 5.1GB with filter-only pushdown, 0.75GB when aggregation is added, and merely 0.0005GB with complete pushdown of all three operators (filter, aggregation, and top-N). These reductions in data movement directly lead to performance improvements: execution time decreases from 2,710 seconds at baseline to 1,015 seconds with filter-only pushdown, 828 seconds with filter and aggregation pushdown, and 450 seconds with full operator pushdown. This implies that minimizing data movement is critical to performance optimization. Notably, the complete pushdown configuration reduces data movement by 99.99% (from 5.1 GB to 0.0005 GB) and achieves a 2.25× speedup compared to the filter-only pushdown approach employed by traditional object storage systems. These results highlight the limitations of traditional object storage systems that support only filter and column projection pushdown. The Presto-OCS connector leverages OCS's extended pushdown capabilities to offload aggregation and top-N operators, enabling in-storage processing that delivers performance gains beyond those achievable through conventional filter-based approaches.

Q2. Is pushdown always beneficial regardless of operator type?: Query pushdown does not guarantee performance improvements for every operator type. As shown in Figure 5 (b), for the *Deep Water Impact* dataset, filter-only pushdown reduced data movement from 30GB to 5.37GB (82% reduction) and execution time from 1,033 seconds to 441 seconds (2.33× speedup) compared to no pushdown. In Figure 5 (c), TPC-H Q1 showed minimal data movement reduction from 194MB to 192MB (1.03% reduction) but still achieved a 1.22× speedup with filter-only pushdown. However, in both cases, expression projection pushdown resulted in increased execution times. Specifically, *Deep Water Impact* experienced a 7% slowdown and TPC-H Q1 experienced a 55% slowdown when projection pushdown was added to filter-only pushdown. This is due to the computational overhead of expression evaluation in projections, which reference multiple columns and involve complex arithmetic.

Offloading such projections to OCS rather than computing them on the more capable compute node introduced additional latency while providing no data movement reduction. In both datasets, adding aggregation pushdown after filter and projection dramatically reduced data movement and recovered performance. Compared to filter-only pushdown, *Deep Water Impact* achieved a 1.32×

Table 2: Queries used for each dataset and their selectivity.

Dataset	Query	Selectivity	Execution Plan
Laghos	SELECT min(vertex_id) AS VID, min(x), min(y), min(z), avg(e) FROM parquet WHERE x, y, z BETWEEN 0.8 AND 3.2 GROUP BY vertex_id ORDER BY E LIMIT 100	0.0023842%	TableScan → Filter → Aggregation → Top-N
Deep Water	SELECT MAX((rowid % (500*500))/500), timestep FROM parquet WHERE v02 >0.1 GROUP BY timestep	0.0000032%	TableScan → Filter → Project → Aggregation
TPC-H	SELECT returnflag, linestatus, SUM(quantity), SUM(extendedprice), SUM(extendedprice * (1 - discount)), SUM(extendedprice * (1 - discount) * (1 + tax)), AVG(quantity), AVG(extendedprice), AVG(discount), COUNT(*) FROM lineitem WHERE shipdate ≤ DATE '1998-12-01' - INTERVAL '90 DAY' GROUP BY returnflag, linestatus ORDER BY returnflag, linestatus	0.0000667%	TableScan → Filter → Project → Aggregation → Sort



(a) Laghos (b) Deep Water Impact (c) TPC-H

Figure 5: Execution time comparison. For each dataset, query pushdown was progressively applied to SQL operators in execution order to observe changes in runtime and data movement.

speedup (441 to 335 seconds) and reduced data movement from 5.37GB to 1MB, while TPC-H Q1 achieved a 4.07× speedup (9 to 2.21 seconds) and reduced data movement from 192MB to 0.5MB. This is attributed to the nature of aggregation operators, which consolidate multiple rows into a single row per group using functions like MAX, SUM, and AVG. Since the query for the *Deep Water Impact* dataset has a single GROUP BY key that produces only one distinct group and the TPC-H Q1 query has multiple GROUP BY keys that result in just four groups, the aggregation results contain just one and four rows respectively, reducing the data size. These results highlight that the effectiveness of pushdown depends on the operator type, complexity, and order, as well as the trade-off between compute and network overhead.

