CORD: Parallelizing Query Processing across Multiple Computational Storage Devices

Wahid Uz Zaman, Cyan Subhra Mishra, Saleh AlSaleh, Abutalib Aghayev, Mahmut Taylan Kandemir
The Pennsylvania State University, USA
{wzz5219, cyan, saa6315, agayev, mtk2}@psu.edu

2025 IEEE International Parallel and Distributed Processing Symposium (IPDPS) | 979-8-3315-3237-6/25/$31.00 ©2025 IEEE | DOI: 10.1109/IPDPS64566.2025.00104

Abstract*—Query processing on large-scale scientific datasets often suffers from performance bottlenecks due to significant data transfers between storage nodes and applications in decoupled distributed storage environments. This issue is particularly pronounced in high-selectivity queries where unnecessary data is transferred between the storage plane and the compute plane. To tackle this challenge, we introduce the integration of SmartSSDs, functioning as Computational Storage Devices (CSDs), into the storage layer. By offloading simple filter-projection operations to these CSDs, we significantly reduce data transfer bottlenecks, leading to lower query latency and higher throughput. Our novel framework, CORD (parallelizing query processing across multiple Computational stORage Devices), facilitates parallel query execution across multiple CSDs while considering data locality. CORD is compatible with any decoupled storage system equipped with CSDs. Our extensive empirical evaluation demonstrates that CORD achieves up to 93× speedup for high-selectivity queries compared to traditional (compute plane) execution strategy and offers a further 1.64× speedup in cases of uneven data distribution. Additionally, we present two optimizations for batch query processing. Results from our experiments with 4 CSDs reveal substantial performance improvements provided by the optimizations embedded in CORD.

I. Introduction

In the fields of cloud computing and high performance computing (HPC), "disaggregated" and "distributed" storage architectures are pivotal for optimizing data storage. Disaggregated storage *separates* compute and storage functionalities, allowing for independent scaling of each and flexible data access. Distributed storage, on the other hand, *spreads* data across multiple nodes to increase availability and performance. However, both architectures frequently face significant data transfer overhead between storage nodes (SN) and client or compute nodes, especially when processing large volumes of small scientific data records generated, for example, by massively parallel scientific simulations and accessed later for interactive data analytics [1]–[3], particularly with high-selectivity queries. Even large-scale data analytics engines such as Presto [4], Hive [5], and Spark [6] encounter these overheads during query processing, despite using columnar data formats. This often leads to inefficient resource utilization and extended query response times. Therefore, reducing data transfer is essential to improve the effectiveness of modern data analytics platforms and their associated query engines.

Recent advances in **computational storage devices** (CSDs), such as SmartSSD [7], [8], KV-CSD [9], and Scaleflux [10] represent a transformative shift in data processing by integrating computational capabilities directly within the storage medium. This integration facilitates the processing of data closer to its storage location, markedly reducing data movement, thereby accelerating processing times and decreasing latency. This approach is particularly advantageous for the massive datasets prevalent in scientific and big data applications. SmartSSD (henceforth referred to interchangeably as CSD) has demonstrated promising benefits in query processing tasks with columnar data [8], [11], [12], yet its potential in distributed and disaggregated storage architectures remains largely unexplored, particularly for high-selectivity queries. Our study explores how SmartSSD enhances the processing of filter-projection-based queries (query format is defined in Section IV) by offloading the task to storage nodes, with the aim of reducing data transfer costs and improving performance and energy efficiency, particularly for highly selective queries.

However, integrating CSDs within storage nodes introduces several challenges. For instance, applications or query engines might need to coordinate query execution across multiple CSDs, treating them as distinct compute units within the storage architecture. The effective utilization of multiple CSDs within a single storage node is crucial to improve computational efficiency. Moreover, executing batch queries in parallel at the storage node level and potentially running queries on multiple CSDs in an out-of-order manner are key to optimizing resource use. This becomes even more critical in large-scale query processing environments, where data is distributed across multiple CSDs and storage nodes, often scaling from terabytes to petabytes. There is also a substantial opportunity to improve system performance through intelligent caching strategies or by reusing results from previously executed queries, thereby minimizing redundant computations.

To address these challenges, we propose a holistic system architecture that seamlessly integrates CSD capabilities within modern storage nodes for optimized query processing. This architecture not only boosts the efficiency of data processing, but also offers a flexible framework to meet the dynamic demands of contemporary scientific and data-intensive applications. A crucial element of this solution is to provide application programmers with an "interface" to facilitate query offloading to CSDs, taking into account the distributed nature of data and the parallel processing of queries. To this end, we introduce **CORD** (parallelizing query processing across multiple **C**omputational st**OR**age **D**evices), a computation offloading framework designed to parallelize query processing across multiple CSDs. This paper makes the following **primary contributions**:

- Observing that the current low-level interfaces that accompany emerging CSDs slow their adoption and make code porting from one CSD to another difficult, CORD provides three "high-level interfaces" for programmers to interact

1530-2075/25/$31.00 ©2025 IEEE
DOI 10.1109/IPDPS64566.2025.00104

with the CSDs. These interfaces are general and modular enough to integrate with various file systems and databases. While our **prescriptive** interface enables the programmer to have complete control over the storage-side query execution and corresponding data movements, our **descriptive** and **predictive** interfaces enable *automated* storage system-wide optimizations – by balancing storage-side parallelism and storage-side locality – for overall performance gains.

- CORD employs a unique "caching mechanism" that caches result bitmaps generated by processed filter functions in a "history" data structure. This approach allows CORD to reuse previously computed filter results as needed, thereby eliminating redundant computations. Consequently, this design conserves both CSD resources and processing time.
- CORD also introduces "multi-query optimization" across multiple CSDs and storage nodes by leveraging inter-query dependencies to execute queries in an out-of-order fashion while maximizing storage-node level parallelism and locality.

Our extensive experimental evaluation, using three distinct datasets in a distributed storage environment with dual SmartSSDs per storage node, reveals significant performance gains. More specifically, CORD can achieve up to 93× speedup for highly selective queries. Our results indicate that descriptive execution works better than prescriptive locality-aware execution for all kinds of uneven data distribution within a storage node. Moreover, the predictive execution of batch queries shows promising improvements based on the interrelations among the queries.

II. Related Work

Computational storage has gained considerable attention for its ability to perform computations near the data, reducing data transfer costs and enhancing query performance. Many studies [13]–[28] have demonstrated its potential, with commercial solutions like ScaleFlux [10], Newport CSD [18], and Catalina [22] gaining traction. SmartSSD has been a focal point, particularly in query analytics, where its FPGA-accelerated capabilities have shown significant improvements in data processing [11], [12]. However, evaluations have either been limited to single-node environments or have not examined the parallelization of query processing across SmartSSDs.

In the context of disaggregated storage, the rise of serverless computing and hardware specialization brings both challenges and opportunities, especially concerning the overhead of communicating with remote storage. Mahapatra's work [29] introduces a serverless execution model that integrates domain-specific accelerators directly into storage nodes, showcasing a path for enhancing computation in disaggregated storage environments, particularly in machine learning (ML) applications. Alibaba's POLARDB [30], a cloud-native relational database, incorporates computational storage drives, which improve query processing latency by offloading tasks to CSD-equipped storage nodes. However, this approach does not fully address the optimization of query parallelization when data is unevenly distributed across multiple CSDs. In comparison, Skyhook [31] adopts a "programmable storage" model, utilizing Ceph object storage to handle data management directly within the storage layer by performing computations on storage node CPUs (not CSDs). By enabling query push-down at the storage level, Skyhook reduces data transfer bottlenecks, although resource contention on the limited storage node CPU remains a potential concern.

CORD provides interfaces for query engines and programmers to leverage the processing power of SmartSSDs, allowing highly selective query processing to be offloaded to and parallelized over storage nodes and their SmartSSDs. We want to emphasize that, unlike prior art, our descriptive and predictive functions *automate* such offloading process and automatically *parallelize* the offloaded computation across multiple CSDs. Further, CORD can efficiently utilize all available compute engines within storage nodes, including both SmartSSDs and storage node CPUs. As such, it alleviates the data transfer bottleneck between storage plane and compute plane.

