Optimizing Near-Data Processing for Spark

Sri Pramodh Rachuri
Stony Brook University
srachuri@cs.stonybrook.edu

Arun Gantasala
Stony Brook University
agantasala@cs.stonybrook.edu

Prajeeth Emanuel
Stony Brook University
pvethanayaga@cs.stonybrook.edu

Anshul Gandhi
Stony Brook University
anshul@cs.stonybrook.edu

Robert Foley
FutureWei
robfoley972@gmail.com

Peter Puhov
FutureWei
peterpuhov@gmail.com

Theodoros Gkountouvas
OpenInfra Labs
tedgoud@gmail.com

Hui Lei
OpenInfra Labs
dr.huilei@gmail.com

***Abstract*—Resource disaggregation (RD) is an emerging paradigm for data center computing whereby resource-optimized servers are employed to minimize resource fragmentation and improve resource utilization. Apache Spark deployed under the RD paradigm employs a cluster of compute-optimized servers to run executors and a cluster of storage-optimized servers to host the data on HDFS. However, the network transfer from storage to compute cluster becomes a severe bottleneck for big data processing. Near-data processing (NDP) is a concept that aims to alleviate network load in such cases by offloading (or "pushing down") some of the compute tasks to the storage cluster. Employing NDP for Spark under the RD paradigm is challenging because storage-optimized servers have limited computational resources and cannot host the entire Spark processing stack. Further, even if such a lightweight stack could be developed and deployed on the storage cluster, it is not entirely obvious which Spark queries would benefit from pushdown, and which tasks of a given query should be pushed down to storage.**

This paper presents the design and implementation of a near-data processing system for Spark, SparkNDP, that aims to address the aforementioned challenges. SparkNDP works by implementing novel NDP Spark capabilities on the storage cluster using a lightweight library of SQL operators and then developing an analytical model to help determine which Spark tasks should be pushed down to storage based on the current network and system state. Simulation and prototype implementation results show that SparkNDP can help reduce Spark query execution times when compared to both the default approach of not pushing down any tasks to storage and the outright NDP approach of pushing all tasks to storage.

***Index Terms*—resource disaggregation, near-data processing; spark; pushdown; modeling.**

I. Introduction

Data centers are typically composed of racks of general purpose servers, with each server equipped with an adequate amount of compute, memory, and storage resources to execute a variety of applications. However, when a specific application (e.g., a compute-intensive job) is executed on a general purpose server, the non-dominant resources (memory and storage, in this example) remain severely underutilized. To avoid resource fragmentation and improve utilization, data centers are moving towards a *disaggregated infrastructure* (DI) model, also referred to as resource disaggregation, whereby servers are built to optimize for a specific resource, such as compute or storage [1]. DI also helps to make upgrades to technologies independently and reduces time to adoption as

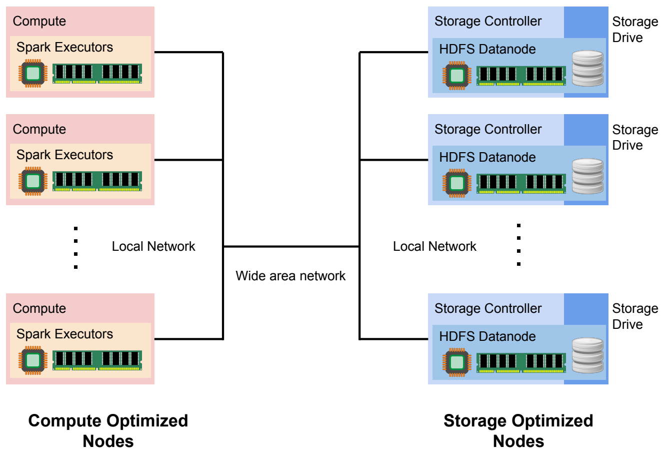


Fig. 1. Illustration of a disaggregated infrastructure (DI) deployment of Spark over a cluster of compute optimized nodes (running Spark executors) connected over the network to a cluster of storage optimized nodes (hosting HDFS datanodes).

the development of new motherboard designs and integration testings are no longer needed.

Under the DI model, resource-optimized servers are interconnected over the network. As such, data transfer between groups of optimized servers can quickly overwhelm the network, resulting in a performance bottleneck [2]–[4]. A popular application that is often deployed under the DI model in enterprise settings is Spark data processing [4], [5]. Consider the DI model shown in Figure 1 which consists of a cluster of compute-optimized nodes (servers or VMs or containers) running Apache Spark [6] connected via the network to a cluster of storage-optimized nodes hosting data on HDFS [7]. Typically, the Spark executors on the compute nodes would pull the entire data, in the form of partitions, from HDFS datanodes over the finite-bandwidth network [8]. Since modern datasets can be several hundred or gigabytes or even petabytes in size [9], the data transfer overhead poses a significant barrier for deploying Spark over DI.

Near-data processing (NDP) attempts to alleviate network latencies for large data transfers by moving computation closer to the source data [10]. In the aforementioned Spark example, consider the dataset to consist of order history for a bookstore. A user interested in analyzing the dataset might first filter the data based on a date range and then apply analysis on the filtered data; the analysis itself might involve some compute-intensive ML modeling tasks. Without NDP, the data partitions on HDFS will have to be transferred over to the Spark executors before the individual filter and analysis operations can be executed on them. However, if we can leverage any available compute power at the storage nodes to execute the filter operation at the storage nodes, then only the filtered data (and not the full dataset) will have to be transferred over the network to the compute nodes.

Prior work has shown that NDP has the potential to reduce query execution time for Spark over DI by "pushing down" the Spark operations to the storage cluster [11]. However, this outright pushdown of all Spark operations to storage may not work well for all queries as the available compute resources at the storage cluster may get quickly saturated, thereby harming the end-to-end performance. This is especially problematic because the overhead of supporting Spark operations on storage nodes already reduces the available compute resources on the storage cluster that can be leveraged to execute the pushed down operations. What is needed is a careful accounting of the end-to-end performance implications of pushdown and insight into which queries will benefit more from pushdown.

As such, to fully leverage NDP for Spark, there are three key questions that must be addressed: (1) *what software support is required to enable NDP at HDFS nodes?* (2) *which queries will benefit from pushdown?* and (3) *which operations of a given Spark query should be pushed down to HDFS to maximize the NDP benefits?* By default, HDFS nodes do not provide any support for executing parts of a Spark job. Further, since storage-optimized nodes may have very limited computing power, Spark executors or other such regular processes cannot be deployed on the HDFS nodes. What is needed is a lightweight software capable of executing parts of a Spark query. Even if such support is available, it is not entirely obvious whether a given Spark query, are parts of the query, should be pushed down for execution at the HDFS nodes. Different queries will have different compute requirements, and the optimal pushdown strategy will likely be dictated by current network and system conditions.

In this paper, we present the *design and implementation of SparkNDP*, a software framework that enables NDP for Spark. SparkNDP works by deploying a lightweight library of SQL operators on storage-optimized HDFS nodes. To provide sufficient query-level information to HDFS for performing NDP, we create a datasource extension for Spark. This extension converts Scala job queries on-the-fly to SQL commands to be executed at HDFS nodes. Our SparkNDP prototype is publicly available [12].