Q3. How does OCS pushdown perform with data compression?: To evaluate the impact of data compression on pushdown performance, we conducted experiments using the *Deep Water Impact* dataset in its raw form and with lossless compression algorithms: Snappy [18], GZip [17], and Zstd [12]. We limit our experiments to lossless compression because the standard Parquet ecosystem currently supports only lossless codecs. While lossy compression techniques such as SZ [25, 46] or ZFP [26] could achieve higher compression ratios for scientific data, implementing them within Parquet would require custom extensions beyond the current Parquet ecosystem. Exploring the performance when combining query pushdown with lossy compression remains an important direction for future work.

Figure 6 presents the execution times for both filter-only and all-operator pushdown configurations across these compression schemes. The filter-only pushdown represents the approach supported by conventional object storage systems, while the all-operator pushdown leverages OCS's extended capabilities to pushdown the complete operator chain.

The results show that query pushdown and compression are complementary techniques that together achieve superior performance. Compressed data with basic filter-only pushdown (451.7 seconds with Zstd) outperformed uncompressed data with full operator pushdown (530.4 seconds) demonstrating that compression combined with basic filter pushdown can deliver substantial performance gains over advanced pushdown alone. This highlights that compression remains a valuable optimization technique that should not be overlooked when implementing near-data processing. However, when comparing within the same compression scheme, OCS's extended pushdown capabilities consistently outperformed filter-only pushdown across all compression methods.

For uncompressed data, all-operator pushdown achieved a 1.22× speedup (from 649.3 to 530.4 seconds) over filter-only pushdown. This performance gap widened with compression: Snappy showed a 1.37× speedup, GZip a 1.39× speedup, and Zstd achieved a 1.36× speedup when upgrading from filter-only to all-operator pushdown. Furthermore, higher compression ratios correlated with better performance in both pushdown configurations. Execution times decreased progressively from no compression to Zstd, with the most aggressive compression (Zstd) achieving the best results: 451.7 seconds for filter-only and 331.6 seconds for all-operator pushdown. This trend confirms that the I/O reduction from compression

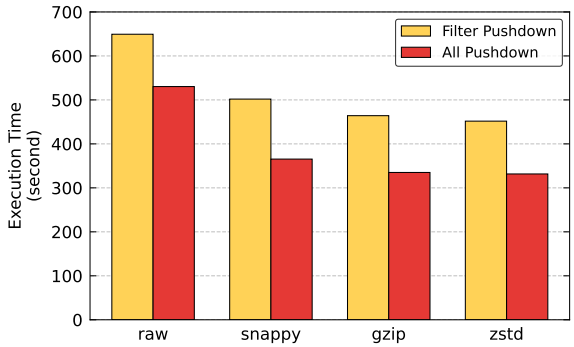


Figure 6: Impact of compression algorithms on pushdown performance. Lower execution times indicate better performance.

Table 3: Breakdown of execution time for a single query

Execution Stage	Time (ms)	Share (%)
Logical Plan Analysis	1	0.06%
Substrait IR Generation	33	1.94%
Pushdown & Result Transfer	682	40.12%
Presto Execution (Post-Scan)	814	47.90%
Others	169	9.97%
Total	1,700	100%

outweighs decompression overhead, even in storage-side query processing.

These findings suggest that OCS-enabled operator pushdown serves as a powerful complement to existing compression techniques. While compression reduces data size at rest and during transfer, extended pushdown with aggregation and projection further reduces data movement by processing queries in-storage.