III. Background : SmartSSD

SmartSSD is a 4-TB NVMe SSD that is equipped with an AMD Kintex™ Ultrascale+ KU15P FPGA. A PCIe Gen3x4 bus connects the host processor to the SmartSSD. This bus not only lets one send standard SSD commands, but it also lets one send FPGA computation and read/write requests for FPGA DRAM through the XRT driver. In addition to host-driven commands, SmartSSD enables data movement over the internal data path between its NVMe SSD and the FPGA DRAM through its onboard PCIe switch, which we refer to as "peer-to-peer" (P2P) communication [32]. This internal link is also facilitated by PCIe Gen3x4. However, the unique feature of P2P is that it eliminates the traditional host-to-storage and host-to-accelerator PCIe traffic, thereby enhancing latency. In addition, this P2P guarantees the scalability of SmartSSDs that are connected to the same host by providing internal communication, rather than exhausting the main PCIe bus. FPGA can access its DRAM through a channel that offers a maximum bandwidth of 19.2GB/sec [33].

IV. Supported Query Format and Data Layout

Query Format: CORD supports queries in the format: "select column1,..., columnM from dataset T where filter1 AND (OR) filter2 ... filterN." These queries filter data and retrieve specified columns, with filtering efficiently handled on the CSDs' FPGA. While current support focuses on filtering operations ($>$, $<$, $=$) on non-string data, string-based functions like regex can also be added through FPGA kernel development. CORD enhances query efficiency and minimizes data transfer, specifically optimized for highly selective queries on large-scale, immutable distributed datasets. It can be integrated with database engines or such applications to offload highly selective filter-projection queries to the storage layer, eliminating the need to manage multi-CSD processing. The database engine can then perform complex operations, such as 'aggregation' or 'join' as usual, on the small filtered dataset CORD returns.

Storage Data Layout: CORD operates on datasets where specific column or property value is stored sequentially on

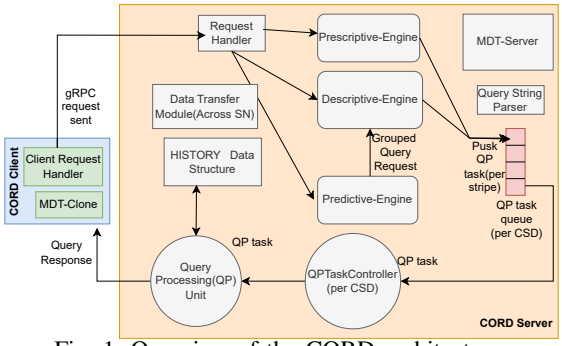


Fig. 1: Overview of the CORD architecture.

disk. This storage format offers a significant advantage: filtering and projection operations can retrieve only the relevant data, avoiding the overhead of unnecessary column or property value reads. Note that immutable datasets are often stored this way to optimize query performance in analytics. Furthermore, SmartSSD boosts this layout's effectiveness, improving query processing performance with Parquet datasets [8]. CORD-supported datasets include: **Tabular Datasets**: These datasets consist of records with a fixed number of columns, stored in either a row-based or columnar format. However, CORD is designed for columnar formats, where values of each column are stored sequentially, enabling efficient query processing. **Grid or Multi-dimensional Arrays**: Typically produced by scientific applications, these datasets map one-dimensional variables onto a multidimensional grid. Additional variables are stored as multidimensional arrays, with each value representing a specific property of a grid cell. The sequential storage of these variables on disk allows CORD to efficiently execute queries on them.

V. System Design

The CORD architecture has two main components: (i) the CORD client, which allows client programs to interact with the system through CORD-provided interfaces, and (ii) the CORD server, embedded within each storage node. For simplicity, we use "SN" to represent both the storage node and the CORD server within it, e.g., a client request handled by the CORD server will be referred to as being handled by the SN. Furthermore, "QP" stands for query processing, with any reference to a QP task or unit indicating a query processing task or unit. Figure 1 gives a simplified overview of the components of the CORD. After receiving a client request, the client handler of CORD client sends gRPC messages to each of the relevant SNs. Upon receiving a query processing request, the SN's request handler delegates the task to the appropriate execution engine (details in V-D). For each stripe[1] of the table, a QP task is added to the relevant CSD's local QP task queue. Each CSD has its own queue and a QPTaskController, which processes tasks one at a time through the QP unit, handling query execution for each stripe. The client and server components of CORD include an "MDT-Clone" that stores metadata about CSDs and files. One SN, the "MDT-Server",

[1]A distributed file, which contains a database "table", is divided into multiple fixed-size units called "stripes", which are distributed across CSDs.

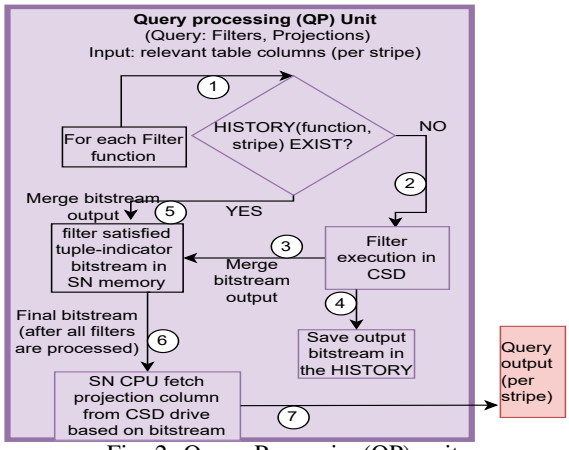


Fig. 2: Query Processing(QP) unit.

manages the master metadata, acting as a simplified version of the metadata server of a distributed file system. MDT-Clone is responsible for responding to clients for metadata-related requests. If it cannot satisfy the request locally, it passes the request on to the MDT-Server. CORD is implemented in C++ and uses SmartSSD's XRT native C++ API. In the following sections, we describe the functioning of this CORD architecture in more detail. First, we explain MDT-Clone in Section V-A and then how query processing is handled per stripe within an SN, as it is the unit of query processing in the system in Section V-B. Then we give details about the various "interfaces" that CORD supports in Section V-C. After that, we provide details on various "execution engines" that are responsible for handling different types of parallel query executions in Section V-D. Finally, we describe the "HISTORY" data structure, which plays a critical role in our query optimization process in Section V-E.

A. MDT-Server and MDT-Clone

The CORD's MDT-Server functions as a traditional metadata server, retrieving file stripe information. Since the system focuses on read-only, immutable datasets, their metadata remains unchanged after creation. The MDT-Clone is a replica of the master MDT-Server. The metadata stored includes details of the file stripe, such as location, size, and additional information. For example, if a stripe contains multiple column values, the metadata specifies the starting and ending offsets for each column and their data types. Similarly, for grid datasets, where stripes represent variable values over specific dimensional ranges, the metadata keeps information regarding these ranges. This allows execution engines to determine the relevance of the stripe for queries and retrieve only the necessary portions of a stripe. Metadata also tracks the number of CSDs in SNs, assigning each CSD a unique ID in the format "SN_id:CSD_id". Both clients and SNs can query their MDT-Clone for file stripes and CSD information.

B. Heterogeneous Execution of Query

In this section, we elaborate on query processing involving a "table stripe" that resides in a specific CSD. An overview of this process within an SN is illustrated in Figure 2. This

operation is managed by the QP unit of the SN. Initially, a query string received by the SN is *parsed* by a "Query String Parser" to distinguish "filter functions" from "projections". The subsequent steps of the QP unit are as follows:
① For each filter function, it is first checked whether "history" exists for that (function, stripe). Subsequent actions involve ② pushing down filter functions to CSDs unless they are already recorded in the "history" (refer to Section V-E for details). The relevant filter column or variable's data of the stripe is then transferred to SmartSSD's DRAM via an internal p2p link, allowing the data to be sequentially accessed and filtered efficiently in the FPGA. In the case of prescriptive execution (as detailed in Section V-D), if a stripe is not local, CORD's "Data Transfer Module" must fetch the data before sending it to the CSD for filter processing. ③ The FPGA kernel returns a bitstream to the SN's primary memory post-filter processing. An 'on-bit' in this bitstream denotes a tuple that satisfies the filter condition. ④ Concurrently, the returned bitstream is also saved in the "history". ⑤ On the other hand, if (function, stripe) exists in the "history", then the recorded bitstream is passed directly for post-processing. ⑥ Multiple filter functions are processed sequentially, and each resulting bitstream is merged with the prior ones based on the filter relationship (AND, OR) by the SN CPU to get the final bitstream. Finally, ⑦ the SN CPU retrieves the projection data based on the final bitstream from the SmartSSD, processing it accordingly.
The FPGA's single DRAM channel (AXI4) may impede performance due to non-sequential accumulation of projection data, thereby justifying the fetching of this data from CSD drive into SN memory by SN CPU. This architecture employs "heterogeneous query execution", leveraging *both* the SN CPU and the CSD's FPGA to optimize processing based on their respective efficiencies: *FPGA is ideal for accessing sequential data for filter processing due to its single-channel DRAM, while the SN CPU is better suited for handling the non-sequential data of projection results.* Thus, our approach harnesses *both* CPU and FPGA by directing tasks to the component where they are most efficiently executed. Moreover, as each data stripe is processed batch-by-batch in this heterogeneous execution, further optimization is achievable through *pipelining* between the filtering and projection phases. We simplify the management of these processes by allocating one SN CPU thread per CSD specifically for projection processing after the CSD's filter processing. We call this "heterogeneous execution", emphasizing that "query processing given to a CSD" implicitly involves CPU thread management for post-filter projection accumulation.