To determine which queries can benefit from pushdown, SparkNDP considers the reduction in data size after each query operator to decide whether the reduction in data transfer size over the network is worth the slow execution time at storage. To figure out which operations should be pushed down to HDFS, SparkNDP relies on an *analytical model* that we develop to estimate the latency benefits of each potential operation pushdown. Our model takes into account the throughput at HDFS and Spark clusters and the network delay in transferring intermediate data between the clusters. We integrate this model with SparkNDP to enable runtime decision making for optimal pushdown for any Spark query. Although we designed SparkNDP on top of HDFS, our system design and model can work with other services such as AWS S3, Azure Data Lake Storage, etc.

To evaluate the pushdown decision making of SparkNDP under different conditions, we develop a practical, data-driven simulator that allows us to consider arbitrary queries and cluster sizes. Our simulation results show that SparkNDP reduces the query execution time by upto 71% compared to the systems without NDP and 6% compared to the already existing NDP solutions. It also reduces the query execution time as much as 38% compared to the already existing NDP solutions when there are multiple jobs running simultaneously.

To validate the real-world potential of SparkNDP, we implement a prototype on top of Spark and HDFS and empirically evaluate its benefits using TPC-H Spark queries. Using our experimental setup, we consider a few different cluster sizes, different network bandwidths, and different operating frequencies for the storage cluster. Our empirical results show that SparkNDP can make the right choice between no pushdown and full pushdown in most cases, and can also sometimes determine the optimal selective pushdown when such an option provides significant benefits. Compared to the default strategy of no pushdown, SparkNDP reduces average query execution time by around 42%. Further, the query execution time under SparkNDP is within 5% of that under the offline Oracle policy, highlighting the near-optimal behavior of SparkNDP.

To summarize, the contributions of this paper are:

- We design and implement SparkNDP, a lightweight system built on top of Spark and HDFS that enables selective pushdown (to disaggregated storage) of several Spark query operations.
- We develop an analytical model for SparkNDP to determine which operations of a Spark query should be pushed down as a function of network and system conditions.
- We empirically evaluate SparkNDP on a distributed cluster setup using TPC-H Spark queries and show that SparkNDP can reduce query execution times (by as much as 25%) compared to the best of non-NDP and full-NDP Spark deployments.

The rest of this paper is organized as follows. Section II provides the necessary background on how Spark works in an DI environment. Section III summarizes relevant prior works in the areas of Near Data Processing and Disaggregated Infrastructure. Section IV discussed our design and implementation of SparkNDP. We evaluate SparkNDP in Section V using simulation and empirical results under various network and system conditions. We discuss related work in Section III and conclude the paper in Section VI.

II. Background

A. *Data processing in Spark*

Distributed data processing applications exchange and manipulate data by formatting them into objects or data structures they understand. Spark provides multiple such data structures and we use DataFrame (DF) among them. DFs are optimized for storing the required data on memory and allowing Spark to perform operations like filter, project, aggregate-sum, map, sort, etc. As DFs are generally present in memory (unless memory space is limited), the disk I/O delay is not incurred and data transfers are quick. Spark can load data into DFs either from a local file system or from network storage applications like Hadoop Distributed File System (HDFS) [7]. This gives a possibility for users to disaggregate the persistent storage requirements from computational requirements by using systems like HDFS.

The data processing flow of Spark jobs can be visualized as a Directed Acyclic Graph (DAG) in which every vertex is a user-specified operation on an DFs. In order to use all the available cores, Spark increases the parallelism of a given job by breaking down the input DataFrame into multiple partitions based on memory availability on the executors. An executor is a spark daemon running on worker nodes and are the in charge of running the spark tasks. Consider a Spark job that needs to work on a file of size 100 GB and each executor can handle a maximum of 256 MB of data. Spark master divides the 100GB data into 400 *partitions* and add them into a task queue along with the operations meant to be performed on them. Every time an executor is available and free, it picks up a task from the queue, copies the data from the storage stack, runs the operations on this partition and submits the intermediate data to the next stage. Here, we found an opportunity to pushdown the operations into the storage stack and run them before the copying process.

A spark job can be written in either in Scala or Python. Code written in scala for spark are known to be faster and once compiled, they can be launched on a Spark using spark-submit utility.

B. *Hadoop Distributed File System (HDFS)*

HDFS is a popular utility to store persistent data on multiple nodes in a distributed fashion. An HDFS cluster primarily consists of a namenode that handles all connection initializations and maintains information about the file blocks and a datanode that stores the actual file blocks. HDFS can be configured to add redundancy by copying the same file blocks into multiple datanodes for fault tolerance. It also provides an interface called “WebHDFS” that allows clients (like Spark) to access the contents using an HTTP restful API without installing any Hadoop extensions/libraries. In our implementation, we take advantage of the replication factor to increase the number of datanodes that can perform our pushdown operations and intercept the WebHDFS communication between the client and the datanode to run NDP operations.

```
sales_record = spark.read.format("csv")
                    .load("hdfs://namenode/sales_record.csv")
sales_record_filtered = sales_record.filter($"date" >= "1994-01-01"
                                        && $"date" < "1995-01-01")
total_sale_1994 = sales_record_filtered.agg(sum($"price"))
```

Fig. 2. An example spark code

C. *An example of a Spark job involving HDFS and NDP*

Figure 2 shows an example code written in scala to load a CSV file containing sales records of a store and perform “filter” and “aggregate” operations to output the sum of prices of items sold in the year 1994. Spark first divides the dataset into partitions of manageable sizes; then, each executor working on a partition loads the dataset from a datanode, filters the data based on the date, and aggregates the sum of entries in column “price”. Figure 3 illustrates this flow of data. We note that running the filter operation before transferring the records will reduce the transfer size/time between Spark and HDFS. For instance, if the sales record dataset contains all records from start of 1990 to end of 1999, then the filter operation may reduce the data size by about 10×; if the filter operation is pushed down to storage, then only the filtered data will have to be sent to the compute cluster for running the aggregate operation.

III. Related Work

This section discusses relevant works in the fields of NDP and DI. Subsection III-A discusses works that perform NDP using specialized hardware. Subsection III-B discusses works that design strategies for task distribution to improve performance. Subsection III-C discusses works that have implemented NDP for Spark.

A. *NDP using specialized hardware*

The concept of Near-Data Processing has been around recently and is known to have the potential to improve efficiency of computation in data centres [13]. Modern storage devices built on NVMe and Flash technologies offer great bandwidth and random access to data. For example, nKV [10] proposes a design to implement a fast key-value store for smart storage devices by directly controlling the physical data to achieve high I/O parallelism. Their system design also provides necessary data structures and meta data to the compute elements present on these smart storage devices to allow performing NDP. Experiments show 1.4×–2.7× better performance on complex graph-processing algorithms. This work differs from our work by executing operations on different hardware based on their optimality.

Likewise, Huang et al. [14] propose an in-network computing architecture which has a three-phase processing procedure, offloading the computation near data in the memory network for execution and aggregating the results along their routing path. Their in-network NDP architecture, Active-Routing, can achieve up to 7× speedup with a geometric mean of 60% performance improvement, and can reduce energy-delay product by 80% on average across benchmarks.