Q4. Does the Presto-OCS connector incur significant overhead?: We evaluate whether the Presto-OCS connector introduces significant overhead from query plan traversal for pushdown operator selection or Substrait Intermediate Representation (IR) generation. Table 3 quantifies the overhead of pushdown optimization and Substrait conversion. For a single query on one Parquet file from the Laghos dataset, logical plan traversal and Substrait IR generation accounted for only 0.06% and 1.94% of the total execution time, respectively. The combined overhead for all pushdown-related logic was under 2%, demonstrating that the Presto-OCS connector achieves its optimization goals with negligible impact on performance.

6 Related Work

Scientific computing in HPC environments has addressed data movement bottlenecks through various approaches. Systems like ADIOS [28], Catalyst [10], and Damaris [13] enable in-situ and in-transit processing, analyzing data while still in memory or during transfer to avoid intermediate storage operations. Compression remains the cornerstone of modern scientific data reduction, with methods generally divided into lossless and lossy schemes. Lossless compression [12, 17, 18, 24, 27, 31, 44, 52] guarantees exact fidelity and is required in situations where even the smallest deviation is unacceptable. Lossy compression achieves much higher reduction by tolerating controlled error. Frameworks such as SZ [25, 46], ZFP [26], and MGARD [1] let users specify error bounds, balancing size reduction and accuracy.

Recent query pushdown systems have explored in-storage processing strategies for reducing data movement. PushdownDB [50] explores how query pushdown can improve DBMS performance while leveraging AWS S3. FlexPushdownDB [49] introduces a hybrid approach that combines Weighted-LFU policy caching with computation pushdown to optimize query performance. Fusion [15] addresses the inefficiency of traditional computation pushdown on erasure-coded storage and proposes an object store specifically optimized for query pushdown on erasure-coded data.

Computational Storage Devices (CSDs) can offer hardware-level solutions for data-intensive workloads. KV-CSD [37] embeds key-value processing logic directly in storage devices for HPC applications. YourSQL [21] offloads database operations to CSDs by implementing a storage-side query execution engine that processes SQL queries directly within the storage device.

While compression enables substantial size reductions, it remains constrained by the need to restore data into its original form (even with lossy methods) and therefore must account for all data points in a dataset. By contrast, this paper explores a research direction where reduction is achieved by processing data directly at its storage location. In this model, only the processed results—often orders of magnitude smaller than the raw input—are returned, thereby reducing data movement. This approach reduces data without being constrained by entropy and is particularly effective when analysis is bottlenecked by network transfer rather than storage media bandwidth. However, its benefits diminish when storage bandwidth itself is the primary constraint.

In practice, we expect in-storage processing to be used in tandem with compression, reducing local storage read overhead while further minimizing network transfers to deliver optimal performance.

7 Conclusion

This paper presents the design and implementation of a connector that integrates OCS with Presto, a widely used distributed SQL engine, to enable query pushdown of advanced SQL operators supported by OCS. The proposed Presto-OCS connector leverages Presto's Connector SPI to detect data-reducing operators during query planning, translates them into Substrait intermediate representations, and offloads them to OCS for in-storage execution. A key architectural achievement is that the connector extends the Presto SPI to unlock OCS's advanced pushdown capabilities while maintaining seamless compatibility with the existing ecosystem, all without altering the core Presto framework. Experimental evaluation with real-world HPC and OLAP workloads demonstrates that the Presto-OCS connector achieves up to a 4.07× speedup in query execution and a 99% reduction in data movement compared to filter-only pushdown. Furthermore, achieving up to a 1.39× speedup on compressed data confirms that advanced query pushdown is a powerful complement to data compression.

Acknowledgments

This work was supported by SK hynix Inc.

References

[1] Mark Ainsworth, Ozan Tugluk, Ben Whitney, and Scott Klasky. 2018. Multilevel techniques for compression and reduction of scientific data—the univariate case. *Springer Computing and Visualization in Science* 19, 5–6 (2018).

[2] Amazon Web Services. 2024. Amazon Athena Documentation. https://docs.aws.amazon.com/athena/latest/ug/what-is.html.