C. High-Level Interfaces for Programming CSDs

We propose novel interfaces to simplify CSD programming and parallelize applications, enabling easier orchestration of multi-CSD execution and coordinating data movements i) among CSDs, ii) between CSDs and SNs, and iii) between SNs and the client. Our proposed interfaces can be categorized into three broad types: 1) CSD discovery interfaces, 2) kernel push-down interfaces, and 3) execution interfaces. Through these interfaces, a client program can access CSDs on the storage side, push computations into the storage backend, plan an execution strategy, and then give the storage side full-control over the query processing. Thus, *these interfaces collectively make CSD programming easier and more accessible.*

1) Discovery Interfaces: The goal of this interface is to *expose* CSDs as "first-class compute engines" to the client program. To enhance accessibility to these CSDs, we implemented new interfaces: ***ndc_num_csds***, which returns the number of CSDs, and ***ndc_get_csds***, which populates a user-provided array with CSD descriptors that can be used, by execution interfaces, to specify the CSDs for offloaded query execution. Discovery interfaces, though conceptually similar to SNIA's [34] API model, are implemented within the CORD framework for seamless integration. Additionally, we built two APIs on top of these, to access related CSDs where the stripes of a file are currently stored.

2) Kernel Push-Down Interfaces: Client programs can use this interface to push down kernels to the CSDs. We utilize a filter kernel that runs on the FPGA of the CSD for our query processing. Therefore, we employ this interface to *transfer* the filter kernel bitstream to the MDT server. Subsequently, we establish an additional interface, ***ndc_push_kernels()***, to instruct the MDT server to load the kernel into a specific set of CSDs.

3) Execution Interfaces: Three novel interfaces are provided to the client to offload query processing to the multi-CSD-based storage plane. Depending on the specific interface used, the execution of query processing can vary radically.
Prescriptive Interface: This interface gives "full control" to the application programmer in executing query functions in compute engines (CSD or storage node CPU) and deciding which CSDs to store the output data. As a result, under this interface, the data movements are also decided by the programmer. This has five parameters: i) query string, ii) input file name, iii) output file name, iv) compute ID where query execution will happen, and v) target CSD ID, where the output file will be stored. We provide a simple example of a prescriptive call from a client program:

```
prescriptive_execution(select x from particles where ke >
    0.1, particles, query_result, Server1:CSD1, Server2:
    CSD3);
```

The above call specifies that the query is offloaded to CSD1 of 'Server1'. The "particles" table stripes can be distributed across all SNs. Therefore, the stripes must be transferred to CSD1 for execution, which may require data transfer across the network. Once the query is executed, CSD1 transmits the output to CSD3 of 'Server2'. This output data transfer is costly as it involves data transfers across the SNs over the network. In some cases, this may not be desirable, but there can be cases where it is (e.g., if the call that follows would operate in CSD3 on these output data, then sending it to CSD3 could be beneficial). However, in any case, under this interface, the responsibility of organizing query execution is given to the programmer.[2] The use of this programmer-driven prescriptive

[2]Akin, for example, to the case where explicit parallelism directives in many parallel execution platforms like OpenMP [35] gives the ultimate responsibility of parallel code execution to the programmer.

interface can result in substantial data transfers during query execution, depending on how the user exercises it. This is why we also provide a "Locality-aware (LA)" version of this interface, an example of which is as follows:

```
prescriptive_execution(select x from particles where ke >
    0.1, particles, query_result, LOCAL, LOCAL);
```

Here, the 'LOCAL' keyword poses special attention in the prescriptive call. Specifically, 'LOCAL', as an execution location, indicates that each CSD that has input table stripes must execute the query only on its local stripes, i.e., stripes that reside in the same CSD. On the other hand, 'LOCAL', as a destination for output data, instructs CORD that the result must be stored locally where it is generated. So, the above LA API call instructs that all CSDs that have the "particle" table stripes will take part in processing the query, and each of these CSDs will work, in parallel, on their local stripes and the processed output will be stored locally.

Descriptive Interface: Choreographing prescriptive calls may require significant user effort. To remove this pressure from the user, we provide a descriptive interface. *This interface moves the burden of optimization from the programmer to the runtime system.* Specifically, in this case, a user does not need to indicate where (on which CSDs) the query processing should be executed or where the output data should be stored in the call. Rather, such decisions are made by the runtime *automatically* by considering i) the current locations of involved data and ii) the current state of the CSDs. An example descriptive call is:

```
descriptive_execution(select x from particles where ke >
    0.1, particles, query_result);
```

Notice that the last two parameters of a prescriptive call are not included in the aforementioned call, as a descriptive call is required to specify *only* the query function that the programmer wishes to offload to CSDs *without* specifying the execution location for the computation and storage location (destination CSD) for the generated (output) data.

Predictive Interface: Prescriptive and descriptive interfaces are provided to control single query execution. However, there may exist "optimization opportunities" when multiple queries can be processed as a "batch", and in CORD such opportunities are captured by the use of the predictive interface which takes, as input, a batch of queries to be optimized together. For example, as will be detailed later in the paper, using the predictive interface, cross-query optimization opportunities – in a batch of queries – can be taken advantage of. Such optimizations, which could not be achieved if the queries are optimized/executed one by one, include, but are not limited to, out-of-order (OoO) execution of queries and inter-SN parallelism (executing disjoint queries on CSDs that are connected to different storage nodes).

D. Execution of Query Processing across Multiple CSDs

We now go into the details of how our execution-related interfaces are implemented in the SN (as shown in Figure 1), that is, how a given query is executed and data movements are performed under a given interface. When a client calls any of our execution-related interfaces for query offloading, this call is transformed into one or multiple "RPC calls" to relevant SNs. If the user program specifies a CSD in the prescriptive call, then one RPC call will be sent to the SN that has the CSD. In cases where a specific compute CSD is not specified, the CORD client will initially verify the name of the input table and obtain information on all storage nodes where data stripes of that table are distributed. Subsequently, it will begin sending RPC request calls to each relevant SN. The SN first processes the requested query string by a parser. It extracts three pieces out of the query string: i) filter functions, ii) projection functions, and iii) table or file name. Then it checks the metadata of the input file name and finds the stripe information of that file. After that, the flow of execution varies depending on the specific call type being processed. We elaborate below on our three (execution-related) interfaces.

1) Prescriptive Call Execution: Each SN is equipped with two specialized threads managed by the "Data Transfer Module": one thread handles the transfer of data stripes between storage nodes for query processing, while the other manages the output data movement from the execution CSD to the designated remote target location. This inter-node data transfer occurs batch-by-batch as needed. For instance, consider the execution flow of the following call:

```
prescriptive_execution(select x from particles where ke >
    0.1, particles, query_result, Server1:CSD1, Server2:
    CSD3);
```

Initially, Server1 initiates RPC calls to other storage nodes to fetch remote data stripes of the specified table. The stripes are sent batch-by-batch from their respective servers' SSDs to Server1, which then relays the relevant filter-related columns or variable values from these stripes to the DRAM of CSD1 for processing. CSD1 also processes local data stripes. Once processed, the projection output data is transmitted via RPC to Server2, where it is stored in CSD3, with the output file's metadata managed by the MDT server. If the execution call specifies 'LOCAL' for both compute and output locations, each node receives RPC requests from the client, and relevant CSDs process local data stripes *in parallel*, storing the output back on the originating CSDs. This execution framework is entirely programmer-centric, without any runtime system-initiated optimization, contrasting with the adaptive strategies employed by the descriptive and predictive interfaces.

2) Descriptive Call Execution: In the descriptive execution provided by CORD, the programmer does *not* specify the compute location; instead, the system optimizes the locality of the data and the parallelism of execution within the limitations of intra-SN communication. This strategy aims to balance the workload effectively, employing a "work-stealing" concept for idle CSDs within the same node to maximize parallel execution. For example, if one CSD finishes its workload significantly faster than another within the same SN due to, say, uneven data distribution, it can *steal* work from the slower CSD, thus maintaining efficient processing rates across the node. Our motivation for this approach comes from the P2P feature of SmartSSD through which data transfer from one CSD's drive to another CSD's FPGA DRAM can be directed using XRT (Xilinx Runtime) without going through the primary memory. Since this communication is no longer

costly due to P2P, idle CSDs can work on data from other active CSDs to maximize parallel execution within the same storage node. **QPTaskController** of relevant CSDs ensure descriptive execution in the above mentioned way. The descriptive engine algorithm and QPTaskController algorithm are given in Algorithm 1 and Algorithm 2, respectively.