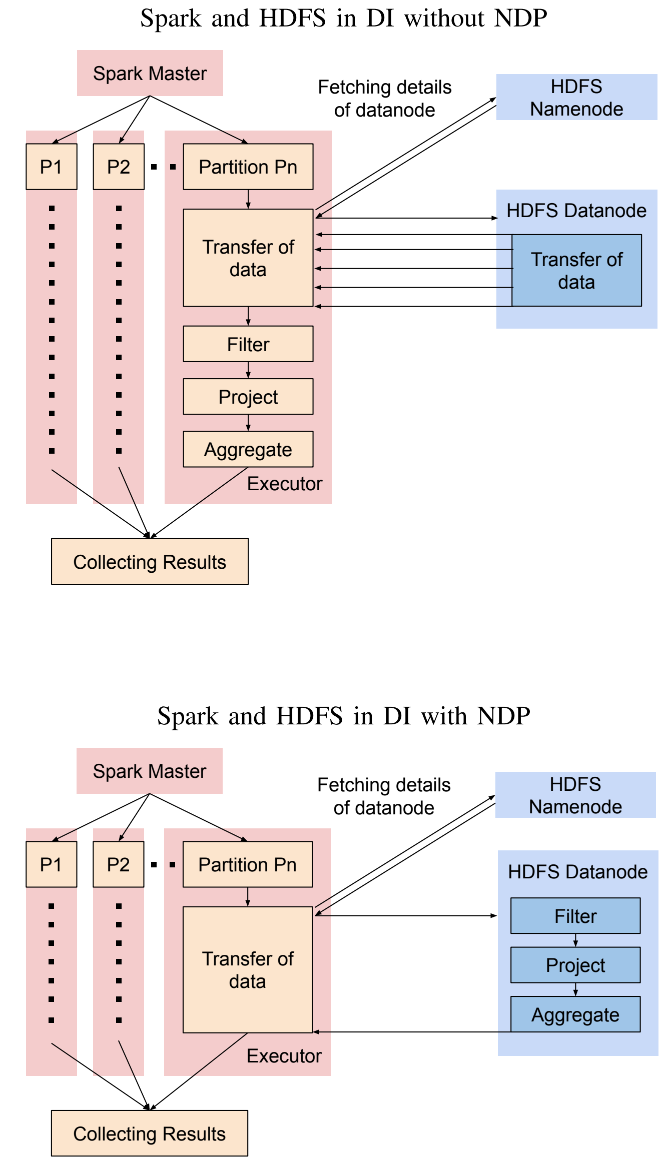


Fig. 3. Overview of how Apache Spark and HDFS share data.

Similar to nKV [10], JetStream [15] incorporates storage data structure in the form of OLAP data cubes, so data is stored for analysis near where it is generated. Experiments show that the end-to-end latency is within a few seconds, even when bandwidth drops by a factor of two. The work differs from our work by implementing data cubes which capture aggregates of data encapsulated within them to serve queries faster.

While the above works propose useful NDP designs, they require specialized hardware architecture support and are thus expensive to deploy in existing data centers.

B. Policies for distributing tasks

Gaia [16] implements a geo-distributed ML system that employs an intelligent communication mechanism over WANs to efficiently utilize the scarce WAN bandwidth. Experiments show a 1.8×-–53.5× speedup over two state-of-the-art distributed ML systems. This paper specifically deals with optimizing ML workloads by storing an approximate copy of the saved model locally at each computing node for decreasing the communication required between the nodes.

Given the scarcity of works that develop policies for NDP, we aim to fill this gap by developing an analytical model for NDP that takes decisions on which operations should be placed where (compute versus storage clusters).

C. NDP implementations

Octopus [17] proposes a technique of query pushdown for performing NDP by allowing Spark to remotely run SQL queries at the MySQL instances located on the storage devices. Experiments show that Octopus outperforms the recent Spark version 1.4.0 by about 5.25× in terms of running time to process an aggregation query. The heuristic solution proposed leverages the data locality principle which minimizes the amount of data movement. While Octopus generates sub-queries that seek to reduce the overall query processing time, our work reduces the time by deciding the exact subset of operations to be pushed down.

PushdownDB [18] implements a new DBMS which pushes down parts of analytics queries into the S3 Select engine of AWS. A program written in C++ and python works as a driver to access the results after performing SQL queries on the DB. An extensive experiments have shown that PushdownDB is 30% cheaper and 6.7× faster than a baseline without it. In our work, we use HDFS as the storage service, SparkNDP as the near-data processing system, Spark for running the jobs and develop an analytical model that can decide which operations should be pushed down to the storage cluster.

An important related work to our project is λFlow [11]. The authors present a framework to perform NDP and push operations like map, flat map, and filter to OpenStack Swift on the storage cluster from Spark on the compute cluster. They implement this as an extension to Crystal [19], a software-defined storage framework for OpenStack Swift. As the application of NDP is in software, the deployment can be inexpensive. λFlow can reduce data transfers significantly and achieve 1.47×–3.39× speedup in job completion times. The evaluations report a 90% reduction in network bandwidth and memory requirements.

In our work, we show that always pushing down every operation to the storage stack is not beneficial due to resource constraints. Instead, by selectively pushing down operations depending on the network and system state, SparkNDP can achieve 25% improvement over complete pushdown strategies like λFlow.

IV. System Design

This section describes the design of our NDP-enabling Spark framework, SparkNDP, and the analytical model we construct to decide on which queries and operations to push down to storage.

In a general use case of Spark, the user launches a Spark driver that processes the incoming queries and distributes them to its worker nodes that may be present on different physical hosts or VMs or containers. When the worker nodes start working on their part of the query, they make individual read requests to the storage system to get the required data and then process it as shown in Figure 3. After receiving the requested data, the workers perform operations on the tables.

This approach of transferring the complete dataset is inefficient as it is slowed down by the network capacity. To alleviate the network load, we investigated supporting the execution of specific operations at the HDFS datanodes, such as Filter, Project, and some Aggregate operations. These operations are simple enough and can be ported into the storage cluster, essentially performing Near Data Processing for Spark. Once NDP is implemented, the data flow in Spark and HDFS clusters is as illustrated in Figure 3. To perform NDP at HDFS, we create extensions to Spark and HDFS instead of modifying them. The extensions we made are described in the following subsections.

A. *Extension for Spark*

For pushdown operations to be run at HDFS, we require Spark to provide sufficient information to HDFS while executing the query. Spark provides an API called the Datasource V2 API to allow building a custom handler to any data source (like HDFS). We leverage this API to create an *NDP Datasource* extension for Spark. Our extension not only supports the requirements of the API but also talks to our NDP-enabled HDFS cluster to perform pushdown. Our implementation allows pushdown of Filter, Project, and some limited Aggregate operations like sum, average, max, and min. When an aggregate operation is pushed down to our NDP-enabled HDFS, the datanodes are responsible for performing the operation only for the partition it is currently handling. We handle the overall aggregation of these individual results in our NDP Datasource Extension.

In our implementation, when a query is being processed by Spark, it first interacts with the datasource extension to get metadata about the tables it needs and gets the partition details from the HDFS and our NDP client extension. The Spark master shares these details with the executors so they can independently communicate with the HDFS cluster. When an executor tries to connect to the datanodes, our NDP proxy intercepts the requests and performs NDP. More details about the NDP proxy is given in the next subsection. We also made an "NDP client" extension for Spark to enable communication with our NDP Proxy on HDFS datanodes.