[3] Amazon Web Services, Inc. 2006. Amazon S3. https://aws.amazon.com/s3/. [Online; accessed 2025-08-24].

[4] Apache Foundation. 2010. Apache Hive. https://hive.apache.org/. [Online; accessed 2025-08-24].

[5] Apache Foundation. 2010. Apache Spark. https://spark.apache.org/. [Online; accessed 2025-08-24].

[6] Apache Foundation. 2013. Apache ORC. https://orc.apache.org/. [Online; accessed 2025-08-24].

[7] Apache Foundation. 2013. Apache Parquet. https://parquet.apache.org/. [Online; accessed 2025-08-24].

[8] Apache Foundation. 2013. HDFS Architecture Guide. https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html. Accessed: 2025-08-23.

[9] Apache Foundation. 2016. Apache Arrow. https://arrow.apache.org/. [Online; accessed 2025-08-24].

[10] Utkarsh Ayachit, Andrew Bauer, Berk Geveci, Patrick O'Leary, Kenneth Moreland, Nathan Fabian, and Jeffrey Mauldin. 2015. ParaView Catalyst: Enabling In Situ Data Analysis and Visualization. In *Proceedings of the First Workshop on In Situ Infrastructures for Enabling Extreme-Scale Analysis and Visualization*. doi:10.1145/2828612.2828624

[11] Zbigniew Baranowski and Vasileios Dimakopoulos. 2020. Introduction to Presto. CERN Indico Event 869037. https://indico.cern.ch/event/869037/contributions/3663775/attachments/1960650/3258410/Introduction_to_Presto.pdf

[12] Yann Collet and Chip Turner. 2016. Zstandard - Real-time Data Compression Algorithm. https://github.com/facebook/zstd.

[13] Matthieu Dorier, Gabriel Antoniu, Franck Cappello, Marc Snir, Robert Sisneros, Orcun Yildiz, Shadi Ibrahim, Tom Peterka, and Leigh Orf. 2016. Damaris: Addressing performance variability in data management for post-petascale simulations. *ACM Transactions on Parallel Computing* 3, 3 (2016). doi:10.1145/2987371

[14] Dominik Durner, Viktor Leis, and Thomas Neumann. 2023. Exploiting Cloud Object Storage for High-Performance Analytics. *Proceedings of the VLDB Endowment* 16, 11 (2023). doi:10.14778/3603581.3603592

[15] Michael Freedman, Qiwen Zhang, Erik Peterson, and Frank McSherry. 2025. Fusion: An Analytics Object Store Optimized for Query Pushdown. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. doi:10.1145/3669940.3707234

[16] Frank Gadban and Julian Kunkel. 2021. Analyzing the Performance of the S3 Object Storage API for HPC Workloads. *MDPI Applied Sciences* 11, 18 (2021). doi:10.3390/app11188540

[17] GNU Project. 2022. GNU Gzip: General-Purpose Data Compression Software. https://www.gnu.org/software/gzip/.

[18] Google. 2011. Snappy. https://github.com/google/snappy. [Online; accessed 2025-08-24].

[19] Google. 2015. gRPC. https://grpc.io. [Online; accessed 2025-08-24].

[20] Junmin Gu, Scott A. Klasky, Norbert Podhorszki, Ji Qiang, and Kesheng Wu. 2018. Querying Large Scientific Data Sets with Adaptable IO System ADIOS. In *Supercomputing Frontiers*, Vol. 10776. Oak Ridge National Laboratory. doi:10.1007/978-3-319-69953-0_4

[21] Insoon Jo, Duck-Ho Bae, Andre S. Yoon, Jeong-Uk Kang, Sangyeun Cho, Daniel D. G. Lee, and Jaeheon Jeong. 2016. YourSQL: a high-performance database system leveraging in-storage computing. *Proc. VLDB Endow.* 9, 12 (2016). doi:10.14778/2994509.2994512

[22] Patchett John, Samsel Francesca, Tsai Karen, Gisler Galen, Rogers David, Abram Greg, and Turton Terece. 2016. Visualization and analysis of threats from asteroid ocean impacts. Los Alamos National Laboratory Technical Report.