Algorithm 1 Descriptive Query Execution

```
1: procedure DESCRIPTIVEENGINE(queries)
2:     tablename ← queries[0].tablename
3:     localStripes ← GETLOCALSTRIPES(tablename)
4:     if localStripes.size() == 0 then
5:         return
6:     end if
7:     Initialize Semaphore semId(0)
8:     for j ← 0 to localStripes.size() − 1 do
9:         stripe ← localStripes[j]
10:        csdId ← GETCSDID(stripe)
11:        taskId ← CREATEQPTASK(queries, stripe, STEALABLE)
12:        ADDTASKTOQPTASKQUEUE(csdId, taskId, semId)
13:    end for
14:    for j ← 0 to localStripes.size() − 1 do
15:        SEM_WAIT(semId)
16:    end for
17: end procedure
```

Algorithm 2 QPTaskController (per CSD)

```
1: procedure QPTASKCONTROLLER(csdId)
2:     while True do
3:         if Local QP task queue is not empty then
4:             taskId, semId ← localQPTaskQueue.pop()
5:             EXECUTEQPTASK(taskId)
6:             SEM_POST(semId)
7:         else
8:             stealedTaskId ← STEALQPTASK()
9:             if stealedTaskId ≠ NULL then
10:                stealedTaskId is put in local QP task queue
11:            else
12:                Wait Sometime for Tasks
13:            end if
14:        end if
15:    end while
16: end procedure
```

For each stripe of the table, the descriptive engine creates a **STEALABLE** task and puts that task in the stripe-owner CSD's local QP task queue initially, to prefer locality-aware execution. The QPTaskController of each CSD continuously picks up a task from its task queue and works on it. Whenever the local task queue is empty, QPTaskController checks other CSDs' task queues and tries to find the CSD that has the most pending "stealable" tasks (it is important to note that tasks generated by a prescriptive engine are *not* stealable). If such a CSD is found, it will pop a task from that CSD's task queue, thus *stealing* it and putting it on its queue for processing. **STEALQPTASK** method is responsible for this stealing. Once QP tasks are placed in their respective queues, the descriptive engine remains blocked until all QPTaskControllers complete their tasks. This synchronization is managed using a counting semaphore: **SEM_WAIT** (line 15 in Algorithm 1) blocks execution until the QPTaskController signals task completion through **SEM_POST** calls (line 6 in Algorithm 2). To guarantee exclusive access and updates, a mutex is kept for every QP task queue (not included in the algorithms for simplicity). In descriptive execution, the output result can either be sent back to the user (client) or stored as stripes of the output file in the execution CSD.

3) Predictive Call Execution: Predictive execution occurs for a batch of queries, instead of a single query. This execution enables system optimizations that go beyond single-query execution. It is possible to identify numerous optimization opportunities by examining the data-interrelationship of the queries in the batch. In the worst case, when there is no such (inter-query optimization) opportunity, this call is just transformed into sequential descriptive executions of the queries in the batch. We propose two optimization opportunities for predictive execution: i) *inter-SN parallelism* and ii) *out-of-order (OoO) execution*, which are elaborated in the following. Our framework puts a "predictive engine" in each SN as shown in figure 1 to efficiently handle predictive calls. Our algorithm for the predictive engine is given in Algorithm 3.

Inter-SN Parallelism: As an illustration, let us assume that the file stripes of two distinct tables, T1 and T2, are distributed across distinct sets of SNs: stripes of T1 are located in SN1, while stripes of T2 are located in SN2. Parallel processing of two queries, Q1 and Q2, in the same batch, is feasible, provided that Q1 is associated with T1 and Q2 is associated with T2. Note that this enables parallel execution at different SNs as well as different CSDs connected to an SN. Suppose, on the other hand, these queries are not put in as a batch predictive call and, instead, two distinct descriptive calls are issued for these queries. In that case, the regular descriptive call execution can ensure inter-CSD parallel execution (intra-SN) but *not* inter-SN parallelism because SN2 would remain idle when Q1 is being processed in SN1, and similarly, SN1 would remain idle when Q2 is being processed. To make sure that storage nodes can run multiple queries at the same time during a batch predictive call, our design suggests that, if a query's data stripes are not present in a storage node, that storage node's predictive engine should leave that query out of the batch. Therefore, in the preceding illustration, the predictive engine of SN1 eliminates Q2 from its batch execution, while the predictive engine of SN2 eliminates Q1. Then, both SNs *simultaneously* execute distinct queries, thereby guaranteeing parallel execution at the storage node level. Line 5 in Algorithm 3 satisfies this design approach.

Out-of-Order(OoO) Execution: Queries in a batch can be *reordered* to achieve data reuse benefits. If two queries need to work on the same filter data or projection data, then whenever data are fetched into CSD memory for processing one query, the other query can *reuse* those data. This data reuse benefits us by reducing I/O requests and preventing relevant execution stalls. Figure 3 gives an example of a predictive batch call that highlights the opportunity for OoO execution.

The diagram shows a set of batch queries that present significant data reuse opportunities, particularly across the 1st, 3rd, and 6th queries which all utilize a filter function on the 'ke' column. This configuration allows the 'ke' data fetched from the disk during the first query to be reused in the subsequent

Algorithm 3 Predictive Batch Query Execution

```
1: procedure PredictiveEngine(queries)
2:     for i ← 0 to queries.size() − 1 do
3:         tablename ← queries[i].tablename
4:         if GetLocalStripes(tablename) == ∅ then
5:             queries.remove(i)
6:         end if
7:     end for
8:     while queries.size() ≠ 0 do
9:         newGroup ← ∅
10:        newGroup.add(queries[0])
11:        while newGroup.size() < GROUPSIZELIMIT do
12:            maxReuseScore ← THRESHOLD − 1
13:            probableQueryid ← −1
14:            for i ← 1 to queries.size() − 1 do
15:                if queries[i] is in newGroup then
16:                    continue
17:                end if
18:                reuseScore ← DataReuseScore
                                    (newGroup, queries[i])
19:                if maxReuseScore < reuseScore then
20:                    maxReuseScore ← reuseScore
21:                    probableQueryid ← i
22:                end if
23:            end for
24:            if maxReuseScore < THRESHOLD then
25:                break
26:            else
27:                newGroup.add(queries[probableQueryid])
28:            end if
29:        end while
30:        async DescriptiveEngine(newGroup)
31:        queries.removeAll(newGroup)
32:    end while
33:    Blocked until all asynchronous calls are completed
34: end procedure
```

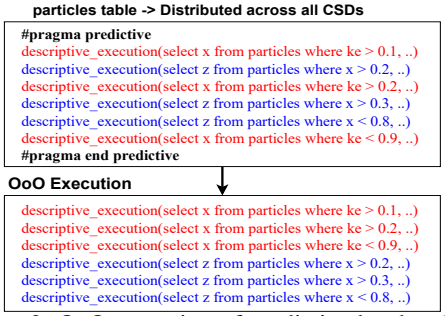


Fig. 3: OoO execution of predictive batch call.

two queries without additional I/O operations. Additionally, these queries share the projection column 'x', emphasizing the efficiency of OoO processing, despite their "non-sequential" order to maximize data reuse. Processing these queries as a group (stripe-by-stripe) is essential, given the impracticality of retaining all 'ke' and 'x' data in FPGA DRAM memory and primary memory, respectively. Similarly, the other three queries, which involve projection on column 'z' and filter functions on column 'x', also demonstrate the data reuse potential and should be grouped for processing. Our predictive engine enhances this approach by employing the following methodology, as illustrated in lines 8–33 of Algorithm 3: i) Queries with high data reuse scores are grouped. Data reuse score between two queries involves increasing the score by 10 for each filter function on the same column and by 1 for projection functions on the same column. The filter function is prioritized in scoring due to the history data structure described in Section V-E. The total data reuse score for an ungrouped query relative to a new group is calculated as follows:

$$DataReuseScore(newgroup, ungroupedQuery) = \sum_{i=1}^{k} DataReuseScore(newgroup(query_i), ungroupedQuery)$$

(ii) The query with the highest data reuse score surpassing a user-defined THRESHOLD is incorporated into the new group. If no such query exists or the group reaches the size limit, the group formation concludes. (iii) The formed groups are then scheduled for group-based execution by the descriptive engine, and the process repeats until no ungrouped queries remain.