The actual access to HDFS data blocks in executors is achieved through Java's InputStream [20]. We added a new parameter called "readParam" to HDFS's open API (that initializes InputStream) to send XML data required for filesystem operations and NDP pushdown. For performing NDP at the datanodes, we use SQLite. We convert the queries in Scala jobs to equivalent SQL commands on the fly at NDP Datasource to be sent to the NDP Proxy. For example, the following Scala can be converted to an SQL code as shown below.

```
Scala −
employee. filter ($‘‘department’’=‘‘HR’’). select ($‘‘name’’)
SQL −
SELECT name FROM employee WHERE department = ‘‘HR’’
```

In this query, "FROM employee" gives us the table to be read from storage. "WHERE department = "HR"" is the filter operation and "SELECT name" is the project operation. We send this SQL query through the "readParam" parameter in XML data format. Below is an example of how the XML data looks like. We add the column names of the CSV table we wish to process to the "Scheme" element, the SQL query to "Query" element and other miscellaneous parameters like delimiters required to parse the CSV file as a table.

```
<?xml version='1.0' encoding='UTF−8'?>
<Processor>
<Name>dikeSQL</Name>
<Version>0.1 </Version>
<Configuration>
<Schema>l_orderkey LONG, l_partkey LONG, l_suppkey LONG,
     l_linenumber LONG, l_quantity NUMERIC, l_extendedprice
    NUMERIC, l_discount NUMERIC, l_tax NUMERIC,
    l_returnflag STRING, l_linestatus STRING, l_shipdate
    STRING, l_commitdate STRING, l_receiptdate STRING,
    l_shipinstruct STRING, l_shipmode STRING, l_comment
    STRING
</Schema>
<Query><![CDATA[SELECT SUM("l_extendedprice" ∗ "
    l_discount") FROM CaerusObject s WHERE l_shipdate IS
    NOT NULL AND s."l_shipdate" >= '1994−01−01' AND s."
    l_shipdate" < '1995−01−01' AND s."l_discount" >= 0.05
    AND s."l_discount" <= 0.07 AND s."l_quantity" < 24.0 ]]>
</Query>
<BlockSize>134217728</BlockSize>
<FieldDelimiter>44</FieldDelimiter>
<RowDelimiter>10</RowDelimiter>
<QuoteDelimiter>34</QuoteDelimiter>
</Configuration>
</Processor>
```

B. *Extension for HDFS*

HDFS provides options to connect using the protocols Hadoop RPC (Google proto-buff based) and Hadoop Web-HDFS (REST API based). We decided to work with Web-HDFS because of its similarity with AWS S3 (for possible future compatibility) and since it allows for a more flexible architecture. For implementing NDP at HDFS datanodes, we chose to implement a reverse proxy (or called Application Proxy in Hadoop's terms) and deploy it on every datanode to intercept the communication between HDFS processes and Spark executors. We implemented this reverse proxy, which we call *NDP Proxy*, using the POCO framework [21].

While stock WebHDFS open/read APIs contain all the parameters required to provide access to file blocks, they have no provision to pass the XML file containing the additional parameters needed to perform NDP. The following is the syntax of the API provided in the official documentation [22].

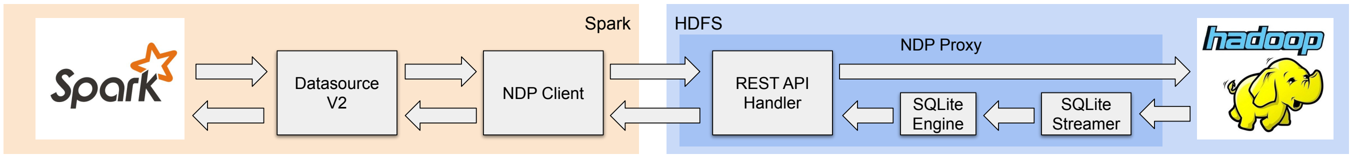


Fig. 4. Dataflow diagram of our SparkNDP implementation.

```
http ://<HOST>:<PORT>/webhdfs/v1/<PATH>?op=OPEN[&
    offset=<LONG>][&length=<LONG>][&buffersize=<INT>]
```

We work around this problem by injecting the XML code mentioned in the previous subsection into the HTTP message header. Our NDP Proxy on datanodes inspects the header of incoming requests to decide if it should perform NDP or simply forward the request (as is currently the case). In case it receives a request with the NDP header, it performs the following actions to perform NDP.

1) Extract and validate XML configuration from HTTP Header.
2) Connect to HDFS Datanode process to get the data.
3) Launch a new SQLite instance with a streaming plugin.
4) Start processing data on the fly while retrieving data using SQLite.
5) Send results back to the Spark executor.
6) Destroy SQLite instance.

Figure 4 summarizes the flow of requests through our extensions in Spark and HDFS. We have open sourced our implementation [12].

C. Analytical model to decide the best pushdown strategy

While our aforementioned SparkNDP design enables selective push down (of a given operator in a query), it is not entirely obvious which operators should be selectively pushed down. Further, the decision to push down an operator is not deterministic and depends on the network and system state; for example, if the network is expected to be congested, then pushdown may be preferred to minimize the subsequent data transfer size from storage to compute cluster.

To help decide on selective pushdown, we construct an analytical model for SparkNDP. Our analytical model, ***Net-Aware***, works by estimating: (i) the execution time of an operation (across all partitions), including any queueing time, if served at the compute cluster; (ii) the execution time of an operation (across all partitions), including any queueing time, if served at the storage cluster; (iii) the time needed to transfer the input data for the operation across the network; and (iv) the time needed to transfer the output data for the operation across the network. Typically, the ***reduction factor***, the ratio of input data size to output data size for an operation, is large enough in practice that item (iv) can be considered negligible. If the sum of items (i) and (iii) is greater than the items (ii) and (iv), then Net-Aware recommends SparkNDP to push down the operator (associated with the operation) to the storage cluster. For a given query, Net-Aware iteratively makes this decision for each operation in the query while taking into account the implementation requirement that if an operator is not pushed down, then all subsequent operations cannot be pushed down either.

To estimate the execution time of an operator at a cluster, Net-Aware considers the outstanding queue size, Q (in terms of number of partitions) at that cluster and the "drain rate", X (in units of partitions/s) at the cluster. If the operator being considered for pushed down is working on P partitions, then the execution time at a cluster, in seconds, is $T_c(Q, X) = \frac{Q+P}{X}$. For simplicity, the queue size and drain rate are considered for the entire cluster (as opposed to per core or per node) and the competing network load is assumed to be constant.

To estimate the transfer time for a given data of size D GB over a network with bandwidth B Gbps, Net-Aware also considers the number of simultaneous data transfers at the network, S. We assume that the number of queries, S_{max}, that can simultaneously transfer data between HDFS and Spark clusters is limited by the application configuration parameters, e.g., the max-jobs-per-context parameter in Spark. Alternatively, the cluster operator might decide to set the concurrent queries limit to the ratio of number of executors to number of datanodes, or something similar. Regardless, if $S < S_{max}$, then the network transfer time is simply $T_n(D) = \frac{D\times 8}{B/S}$ seconds, assuming that new data transfers cannot overtake existing transfers in the network. Note that some active transfers can complete before our target data transfer but we ignore the subsequent potential increase in network bandwidth for simplicity. However, if $S \geq S_{max}$, then the query in consideration will first have to wait until sufficient outstanding data, D_S GB, can be transferred before it can start sharing the network (i.e., until the number of active transfers falls below S_{max}) to transfer its data. In this case, we have $T_n(D) = \frac{(D_S+D)\times 8}{B/S_{max}}$ seconds.