[23] Jongryool Kim. 2023. Accelerating Data Analytics Using Object Based Computational Storage in a HPC. https://sc23.supercomputing.org/proceedings/exhibitor_forum/exhibitor_forum_pages/exforum116.html The International Conference for High Performance Computing, Networking, Storage, and Analysis (SC).

[24] Fabian Knorr, Peter Thoman, and Thomas Fahringer. 2021. ndzip: A high-throughput parallel lossless compressor for scientific data. In *2021 Data Compression Conference (DCC)*.

[25] Xin Liang, Sheng Di, Dingwen Tao, Sihuan Li, Shaomeng Li, Hanqi Guo, Zizhong Chen, and Franck Cappello. 2018. Error-Controlled Lossy Compression Optimized for High Compression Ratios of Scientific Datasets. In *Proceedings of the 2018 IEEE International Conference on Big Data (Big Data)*. doi:10.1109/BigData.2018.8622520

[26] Peter Lindstrom. 2014. Fixed-Rate Compressed Floating-Point Arrays. *IEEE Transactions on Visualization and Computer Graphics* 20, 12 (2014). doi:10.1109/TVCG.2014.2346458

[27] Peter Lindstrom and Martin Isenburg. 2006. Fast and Efficient Compression of Floating-Point Data. *IEEE Transactions on Visualization and Computer Graphics* 12, 5 (2006). doi:10.1109/TVCG.2006.143

[28] Jay F. Lofstead, Scott Klasky, Karsten Schwan, Norbert Podhorszki, and Chen Jin. 2008. Flexible IO and integration for scientific codes through the adaptable IO system (ADIOS). In *Proceedings of the 6th International Workshop on Challenges of Large Applications in Distributed Environments*. doi:10.1145/1383529.1383533

[29] Los Alamos National Laboratory. 2024. Laghos Sample Dataset. https://github.com/lanl-ocs/laghos-sample-dataset. [Online; accessed 2025-08-24].

[30] Zhenxiao Luo, Lu Niu, Venki Korukanti, Yutian Sun, Masha Basmanova, Yi He, Beinan Wang, Devesh Agrawal, Hao Luo, Chunxu Tang, Ashish Singh, Yao Li, Peng Du, Girish Baliga, and Maosong Fu. 2022. From Batch Processing to Real Time Analytics: Running Presto at Scale. In *Proceedings of the 2022 IEEE 38th International Conference on Data Engineering (ICDE)*. doi:10.1109/ICDE53745.2022.00165

[31] LZ4. 2011. LZ4 - Extremely fast compression. https://github.com/lz4/lz4. [Online; accessed 2025-08-24].

[32] Dominic Manno. 2023. Improving Storage Systems for Simulation Science with Computational Storage. Compute+Memory+Storage Summit.

[33] Sergey Melnik, Andrey Gubarev, Jing Jing Long, Geoffrey Romer, Shiva Shivakumar, Matt Tolton, Theo Vassilakis, Hossein Ahmadi, Dan Delorey, Slava Min, Mosha Pasumansky, and Jeff Shute. 2020. Dremel: A Decade of Interactive SQL Analysis at Web Scale. *Proceedings of the VLDB Endowment* 13, 12 (2020). doi:10.14778/3415478.3415568

[34] MinIO, Inc. 2016. MinIO: S3 Compatible, Exascale Object Store for AI. https://min.io/. Accessed: 2025-08-23.

[35] MinIO, Inc. 2019. MinIO Select: S3 Select API Support. https://github.com/minio/minio/blob/master/docs/select/README.md. Accessed: 2025-08-23.