E. Taking Advantage of the Filter History

In CORD, filter processing occurs within the CSD's FPGA, and the output for a processed column or variable is a "bitstream", where each bit's position corresponds to a tuple or a grid cell, and an on-bit indicates that it meets the filter condition. By storing each filter function's bitstream output in a hashed data structure, we can circumvent the need for re-executing filters in the CSD under two conditions. Firstly, if the bitstream for a given filter function on a specific column is already saved from previous processing, it can be retrieved directly from this history, eliminating the need for re-processing. Secondly, for complementary filter functions, such as changing from 'ke $>=$ 0.1' to 'ke $<$ 0.1', we can employ a "bitwise trick" by *inverting* the bits of the previously saved bitstream for 'ke $>=$ 0.1'. As the filters are opposite, inverting the bits in the bitstream of one represents the other. The important point to note is that this method effectively *reuses* the filter results *without* executing the filter anew in the CSD, showcasing a significant optimization in filter processing by leveraging "history". This custom filter function history not only saves CSD compute resources, but also avoids data fetching from the drive, thus speeding up the overall execution. An example case in which subsequent queries can utilize history (formed by the previous queries) to prevent filter execution (and reduce I/O) can be illustrated as follows:

```
select x from particles where ke >= 0.1
select x, y from particles where ke >= 0.1
select x, y from particles where z >= 0.1
select x, y from particles where ke >= 0.1 and z >=  0.1
select x, y from particles where ke < 0.1 and z < 0.1
```

In this case, subsequent queries can benefit from the execution history of previous filters, and this is particularly evident when conditions such as 'ke $>=$ 0.1' and 'z $>=$ 0.1' are processed. This allows the last two queries in the set to leverage filter histories. Although the filter functions in the last query are not exact, employing bitwise operations on saved histories can help bypass redundant filter executions, where the extent of speedup varies depending on the query and existing history. Our hashed data structure has been designed and implemented to optimize such scenarios. For each filter condition like 'ke $>=$ 0.1', a unique key is generated per stripe in the format: $stripe\_id\_ColumnID\_Comparator\_Value$ and the value is the bitstream of the filtered results for that stripe. To verify the presence of a filter's result for a specific stripe, a key is constructed in the same format and searched

using 'find(key)'. To manage memory efficiently and address capacity issues, the size of the hashmap can be adjusted by users through the settings.xml file, and an LRU (Least Recently Used) policy manages the replacement of "history" entries upon reaching the maximum capacity. In particular, each SN maintains its "local history" for local data stripes, eschewing centralized history management. *Our system architecture facilitates both descriptive and predictive executions by utilizing this history, which particularly enhances performance in predictive scenarios due to group-based execution.* Here, queries operating on the same data stripe often share the same/similar set of filter functions, enabling a single query within a group to benefit from the history established by other queries, thus reducing redundant computations significantly and improving overall efficiency. History is maintained at the granularity of the stripe. This means that if a filter function applies to only a subset of data within a stripe, its history for that stripe is not stored. This scenario can occur during query processing on grid datasets, where a stripe spans a range of dimension values, but the query overlaps only a portion of the data in the stripe, rather than the entire stripe.

VI. Evaluation and Results

A. Experimental Setup

System Topology: Our experimental topology consists of two storage nodes (SNs), each equipped with two SmartSSDs (SN1 with CSD1 and CSD2, and SN2 with CSD3 and CSD4). A compute/client node runs client programs with all machines connected through a local area network. The maximum data transfer rate per machine is 1 Gbps, as network switches offer a maximum bandwidth of 1 Gbps per port, which is consistent with the bandwidth of the machine NIC. All machines are running Ubuntu 20.04.6 LTS. SN1 is equipped with an Intel® Core™ i7-8700K CPU @3.70GHz, SN2 with an Intel® Xeon® CPU E5-1620 v3 @3.50GHz, and the client node with an Intel® Core™ i5-7500 CPU@3.40GHz. Despite the client node having less computational power compared to the storage nodes, this difference is negligible in our experiments, as the data transfer bottleneck between the storage nodes and the client node overrides the lower compute power. In our setup, all table files are striped, and the stripes are distributed across CSDs. In the plots presented below, "full-system" means that table stripes are distributed across all 4 CSDs.

Workload Details: The workload primarily involves querying large, read-heavy tables stored in a columnar format. These queries typically necessitate data transfers from the storage nodes to the client node for processing, with this data transfer serving as a major bottleneck that significantly impacts overall query performance. We use a distributed file representing a 'particles' table in columnar format, derived from the VPIC simulation binary files [36]. This table is distributed across multiple files, each corresponding to a stripe of the particles table, resulting in a total data size of approximately 122GB. The purpose of this benchmark is to comprehensively evaluate the behavior of our proposed system from multiple perspectives. In addition to this benchmark, we evaluate CORD on two other datasets to show its applicability on different datasets. The first is a grid-based dataset from the CMIP5 project, produced using the NCAR Community Earth System Model [37]. The second dataset is derived from the TPC-H suite [38]. Specifically, we generated the 'LINEITEM' table from the TPC-H benchmark using a scale factor of 200 and evaluated CORD on TPC-H Q6.

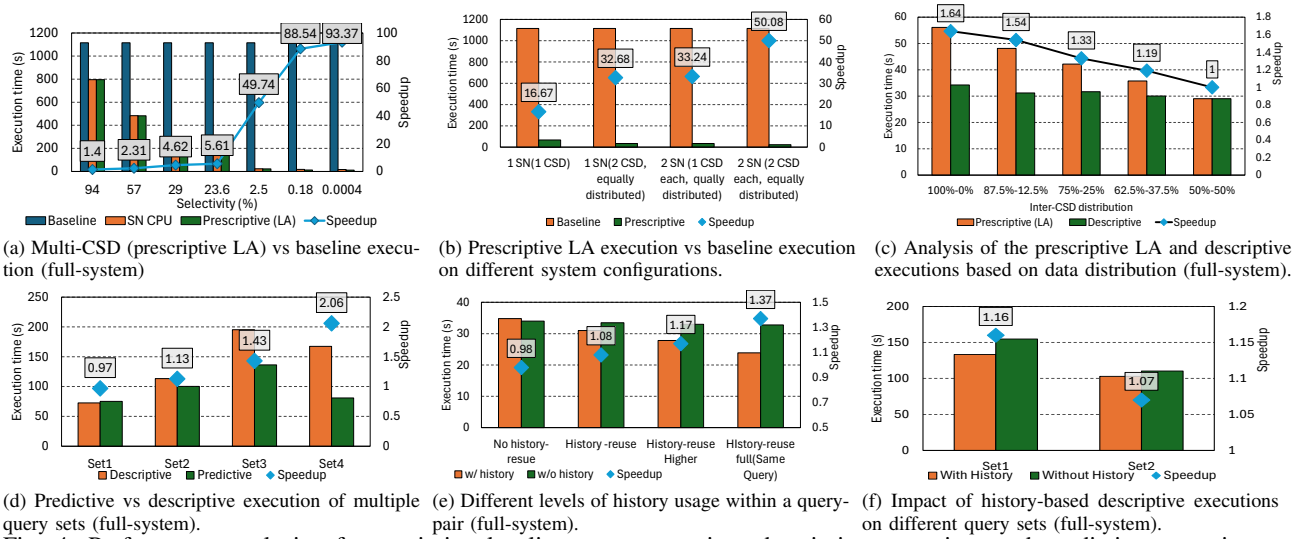
B. Baseline Design

Our baseline system emulates traditional query processing in a distributed storage environment, where the query engine running on the client (host) node fetches all the data from the storage node to the client node. Each storage node runs a server-side gRPC instance that facilitates data transmission, while the client-side gRPC component running in the client node initiates data fetch requests to the storage nodes. The client-side parser first processes the query string, and then sequentially fetches filter and projection column data batch-by-batch by issuing gRPC calls to each of the relevant storage nodes. Upon receiving a request, each server node retrieves table stripes from its local CSD and transmits them back to the client as gRPC messages. The client, after accumulating a batch of table data, initiates filter processing and performs projection operations. The processed data are retained in their memory for subsequent processing or can be routed back to the storage back-end as a new persistent distributed file.

C. Experimental Results

To evaluate the effectiveness of CORD, we use our proposed prescriptive, descriptive, and predictive interfaces to offload queries from the client node to the storage back-end equipped with multiple SmartSSDs. On the other hand, our baseline client program performs query processing in the client node.