Based on the above, Net-Aware decides to push down an operator if $T_c(Q_{Spark}, X_{Spark}) + T_n(D_{input}) > T_c(Q_{HDFS}, X_{HDFS}) + T_n(D_{output})$. Here, D_{input} and D_{output} denote the data size, in GB, for the input and output of the operator, respectively; and the subscript of Q and X represent the cluster whose queue size and throughput they denote, respectively. Finally, considering a query is made up of a sequence of n operators, $o_1, o_2, \ldots, o_n$, Net-Aware will push down the first i operators if it determines (using the above decision equation) that operations 1 through i should be pushed down but not operation $(i+1)$. For example, if operations 1 and 2 benefit from pushdown but not operation

3, then Net-Aware will disregard the decision for subsequent operations and only push down operations 1 and 2. This is because, in practice, once the query is sent for execution to the compute cluster (Spark), it cannot subsequently be pushed down again to the HDFS cluster. While other query topologies, like trees, are possible, Net-Aware currently only consider the bottleneck path as the sequence of operations.

V. Evaluation

We now evaluate SparkNDP via simulations and prototype experiments. We start by exploring a wide range of scenarios via simulation and then present empirical results based on our prototype implementation (as described in Section IV) on our experimental setup. We run our Net-Aware analytical model before launching every query to estimate the latest optimal pushdown strategy for that query. We compare our work against two policies mainly: *No-Pushdown* where all the data is brought to the compute cluster for processing (i.e., no NDP), and λ*Flow* [11] which pushes down all the compatible operations (Filter, Project, and Aggregate) to the storage cluster (full NDP). In some results, we show the *Oracle* policy, which is the offline optimal pushdown policy, for comparison. In some cases, we also include the results from partial pushdowns. *Filter Pushdown* policy performs only the Filter operation at the storage cluster where as *Filter+Project Pushdown* performs Filter and Project operations at the storage cluster but not Aggregate. We use the terms jobs and queries interchangeably in this section as we run "jobs" on Spark by submitting a TPC-H "query". Note that, for all jobs, the input dataset initially resides at the storage cluster and the final query output should be available at the compute cluster (since the user interfaces with Spark).

A. Simulation Results

We designed a discrete event simulator to simulate how Spark processes tasks using an HDFS cluster in a DI setting. A discrete event simulator models a system in the order of occurrence of events instead of progression of time. This allows us to simulate clusters of any size and very long events in a very short time, making our simulator highly scalable. We use SimPy [23] to create the simulator.

We simulate resources like CPU cores as simple discrete resources that can be requested by jobs. If a core is taken up by a task, all other task requests to the same core will be added to a queue and they will be served in a FIFO manner. Simulating network congestion is an important requirement since, in real life, concurrent data transfers between Spark executors and datanodes end up sharing the total available bandwidth. We chose to simulate the network by breaking the data transfers into smaller chunks, similar to the Maximum Transmission Unit (MTU) in computer networks, that reserve the network channel for a very short duration. We found that this abstraction handles network congestion quite well and thus the executors of the same job will also compete for network bandwidth similar to real-world networks.

Parameter	Value
Number of machines in Spark Cluster	35
Number of cores per machine in Spark Cluster	2
Clock Frequency of cores in Spark Cluster	2.27 GHz
Number of machines in HDFS Cluster	4
Number of cores per machine in HDFS Cluster	1-4
Input file size	150 GB
Clock Frequency of cores in HDFS Cluster	2.27 GHz
Bandwidth between the clusters	4 Gbps
Input size of a task	256 MB

TABLE I
CLUSTER CONFIGURATION OF OUR EXPERIMENTAL SETUP

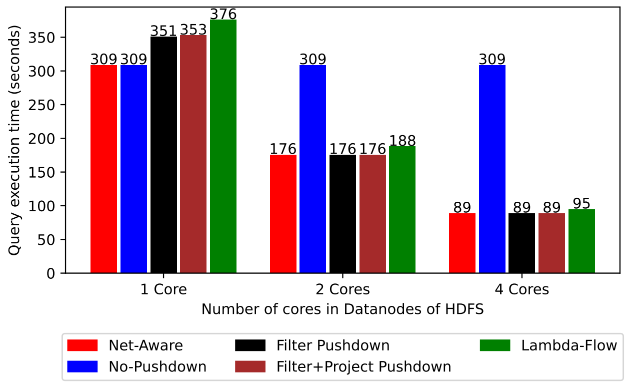


Fig. 5. Simulated query execution time when a single job is being processed.

1) *Single query results:* We first simulate a cluster that has specifications similar to our experimental setup (see Section V-B1) and execute a single job. Table I shows the specifications of our experimental setup, which are also used in our simulator. We report simulation results averaged over 500 runs. Each query consist of three operations: Filter, Project, and Aggregate with respective reduction factors (ratio of input to output data size) of 60, 1, and 10,000 respectively. These numbers are inspired by the reduction factors of real-world Spark jobs that we encountered in production.

Figure 5 shows the average query execution time under different pushdown policies as a function of the number of cores for the HDFS cluster nodes. In the case of 1 core HDFS nodes, we see that the default No-Pushdown policy performs the best, and λFlow performs the worst. However, for the 2 core and 4 core HDFS nodes, No-Pushdown is the worst option. While λFlow performs well in these cases, since storage nodes have reasonable computing power, selective pushdown (where we push down a subset of the operators to HDFS cluster) actually performs the best. This is because selective pushdown optimally balances the tradeoff between reducing the data transfer time by pushing down the initial operations to storage and reducing the compute time by leveraging the increased processing capability (70 cores) of the compute cluster. This shows that NDP is beneficial when the storage cluster has sufficient processing capacity; as the processing capacity at the storage cluster increases, the gains from NDP also increase.

In all cases, Net-Aware rightly picks the optimal policy, despite the optimal pushdown strategy being different. Com-

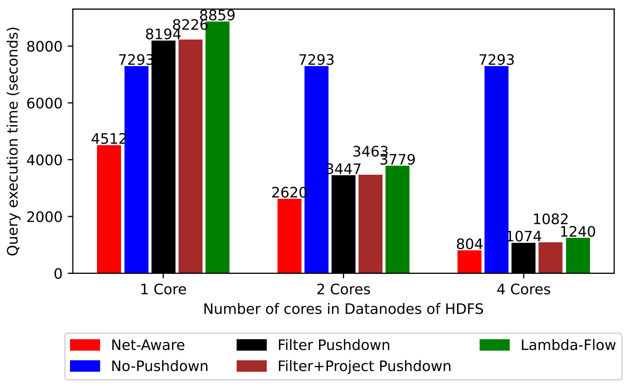


Fig. 6. Simulated query execution time when multiple jobs are processed.

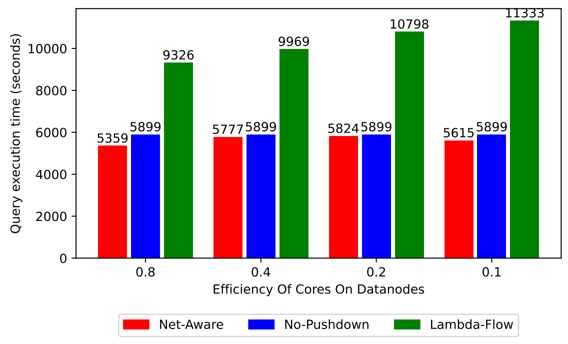
pared to the No-Pushdown policy, Net-Aware reduces query execution time by about 71% for the case of 4 core HDFS nodes. Similarly, compared to the λFlow policy, Net-Aware reduces query execution time by about 18% for the case of 1 core HDFS nodes. Clearly, Net-Aware is superior to No-Pushdown and λFlow. Even compared to the best of No-Pushdown and λFlow, Net-Aware still provides about 6% improvement for the 2 and 4 core HDFS nodes cases.