[36] Xi Pang and Jianguo Wang. 2024. Understanding the Performance Implications of the Design Principles in Storage-Disaggregated Databases. In *Proceedings of the ACM on Management of Data (PACMMOD)*. doi:10.1145/3654983

[37] Inhyuk Park, Qing Zheng, Dominic Manno, Soonyeal Yang, Jason Lee, David Bonnie, Bradley Settlemyer, Youngjae Kim, Woosuk Chung, and Gary Grider. 2023. KV-CSD: A Hardware-Accelerated Key-Value Store for Data-Intensive Applications. In *Proceedings of the 2023 IEEE International Conference on Cluster Computing (CLUSTER)*. doi:10.1109/CLUSTER52292.2023.00019

[38] Presto Foundation. 2013. Hive Connector. https://prestodb.io/docs/current/connector/hive.html. Accessed: 2025-08-23.

[39] Presto Foundation. 2013. SPI Overview. https://prestodb.io/docs/current/develop/spi-overview.html. Accessed: 2025-08-23.

[40] Randall Hunt. 2017. S3 Select and Glacier Select – Retrieving Subsets of Objects. https://aws.amazon.com/ko/blogs/aws/s3-glacier-select/

[41] Raghav Sethi, Martin Traverso, Dain Sundstrom, David Phillips, Wenlei Xie, Yutian Sun, Nezih Yegitbasi, Haozhun Jin, Eric Hwang, Nileema Shingte, and Christopher Berner. 2019. Presto: SQL on Everything. In *Proceedings of the 2019 IEEE 35th International Conference on Data Engineering (ICDE)*.

[42] Nick Smith, Bo Jayatilaka, David Mason, Oliver Gutsche, Alison Peisker, Robert Illingworth, and Chris Jones. 2023. A Ceph S3 Object Data Store for HEP. *arXiv preprint arXiv:2311.16321* (2023). https://arxiv.org/abs/2311.16321

[43] Snowflake Inc. 2014. Snowflake AI Data Cloud. https://www.snowflake.com/. Accessed: 2025-08-23.

[44] James A Storer and Thomas G Szymanski. 1982. Data compression via textual substitution. *Journal of the ACM (JACM)* 29, 4 (1982).

[45] Substrait Project. 2021. Substrait. https://substrait.io/. Accessed: 2025-08-23.

[46] Dingwen Tao, Sheng Di, Zizhong Chen, and Franck Cappello. 2017. Significantly Improving Lossy Compression for Scientific Data Sets Based on Multidimensional Prediction and Error-Controlled Quantization. In *Proceedings of the 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. doi:10.1109/IPDPS.2017.115

[47] Transaction Processing Performance Council. 2017. TPC Benchmark H (Decision Support). http://www.tpc.org/tpch/. Revision 2.17.3.

[48] Andrew Waldman. 2018. Evaluation of the Presto Query Engine for integrating relational databases with big data platforms at scale. CERN openlab Summer Student Lightning Talk. https://cds.cern.ch/record/2634287

[49] Yifei Yang, Xiangyao Yu, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2024. FlexpushdownDB: Rethinking Computation Pushdown for Cloud OLAP DBMSs. *The VLDB Journal* 33, 5 (2024).

[50] Xiangyao Yu, Matt Youill, Matthew Woicik, Abdurrahman Ghanem, Marco Serafini, Ashraf Aboulnaga, and Michael Stonebraker. 2020. PushdownDB: Accelerating a DBMS Using S3 Computation. In *Proceedings of the 2020 IEEE 36th International Conference on Data Engineering (ICDE)*. doi:10.1109/ICDE48307.2020.00174

[51] Qing Zheng. 2023. Toward Open Object-Based Computational Storage For Analysis Query Pushdown. The 9th International Parallel Data Systems Workshop (PDSW) Work-in-Progress (WIP) Session.

[52] Jacob Ziv and Abraham Lempel. 1977. A universal algorithm for sequential data compression. *IEEE Transactions on Information Theory* 23, 3 (1977).