*1) **Baseline vs CSD-Based Query Execution**:* One important analysis from the VPIC simulations is answering the question: *What is the spatial distribution of highly energetic particles?* [2]. To address this, we execute multiple queries in the following format on both the baseline client program and our CORD client program, which offloads processing using the prescriptive interface with locality-aware parallel CSD execution: "**select x, y, z from particles where ke $>$ VAL**". Note that different "VAL" values in the queries result in varying "selectivities" in the output result. To observe data transfer overheads during execution time, the output result is returned to the client program in *both* types of runs. Figure 4a gives the comparison of the total execution time of queries based on different selectivities. The baseline execution time remains constant regardless of filter selectivity because it consistently fetches data from the storage backend. With a network speed of 1 Gbps, the entire 122 GB of data is transferred to the client node, making data transfer the constant bottleneck that hinders overall execution, irrespective of the filter function's 'VAL'. On the contrary, the time taken by our proposed multi-CSD execution (CORD) is a function of the filter selectivity because the output data that needs to be sent back to the client *varies* based on the filter selectivity. Specifically, if selectivity ratio is higher for a certain 'VAL', it produces a larger projection



(a) Multi-CSD (prescriptive LA) vs baseline execution (full-system)

(b) Prescriptive LA execution vs baseline execution on different system configurations.

(c) Analysis of the prescriptive LA and descriptive executions based on data distribution (full-system).

(d) Predictive vs descriptive execution of multiple query sets (full-system).

(e) Different levels of history usage within a query-pair (full-system).

(f) Impact of history-based descriptive executions on different query sets (full-system).

Fig. 4: Performance analysis of prescriptive locality-aware execution, descriptive execution, and predictive execution on 'particles' dataset.

data and thus requires more time to send to the client program over the network. This is why the speedup increases with the decrease in filter selectivity ratio for the same reason. Thus, offloading the query to multiple CSDs can provide a maximum speedup of 93× (over the baseline) for a 0.0004% selectivity ratio(high-selectivity query), and at least a speedup of 1.4× for a 94% selectivity ratio.[3] In this graph, we also compare our setup with the Skyhook [31] "SN CPU execution", where query processing is entirely offloaded to the SN CPUs. The performance is similar to CORD, primarily because the filtering operation is relatively smaller compared to the projection. As we explain in Section V-B, CORD handles projection on the SN CPU while performing filtering in the CSD. Therefore, the benefits of CSD acceleration are not apparent for this particular query. If the filtering were more compute-intensive, we would likely observe significant improvements in CORD compared to the Skyhook version. Although CORD shows only slight performance gains, Skyhook's higher SN CPU utilization—which is undesirable in many systems—is notable (though not shown here, as it has already been demonstrated in POLARDB [30]).

Performance of Prescriptive Locality-Aware (LA) Execution on Different System Configurations: To evaluate the usability and performance of the prescriptive (locality-aware) execution on various system configurations, we executed the same query "**select x, y, z from particles where ke $>$ 0.2**" on different system configurations – the number of SNs ranging from 1 to 2 and the CSD count ranging from 1 to 4. The selectivity ratio of the above query is 2.5%. Figure 4b plots the performance of the prescriptive (locality-aware version) call over the baseline. Prescriptive execution is approximately 16× faster than the baseline in a 1 SN and 1 CSD scenario. A two-fold performance boost is observed when increasing the CSDs from 1 to 2, highlighting the improved effectiveness of the prescriptive call with more CSDs. Furthermore, even if we do not raise the CSD numbers in an SN but instead increase the SN from 1 to 2, with one CSD in each SN, this still results in a speedup of about 33× faster than the baseline and a factor of 2 faster than in the single SN-single CSD case. Therefore, the speedup of our prescriptive execution scales reasonably with the number of SNs present in the system. Our complete system, consisting of 2 SNs and 2 CSDs per SN, achieves a maximum speedup of approximately 50× over the baseline through parallel execution across all 4 CSDs.

*2) **Prescriptive vs Descriptive Execution**:* Each storage node can have a variable number of CSDs connected to it and a file may be distributed across storage nodes in a balanced fashion based on the stripe placement logic. However, data distribution across CSDs in a storage node can likely be unbalanced. In that case, prescriptive locality-aware execution may not utilize all the CSD compute efficiently, and in fact, it may be difficult for the programmer to organize computations and data movements. In such cases, the descriptive interface can be very useful as it can perform optimized execution without much programmer involvement. To test our descriptive execution, we process this query: "**select x, y, z from particles where ke $>$ 0.1**". Figure 4c shows that the descriptive execution works better than the prescriptive execution for different types of data distribution (across CSDs in the same storage node). Our evaluation indicates that the prescriptive execution becomes less effective with uneven data distribution. In contrast, descriptive execution optimizes CSD performance and shows increased speedup with greater data unevenness. However, both execution types perform similarly under balanced data distribution. Note that Figure 4c compares the descriptive execution performance against the prescriptive LA execution, while other programmer-driven prescriptive

[3]We would like to emphasize that our proposed multi-CSD solution is generic and can work with any number of CSDs and any number of SNs.

executions are excluded from this analysis.

*3) **Predictive Execution vs Descriptive Execution**:* Recall from our earlier discussion in Section V-D3 that the predictive interface is designed to perform optimizations across a batch (group) of queries. To have a fair analysis of the predictive interface and quantify the benefits of the predictive execution over the descriptive execution of a batch of queries, we generated 4 types of sets: **Set1**: There is no data reuse and no parallel SN execution opportunity across the queries in the set. **Set2**: There exist data reuse opportunities across the queries in the set. **Set3**: There are queries in this set that can be executed in parallel on different SNs. **Set4**: Both data-reuse and parallel SN execution opportunities exist across the queries. We run each of these sets by using separate descriptive calls for each query in the set and then we use one predictive call for each set to execute the whole set as a batch. Figure 4d plots the performance of the predictive call over the descriptive execution. Predictive execution is unable to improve its performance in Set1 because there is no opportunity for optimization and the predictive engine brings some overheads. In fact, in this case, the prediction execution is slower than executing the queries one by one. Set2 and Set3 exhibit improvements due to the opportunities that predictive execution is designed to exploit. Parallel query execution on multiple storage nodes manifests a greater speedup than data-reuse exploitation in terms of the total execution time. Although the speedup in execution time is not as great when data reuse is used, it does reduce the system's overall disk I/O, which is not assessed here. Finally, Set4 poses the best-case scenario where the predictive engine can exploit both OoO and inter-SN execution. It can be seen that the predictive execution gets 2× speedup for Set4. So, the more opportunities there are for data reuse and parallel SN execution, the higher the speedup in the case of predictive execution.

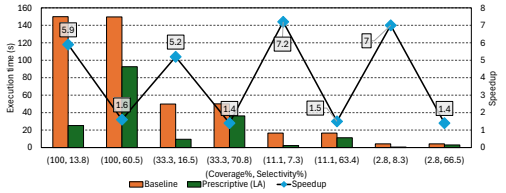
*4) **Impact of Custom History in Descriptive Execution**:* We create four pairs of queries with varying levels of history utilization, each pair having various filter functions, to demonstrate the benefits of history on system-optimized descriptive execution and evaluate its overall efficacy. Figure 4e plots our experimental results on history performance. In this experiment, the outcomes of history-based performance are compared to those of non-history-based performance in descriptive executions. The plot in Figure 4e indicates that there is no advantage in utilizing history when the two queries are different; instead, it takes time to save the filter functions for each query in the history data structure. Because of this, non-history-based runs marginally better. However, if there are similarities between the filter functions, the second query can *use* history, instead of executing the filter function. The plot indicates that, in certain circumstances, history-based running is faster, and that the rate of speedup increases with reuse. Both the queries in the last pair are the same. Therefore, the second query does not perform any filter functions, and all of the filter results are derived from the history created by the first query. The speedup over non-history-based descriptive execution is around 1.37×. All the cases use a set of two queries to demonstrate how varying overlap levels in filter functions between queries can lead to speedups from history. However, if we have multiple queries in the set, the speedup due to history usage is always dependent on the similarities across the queries. We also tested two sets of queries (each set has 6 queries) to check the history performance. Figure 4f shows the history performance for these two sets of queries. As can be observed from this plot, Set2 has lower speedup than Set1 because the history utilization is lower in Set2.