2) Multiple query results: We now consider scenarios where a stream of queries is processed as opposed to a single query, similar to the real-world scenario where multiple users are using the Spark cluster at the same time. In such cases, a job competes for resources with other active jobs already in the system. We launch 50 jobs in the simulation at a rate of 1 job every 50 seconds. We report the average query execution time across all 50 jobs. We use the same query and system configuration as in Section V-A1.

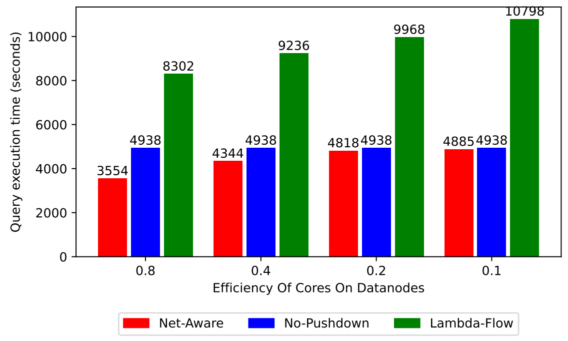
Figure 6 shows our results for the multiple jobs scenario. We see that the differences are more pronounced in this case due to the build of jobs in queues and the contention for shared compute and network resources. As for the case of single job, λFlow is better than No-Pushdown only when HDFS nodes have more than 1 core. In all cases, Net-Aware is always the optimal policy. In fact, compared to the best of λFlow and No-Pushdown, Net-Aware provides about 38% reduction in query execution time for the 1 core case, a significant improvement.

Interestingly, Net-Aware is superior to individual selective pushdown options as well. This is because, the same selective pushdown option might not be optimal for all 50 jobs in the stream. For example, while the first few jobs can benefit from (full) pushdown, subsequent jobs might be better served at the compute cluster as the storage cluster is busy processing the initial jobs. Net-Aware adapts to the current load at each cluster and can thus pick the optimal selective pushdown policy for *each* job. Compared to the best selective pushdown, the dynamic Net-Aware still provides at least 25% reduction in query execution time for all cases in Figure 6.

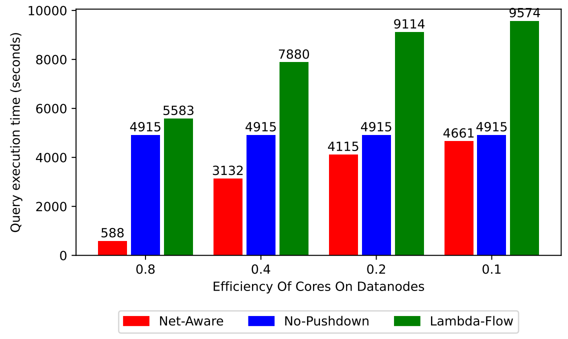
3) Queries with decompression under larger cluster setup: In production jobs, the input dataset is often compressed (e.g., in Parquet data format) to save space. In such cases, before executing the query, the dataset will first have to go through a



(a) 100 cycles/byte needed for Filter and Project



(b) 50 cycles/byte needed for Filter and Project



(c) 20 cycles/byte needed for Filter and Project

Fig. 7. Simulated query execution time when multiple jobs are processed and the input dataset is compressed.

compute-heavy decompression operation. The decompression operation may be better served at the compute cluster so that the larger output data size need not be transferred to compute cluster.

For this simulation, we consider queries similar to those in the previous simulations except that they initially have a decompression operation with reduction factor of 0.2 (meaning a compression factor of 5×). We also consider a larger cluster setup: 150 compute nodes and 50 storage nodes. Figure 7 shows our simulation results for three different job sizes (using cycles/byte needed for operations as a proxy) and as a function of the efficiency of the storage nodes. This efficiency is a representation of the operation slow down caused by execution the operations at the lightweight storage cluster.

In all cases, we see that λFlow is significantly slower at processing queries due to the heavy decompression operation;

as expected, λFlow does worse as the storage nodes' efficiency decreases. While the No-Pushdown option is reasonable in most cases, it does suffer performance degradation when the operations are light enough (Figure 7 (c)) that some of them will benefit from being pushed to storage to leverage the data transfer reduction when moving the query over to the compute cluster. In such cases, Net-Aware carefully balances the selective pushdowns to achieve as much as $8\times$ reduction in query time over the best of No-Pushdown and λFlow.

B. Empirical Results

To evaluate SparkNDP and Net-Aware in real-world clusters (where complexity is much higher than in the simulated setup), we perform a limited evaluation in our experimental testbed.

1) Experimental setup: Our experimental setup consists of 10 servers. We employ 6 of them (each equipped with two Intel Xeon L5520 for a total of 16 cores) for the compute cluster. Each server runs 6 Spark containers, giving us a total of 35 Spark executors each with two cores and 0.5 GB memory. The remaining 4 servers comprise the storage cluster and each run 1 instance of HDFS datanode on a container using a maximum of 4 cores. We underclock the storage cluster CPU from 2.67 GHz to 1.6 GHz in some of our experiments using the CPUFreq governor of linux kernel [24] to experiment with different storage cluster settings. Each server is capable of achieving a network throughput of 1 Gbps, resulting in a maximum of 4Gbps across the clusters. The servers hosting Spark (compute cluster) and the servers hosting HDFS (storage cluster) are physically present on two different racks connected by a 10Gbps network. In some of our experiments, we use Traffic Control (TC) [25] and Network Emulation (NETEM) [26] modules of the Linux kernel to reduce the bandwidth between the clusters. The replication factor is set to 4 on HDFS to make the data available to all datanodes to perform NDP.

2) Experimental methodology: We use the prototype implementation described in Section IV to evaluate SparkNDP experimentally on our aforementioned experimental setup. We use TPC-H Spark queries as our workload and run then on a 100 GB dataset generated by DBGEN. We obtain end-to-end query execution time from Spark, and use this as our metric. For comparison, we also run Spark without any pushdown (No-Pushdown) and Spark with full pushdown (λFlow [11]). Finally, we also denote the best-performing policy among all possible pushdown combinations as Oracle, which is the best offline optimal policy.

C. Experimental results

Figure 8 shows our empirical results for the case of a single job being processed; we show results for two different TPC-H queries, with Q19 being more computationally involved than Q06. The vertical lines at the top of each bar illustrates the standard deviation of that set of results (since each result is averaged over multiple experimental runs). Figure 8 (a) shows our results when we have 4 storage nodes underclocked to the lowest frequency (to emulate weaker storage cluster). We see that No-Pushdown is the best option when the storage nodes have only 1 core, but full pushdown (λFlow) is typically the best when we have more than 1 core. In some cases, selective pushdown is optimal, as in the case of Q19 when we have 4-core storage nodes. We see that, even in real experiments, Net-Aware typically is able to pick the best pushdown option, even if selective pushdown is the optimal option. Compared to No-Pushdown, λFlow, and best of No-Pushdown and λFlow, Net-Aware achieves as much as 42%, 11%, and 5% reduction in query execution time.

In Figure 8 (b), we reduce the bandwidth between the clusters to 1 Gbps. In this case, even single core HDFS nodes provide query execution time reduction over No-Pushdown since the network is a significant bottleneck. Net-Aware adapts to this slower network and ensures that operations are aggressively pushed down to storage; while Net-Aware is not always optimal, it does result in significant improvements over No-Pushdown.

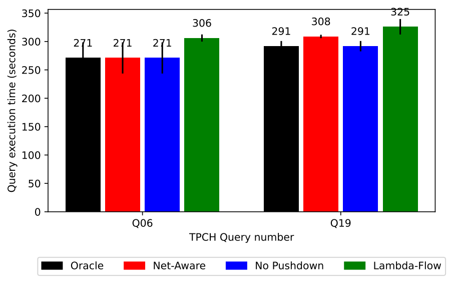
Figure 8 (c) shows our results for the case of 2 HDFS nodes at full clock frequency. In this case, the effective bandwidth between the clusters is 2 Gbps since each HDFS node has a maximum bandwidth of 1 Gbps. This set of results shows that selective pushdown can indeed improve over No-Pushdown and full pushdown even in real experimental settings. In most cases, Net-Aware does recognize selective pushdown as the right policy, providing as much as 6% improvement over the best of No-Pushdown and λFlow.

Finally, we also experiment with the case of multiple jobs being processed. As seen in simulations, the benefit of selective pushdown can be more pronounced in such cases. We consider Query 06 and run 10 jobs with an inter-arrival time of 50s. Figure 9 shows our results under three different settings (as denoted by the x-axis labels). We find that Net-Aware chooses only Filter pushdown as the right option in all three scenarios. By doing so, Net-Aware is always superior to No-Pushdown and near-optimal in all cases. For the case of 2 storage nodes with 2 cores each, Net-Aware indeed picks the optimal selective pushdown policy, providing around 25% reduction in query execution time over the best of No-Pushdown and λFlow. This is a substantial improvement compared to the existing approaches of no NDP and full NDP.

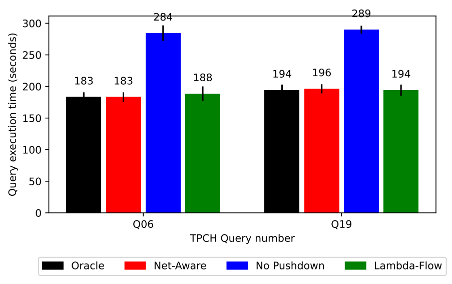
VI. CONCLUSION

Data processing and analysis is a frequently employed task in the industry, and improving the execution time of such tasks can substantially speed up analytics processes. Frameworks like Spark are routinely employed to run such data processing tasks on clusters of servers. Disaggregated infrastructure settings are being increasingly employed in practice to improve resource usage and avoid resource fragmentation; however, the network overhead in such settings poses a serious problem for data processing tasks that require data transfer from the storage cluster to the compute cluster. As such, an important problem is how to best design data processing frameworks for disaggregated infrastructure settings.

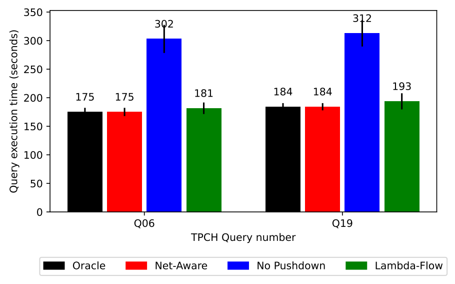
This paper presents our experience and insights from designing and using an NDP-based Spark framework,



(i) 1 core

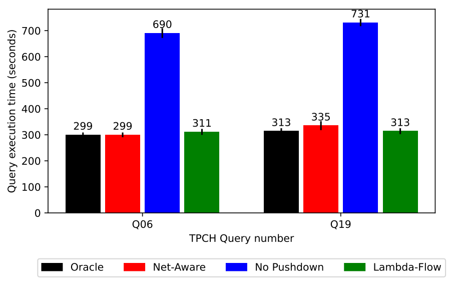


(ii) 2 cores

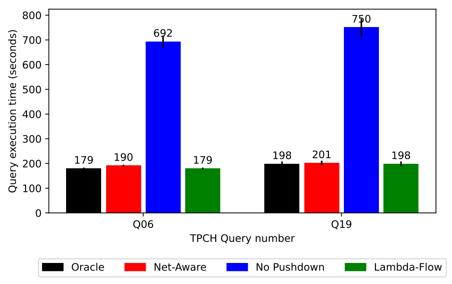


(iii) 4 cores

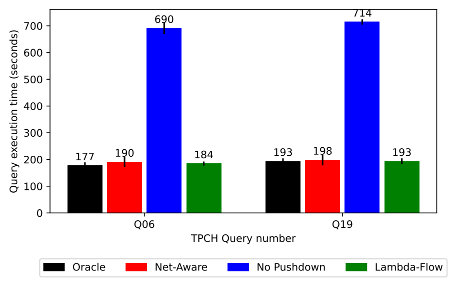
(a) Number of storage nodes = 4, storage nodes clock speed = 1.60 GHz, network bandwidth between clusters = 4 Gbps.



(i) 1 core

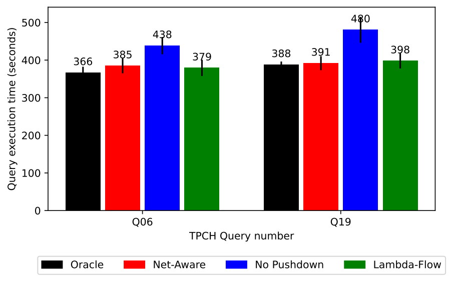


(ii) 2 cores

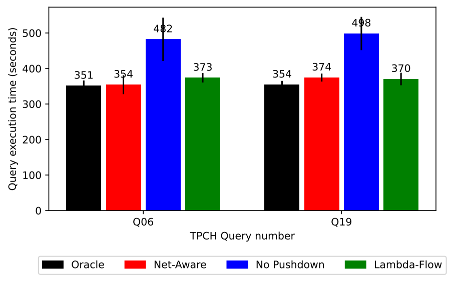


(iii) 4 cores

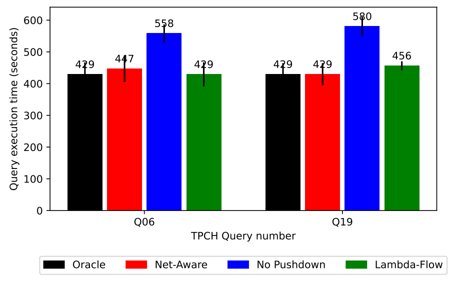
(b) Number of storage nodes = 4, storage nodes clock speed = 1.60 GHz, network bandwidth between clusters = 1 Gbps.



(i) 1 core



(ii) 2 cores



(iii) 4 cores

(c) Number of storage nodes = 2, storage nodes clock speed = 2.67 GHz, network bandwidth between clusters = 2 Gbps.

Fig. 8. Empirical query execution time when a single job is being processed under various system configurations.