*5) **Experiments with Multidimensional Dataset**:* We tested CORD using a 3D grid-based dataset [37] with dimensions for timestamp, latitude, and longitude. The latitude ranges from -90 to +90 with 192 values; the longitude ranges from 0 to 360 with 288 values; and the timestamp covers 3-hour intervals over 30 years (0 to 78,840). The dataset's variable, "Total Cloud Fraction (CLT)", is stored as a distributed dense 3D array, with each stripe containing data for specific ranges of timestamps, latitudes, and longitudes. We experimented with queries to identify events by selecting grid cells where CLT values fall below a specified threshold. These queries, expressed as "**select time, lat, lon from dataset where clt** $<$ **threshold AND lat in [lat1, lat2] AND lon in [lon1, lon2]**", are characterized by two metrics, coverage and selectivity, as defined in FastQuery [39]. Coverage represents the proportion of the dataset accessed by the query, while selectivity denotes the fraction of the accessed data that meets the condition CLT $<$ threshold. Figure 5a presents the performance of the prescriptive locality-aware execution over the baseline across various queries (coverage and selectivity). Multi-CSD execution performs better for highly selective queries (low selectivity ratio), achieving up to a 7× speedup. However, the speedup of CORD over the baseline is slightly less pronounced compared to columnar datasets. This is because, in grid datasets, the baseline execution also avoids fetching data stripes whose dimension ranges fall outside the query range. The performance comparison between descriptive execution and prescriptive locality-aware execution, across various data distributions, is also presented in Figure 5b. Descriptive execution consistently outperforms prescriptive LA execution for all queries. Under highly uneven data distributions, descriptive execution achieves a speedup of 1.5× to 1.85×, depending on the characteristics of the query (coverage, selectivity).

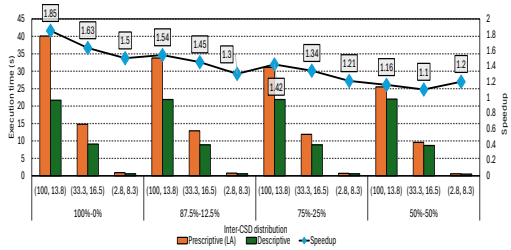
*6) **Experiments with the TPC-H Query 6**:* As in a previous study [26], from the TPC-H benchmark, we used Query 6. With the predicates in query 6 set to default values for SHIPDATE, DISCOUNT, and QUANTITY, our query looks like this:

```
SELECT SUM (EXTENDEDPRICE*DISCOUNT) FROM LINEITEM WHERE
    SHIPDATE >= 1994/01/01 AND SHIPDATE < 1995/01/01 AND
    DISCOUNT > 0.05 AND DISCOUNT < 0.07 AND QUANTITY < 24
```

We tested this query with both a prescriptive locality-aware API and a system-optimized descriptive API from the client program, under different data distributions of the LINEITEM table across the CSDs in SN1. The selectivity ratio of this query is 0.6%. Consequently, the output data sent to the client program is much lower (in volume) than the input data. The product and summation calculations are carried

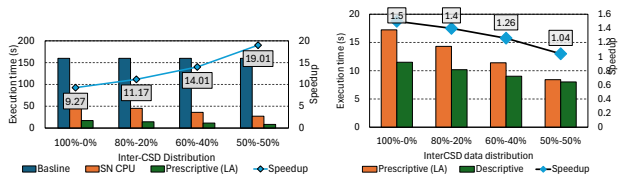


(a) Multi-CSD prescriptive LA execution vs baseline.



(b) Descriptive vs prescriptive LA execution.

Fig. 5: Performance analysis of prescriptive locality-aware (LA) execution and descriptive execution on grid-based climate dataset (dataset is distributed across 2 CSDs in SN1).



(a) Multi-CSD prescriptive LA execution vs baseline.

(b) Descriptive vs prescriptive LA execution

Fig. 6: Performance analysis of prescriptive locality-aware (LA) execution and descriptive execution using query 6 from TPC-H suite (dataset is distributed across 2 CSDs in SN1).

out on the client side, as those parts are not offloaded in our framework. Figure 6a presents the performance of the prescriptive locality-aware execution over the baseline. We see that the prescriptive execution works better when data distribution is even across the CSDs, because, in that case, the maximum parallel execution happens. Since we have 2 CSDs in SN1, the prescriptive execution shows a maximum 19.01× speedup over the baseline execution when the table data are evenly distributed across SN1's CSDs. Additionally, the SN CPU execution (Skyhook version) performs poorly compared to CORD for this query, as it is filter-heavy. In this case, CSD acceleration significantly improves performance compared to execution solely on the SN CPU. Figure 6b plots the performance of descriptive execution for the same distribution sets. This comparison is done with prescriptive locality-aware (LA) execution. The result shows the same trend that we see in the experiment with the 'particles' table. Descriptive execution performs better than prescriptive LA execution when the data distribution is uneven.

VII. Limitations and Opportunities

CORD is a CSD-based filter-projection query processing framework designed for distributed and disaggregated storage environments with multiple CSDs. It is optimized for highly selective queries commonly found in scientific applications. The framework has been evaluated using columnar-formatted tabular datasets, grid-based multidimensional array datasets, and TPC-H Q6, all involving highly selective queries, demonstrating its efficiency in both scientific and decision-support workloads. Although CORD is not initially designed for row-based datasets, its modular architecture allows seamless adaptation. Users can develop custom filter kernels for row-based data and leverage its kernel push-down interface. By focusing on simple filter-projection queries, CORD simplifies execution by handling highly selective components first, offloading them to the CSDs/storage layer, and leaving complex operations to database engines running on compute nodes. Integration with more complex datasets or queries requires minimal modifications, limited to updating stripe-based QP units, as its generic execution engines remain mostly unaffected. The CORD metadata tracks file stripes and their properties, enabling efficient filtering and retrieval of specific data stripes for SmartSSD processing. This design ensures flexibility in handling diverse datasets and storage environments. In distributed storage, file stripes are managed by the file system, while formats such as NetCDF and HDF5 maintain their internal metadata. CORD needs combined information from these metadata sources for stripe-based QP tasks. Thus, compatibility with such formats can be achieved by incorporating a shim layer on the CORD client, which maps internal file metadata to file stripes, integrates the information, and ensures accurate and efficient query processing.

CORD's MDT server or clone does not explicitly manage metadata synchronization or availability issues, as these are typically handled by distributed file system. Instead, it assumes that the MDT server contains the latest information, which can be queried to determine the locations of stripes and their corresponding CSD owners. CORD currently uses history to avoid repetitive CSD computations, but this history may become outdated when data is updated. As our current implementation focuses on read-only workloads, we have not addressed this issue. However, managing updates is part of our plans as we aim to support read-write workloads.

VIII. Concluding Remarks

This paper presents CORD, an innovative framework that enhances query processing efficiency across multiple Computational Storage Devices (CSDs). By offloading query tasks to CSDs and efficiently parallelizing their execution, CORD minimizes data movement and reduces latency, achieving speedups of up to 93× compared to traditional methods. CORD is designed to be flexible and modular, making it easily extendable and adaptable for integration into any data-intensive application where each stripe can be processed independently.

Acknowledgment

This work was supported in part by DOE award DE-SC0023186 as well as NSF awards 1931531 and 1908793.

REFERENCES

[1] J. Gu, S. Klasky, N. Podhorszki, J. Qiang, and K. Wu, "Querying large scientific data sets with adaptable io system adios," in *Supercomputing Frontiers: 4th Asian Conference, SCFA 2018, Singapore, March 26-29, 2018, Proceedings 4*. Springer International Publishing, 2018, pp. 51–69.

[2] S. Byna, J. Chou, O. Rubel, H. Karimabadi, W. S. Daughter, V. Roytershteyn, E. W. Bethel, M. Howison, K.-J. Hsu, K.-W. Lin *et al.*, "Parallel i/o, analysis, and visualization of a trillion particle simulation," in *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*. IEEE, 2012, pp. 1–12.

[3] Q. Zheng, C. D. Cranor, D. Guo, G. R. Ganger, G. Amvrosiadis, G. A. Gibson, B. W. Settlemyer, G. Grider, and F. Guo, "Scaling embedded in-situ indexing with deltafs," in *SC18: International Conference for High Performance Computing, Networking, Storage and Analysis*. IEEE, 2018, pp. 30–44.

[4] R. Sethi, M. Traverso, D. Sundstrom, D. Phillips, W. Xie, Y. Sun, N. Yegitbasi, H. Jin, E. Hwang, N. Shingte, and C. Berner, "Presto: Sql on everything," in *2019 IEEE 35th International Conference on Data Engineering (ICDE)*, 2019, pp. 1802–1813.

[5] J. Camacho-Rodríguez, A. Chauhan, A. Gates, E. Koifman, O. O'Malley, V. Garg, Z. Haindrich, S. Shelukhin, P. Jayachandran, S. Seth *et al.*, "Apache hive: From mapreduce to enterprise-grade big data warehousing," in *Proceedings of the 2019 International Conference on Management of Data*, 2019, pp. 1773–1786.