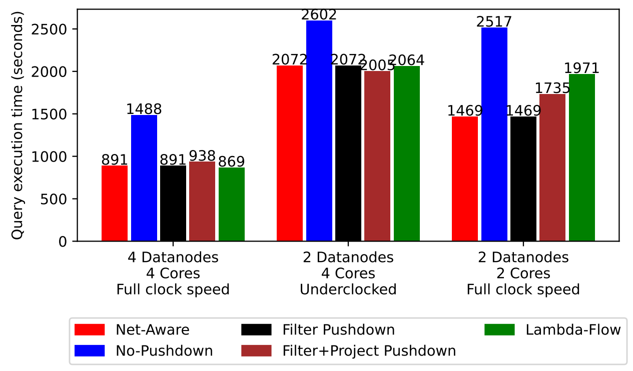


Fig. 9. Empirical query execution time when multiple jobs are processed under various system configurations.

SparkNDP, in a disaggregated infrastructure setting. We designed and open sourced SparkNDP, a framework built on top of Spark and HDFS that allows specific Spark operators to be pushed down to storage, thus enabling NDP for Spark, including selective push down of operations. SparkNDP works by leveraging SQLite to execute a subset of Spark operations in a lightweight manner on the storage cluster. We also constructed an analytical model to help SparkNDP decide which queries to use pushdown for and which specific operations of a query to push down, to optimize query execution time.

Our simulation results show that pushdown should be carefully employed based on the current system and network state and based on the query being executed. We find that full pushdown is only beneficial when the storage cluster has sufficient compute resources as needed for the pushed down query operators or when the initial dataset is too large to be quickly transferred over the network. We also find that *selective pushdown* is often a superior choice compared to the default no pushdown or full pushdown options, thus highlighting a novel NDP push down option that can benefit the community.

Our prototype experimental results show that there is significant query execution time reduction possible by employing the right pushdown strategy. Our implementation results also highlight the importance of taking into account the system and network state when deciding on the pushdown strategy to adopt for a given query.

Acknowledgment

This work was supported by NSF CNS grants 1750109, 1909356, 2106434, and a gift from OpenInfra Labs.

References

[1] Y. Shan, Y. Huang, Y. Chen, and Y. Zhang, "LegoOS: A disseminated, distributed OS for hardware resource disaggregation," in *Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, Carlsbad, CA, USA, 2018, pp. 69–87.

[2] P. X. Gao, A. Narayan, S. Karandikar, J. Carreira, S. Han, R. Agarwal, S. Ratnasamy, and S. Shenker, "Network requirements for resource disaggregation," in *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. Savannah, GA: USENIX Association, Nov. 2016, pp. 249–264. [Online]. Available: https://www.usenix.org/conference/osdi16/technical-sessions/presentation/gao

[3] V. Jalaparti, C. Douglas, M. Ghosh, A. Agrawal, A. Floratou, S. Kandula, I. Menache, J. S. Naor, and S. Rao, "Netco: Cache and i/o management for analytics over disaggregated stores," in *Proceedings of the ACM Symposium on Cloud Computing*, ser. SoCC '18, Carlsbad, CA, USA, 2018, p. 186–198.

[4] A. Klimovic, H. Litz, and C. Kozyrakis, "Selecta: Heterogeneous cloud storage configuration for data analytics," in *Proceedings of the 2018 USENIX Annual Technical Conference (USENIX ATC 18)*, Boston, MA, USA, 2018, pp. 759–773.

[5] B. Cho and E. Seyfe, "Taking advantage of a disaggregated storage and compute architecture," *Spark+ AI Summit*, 2019.

[6] The Apache Software Foundation, "Apache Spark," http://spark.apache.org/.

[7] K. Shvachko, H. Kuang, S. Radia, and R. Chansler, "The hadoop distributed file system," in *2010 IEEE 26th Symposium on Mass Storage Systems and Technologies (MSST)*, Incline Village, NV, USA, 2010, pp. 1–10.

[8] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation*, ser. NSDI'12, San Jose, CA, USA, 2012, pp. 15–28.

[9] R. Xin, "World record set for 100 tb sort by open source and public cloud team," https://opensource.com/business/15/1/apache-spark-new-world-record.

[10] T. Vinçon, A. Bernhardt, I. Petrov, L. Weber, and A. Koch, "Nkv: Near-data processing with kv-stores on native computational storage," in *Proceedings of the 16th International Workshop on Data Management on New Hardware*, ser. DaMoN '20. New York, NY, USA: Association for Computing Machinery, 2020. [Online]. Available: https://doi.org/10.1145/3399666.3399934

[11] R. Gracia-Tinedo, M. Sanchez-Artigas, P. Garcia-Lopez, Y. Moatti, and F. Gluszak, "Lamda-flow: Automatic pushdown of dataflow operators close to the data," in *2019 19th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*, 2019, pp. 112–121.

[12] Open Infrastructure Labs, "Caerus dike," https://github.com/open-infrastructure-labs/caerus-dike/tree/v1.

[13] R. Balasubramonian, J. Chang, T. Manning, J. H. Moreno, R. Murphy, R. Nair, and S. Swanson, "Near-data processing: Insights from a micro-46 workshop," *IEEE Micro*, vol. 34, no. 4, pp. 36–42, 2014.

[14] J. Huang, P. Majumder, S. Kim, T. Fulton, R. R. Puli, K. H. Yum, and E. J. Kim, "Computing en-route for near-data processing," *IEEE Transactions on Computers*, vol. 70, no. 6, pp. 906–921, 2021.

[15] A. Rabkin, M. Arye, S. Sen, V. S. Pai, and M. J. Freedman, "Aggregation and degradation in JetStream: Streaming analytics in the wide area," in *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*. Seattle, WA: USENIX Association, Apr. 2014, pp. 275–288. [Online]. Available: https://www.usenix.org/conference/nsdi14/technical-sessions/presentation/rabkin

[16] K. Hsieh, A. Harlap, N. Vijaykumar, D. Konomis, G. R. Ganger, P. B. Gibbons, and O. Mutlu, "Gaia: Geo-Distributed machine learning approaching LAN speeds," in *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*. Boston, MA: USENIX Association, Mar. 2017, pp. 629–647. [Online]. Available: https://www.usenix.org/conference/nsdi17/technical-sessions/presentation/hsieh

[17] Y. Chen, C. Xu, W. Rao, H. Min, and G. Su, "Octopus: Hybrid big data integration engine," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015, pp. 462–466.

[18] X. Yu, M. Youill, M. Woicik, A. Ghanem, M. Serafini, A. Aboulnaga, and M. Stonebraker, "Pushdowndb: Accelerating a dbms using s3 computation," 2020. [Online]. Available: https://arxiv.org/abs/2002.05837

[19] CRYSTAL, "Crystal: Software-defined storage for openstack swift," http://cloudlab.urv.cat/crystal/.

[20] Oracle, "Java class InputStream," https://docs.oracle.com/javase/8/docs/api/java/io/InputStream.html.

[21] Applied Informatics Software Engineering GmbH, "POCO C++ Libraries - Simplify C++ Development," https://pocoproject.org/index.htmll.

[22] The Apache Software Foundation, "WebHDFS REST API," https://hadoop.apache.org/docs/r1.0.4/webhdfs.html.

[23] "Simpy," https://simpy.readthedocs.io/en/latest/.

[24] Linux Kernel Organization Inc., "Cpufreq governors," https://www.kernel.org/doc/Documentation/cpu-freq/governors.txt.

[25] B. Hubert, "Traffic control in the linux kernel," https://man7.org/linux/man-pages/man8/tc.8.html.

[26] Linux Foundation, "Network emulation," https://wiki.linuxfoundation.org/networking/netem.