[6] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica, "Spark: Cluster computing with working sets," in *2nd USENIX workshop on hot topics in cloud computing (HotCloud 10)*, 2010.

[7] Samsung, "Samsung smartssd computational storage drive," https://download.semiconductor.samsung.com/resources/brochure/Samsung%20SmartSSD%20Computational%20Storage%20Drive.pdf, 2024, (Accessed on 10/10/2024).

[8] AMD, "Smartssd® computational storage drive," 2023. [Online]. Available: https://www.xilinx.com/content/dam/xilinx/publications/product-briefs/xilinx-smartssd-computational-storage-drive-product-brief.pdf

[9] 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)*, 2023, pp. 132–144.

[10] ScaleFlux, "Csd 3000," https://scaleflux.com/products/csd-3000/, 2024, accessed on 07/13/2024.

[11] J. H. Lee, H. Zhang, V. Lagrange, P. Krishnamoorthy, X. Zhao, and Y. S. Ki, "SmartSSD: FPGA Accelerated Near-Storage Data Analytics on SSD," *IEEE Computer Architecture Letters*, vol. 19, no. 2, pp. 110–113, Jul. 2020, conference Name: IEEE Computer Architecture Letters. [Online]. Available: https://ieeexplore.ieee.org/abstract/document/9141369?casa_token=CHPzBnJXvHYAAAAA:oUXw8TOyQbIgvC3k0TMj_3kEh4n_x2JQhggonQDxVmsb23iV6mXCNeTL93-xB24ZWcI4h5akkv8

[12] V. Lagrange Moutinho dos Reis, H. Li, and A. Shayesteh, "Modeling analytics for computational storage," in *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, 2020, pp. 88–99.

[13] E. Riedel, C. Faloutsos, G. A. Gibson, and D. Nagle, "Active disks for large-scale data processing," *Computer*, vol. 34, no. 6, pp. 68–74, 2001.

[14] D. Tiwari, S. Boboila, S. Vazhkudai, Y. Kim, X. Ma, P. Desnoyers, and Y. Solihin, "Active flash: Towards Energy-Efficient, In-Situ data analytics on Extreme-Scale machines," in *11th USENIX Conference on File and Storage Technologies (FAST 13)*. San Jose, CA: USENIX Association, Feb. 2013, pp. 119–132. [Online]. Available: https://www.usenix.org/conference/fast13/technical-sessions/presentation/tiwari

[15] A. Acharya, M. Uysal, and J. Saltz, "Active disks: programming model, algorithms and evaluation," *SIGOPS Oper. Syst. Rev.*, vol. 32, no. 5, pp. 81–91, Oct. 1998. [Online]. Available: https://doi.org/10.1145/384265.291026

[16] A. HeydariGorji, M. Torabzadehkashi, S. Rezaei, H. Bobarshad, V. Alves, and P. H. Chou, "In-storage Processing of I/O Intensive Applications on Computational Storage Drives," in *2022 23rd International Symposium on Quality Electronic Design (ISQED)*, Apr. 2022, pp. 1–6, iSSN: 1948-3295. [Online]. Available: https://ieeexplore.ieee.org/document/9806270

[17] A. HeydariGorji, S. Rezaei, M. Torabzadehkashi, H. Bobarshad, V. Alves, and P. H. Chou, "Hypertune: Dynamic hyperparameter tuning for efficient distribution of dnn training over heterogeneous systems," in *Proceedings of the 39th International Conference on Computer-Aided Design*, 2020, pp. 1–8.

[18] J. Do, V. C. Ferreira, H. Bobarshad, M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, D. Souza, B. F. Goldstein, L. Santiago, M. S. Kim, P. M. V. Lima, F. M. G. França, and V. Alves, "Cost-effective, energy-efficient, and scalable storage computing for large-scale ai applications," *ACM Trans. Storage*, vol. 16, no. 4, oct 2020. [Online]. Available: https://doi.org/10.1145/3415580

[19] A. Barbalace and J. Do, "Computational storage: Where are we today?" in *CIDR*, 2021.

[20] C. Lukken, G. Frascaria, and A. Trivedi, "Zcsd: a computational storage device over zoned namespaces (zns) ssds. corr abs/2112.00142 (2021)," *arXiv preprint arXiv:2112.00142*, 2021.

[21] G. Frascaria, "E2bpf: An evaluation of in-kernel data processing with ebpf," *Ph. D. dissertation, Universiteit van Amsterdam*, 2021.

[22] M. Torabzadehkashi, S. Rezaei, A. Heydarigorji, H. Bobarshad, V. Alves, and N. Bagherzadeh, "Catalina: in-storage processing acceleration for scalable big data analytics," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. IEEE, 2019, pp. 430–437.

[23] B. Gu, A. S. Yoon, D.-H. Bae, I. Jo, J. Lee, J. Yoon, J.-U. Kang, M. Kwon, C. Yoon, S. Cho *et al.*, "Biscuit: A framework for near-data processing of big data workloads," *ACM SIGARCH Computer Architecture News*, vol. 44, no. 3, pp. 153–165, 2016.

[24] L. A. N. Laboratory, "Los alamos national laboratory and sk hynix to demonstrate first-of-a-kind ordered key-value store computational storage device," https://discover.lanl.gov/news/0728-storage-device/, 2022, (Accessed on 07/13/2023).

[25] B. Martin and J. Molgaard, "Standardizing computational storage," https://www.snia.org/educational-library/standardizing-computational-storage-2023, April 2023, (Accessed on 07/13/2023).

[26] J. Do, Y.-S. Kee, J. M. Patel, C. Park, K. Park, and D. J. DeWitt, "Query processing on smart ssds: Opportunities and challenges," in *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, 2013, pp. 1221–1230.

[27] S. Salamat, A. Haj Aboutalebi, B. Khaleghi, J. H. Lee, Y. S. Ki, and T. Rosing, "Nascent: Near-storage acceleration of database sort on smartssd," in *The 2021 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, 2021, pp. 262–272.

[28] S. Salamat, H. Zhang, Y. S. Ki, and T. Rosing, "Nascent2: Generic near-storage sort accelerator for data analytics on smartssd," *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, vol. 15, no. 2, pp. 1–29, 2022.

[29] R. Mahapatra, S. Ghodrati, B. H. Ahn, S. Kinzer, S.-T. Wang, H. Xu, L. Karthikeyan, H. Sharma, A. Yazdanbakhsh, M. Alian, and H. Esmaeilzadeh, "In-Storage Domain-Specific Acceleration for Serverless Computing," in *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*, ser. ASPLOS '24, vol. 2. New York, NY, USA: Association for Computing Machinery, Apr. 2024, pp. 530–548. [Online]. Available: https://doi.org/10.1145/3620665.3640413

[30] W. Cao, Y. Liu, Z. Cheng, N. Zheng, W. Li, W. Wu, L. Ouyang, P. Wang, Y. Wang, R. Kuan *et al.*, "Polardb meets computational storage: Efficiently support analytical workloads in {Cloud-Native} relational database," in *18th USENIX conference on file and storage technologies (FAST 20)*, 2020, pp. 29–41.

[31] J. Chakraborty, I. Jimenez, S. A. Rodriguez, A. Uta, J. LeFevre, and C. Maltzahn, "Skyhook: Towards an arrow-native storage system," in *2022 22nd IEEE International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022, pp. 81–88.

[32] Xlinx and AMD, "Pcie peer-to-peer (p2p)," https://xilinx.github.io/XRT/master/html/p2p.html, 2024, (Accessed on 07/13/2024).

[33] AMD, "Smartssd computational storage drive installation and user guide (ug1382)," 2024. [Online]. Available: https://docs.amd.com/v/u/en-US/ug1382-smartssd-csd

[34] SNIA, "Storage networking industry association," https://www.snia.org/, 2024, (Accessed on 07/13/2024).

[35] OpenMP, "Openmp," 2012-2023. [Online]. Available: https://www.openmp.org/

[36] VPIC, 2008. [Online]. Available: https://github.com/lanl/vpic

[37] "Climate data at the nsf national center for atmospheric research," 2024. [Online]. Available: https://www.earthsystemgrid.org/
[38] https://www.tpc.org/tpch/ TPC-H, 2024. [Online]. Available: https://www.tpc.org/tpch/
[39] J. Chou, K. Wu, and Prabhat, "Fastquery: A parallel indexing system for scientific data," in *2011 IEEE International Conference on Cluster Computing*, 2011, pp. 455–464.