Modeling Analytics for Computational Storage

Veronica Lagrange
Memory Solutions Lab
Samsung Semiconductor, Inc.
San Jose, U.S.A.
veronica.l@samsung.com

Harry (Huan) Li
Memory Solutions Lab
Samsung Semiconductor, Inc.
San Jose, U.S.A.
harry.li@samsung.com

Anahita Shayesteh
Memory Solutions Lab
Samsung Semiconductor, Inc.
San Jose, U.S.A.
anahita.sh@samsung.com

Abstract

Next generation flash storage will be armed with a substantial amount of computing power. In this paper, we investigate opportunities to utilize this computational capability to optimize Online Analytical Processing (OLAP) applications. We have directed our analysis at the performance of a subset of TPC-DS queries using Hadoop clusters and two database engines, SPARK-SQL and Presto. We model the expected speed-up achieved by offloading a few operations that are executed first within most SQL plans. Offloading these operations requires minimal cooperation from the database engine, and no changes to the existing plan. We show that the speed-up achieved varies significantly among queries and between engines, and that the queries benefiting the most are I/O heavy with high selectivity of the "needle in the haystack" variety. Our main contribution is estimating the speed-up anticipated from pushing the execution of a few key SQL building blocks (scan, filter, and project operations) to computational storage when using read optimized, columnar Parquet format files.

CCS CONCEPTS

• Computing methodologies → Modeling and simulation → Model development and analysis → Model verification and validation;

• Hardware → Communication hardware, interfaces and storage → External storage;

• Information systems → Data management systems → Database management system engines → Database query processing → Query planning;

• Information systems → Data management systems → Database management system engines → Online analytical processing engines;

KEYWORDS

Columnar Database, Parquet, SQL, Smart Storage, acceleration, offloading, TPC-DS, Spark, Presto, OLAP

ACM Reference Format:

Veronica Lagrange, Harry (Huan) Li, Anahita Shayesteh. 2020. Modeling Analytics for Computational Storage. In *Proceedings of the 2020 ACM/SPEC International Conference on Performance Engineering (ICPE'20), April 20–24, 2020, Edmonton, AB, Canada.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3358960.3375794

1 Introduction

Current developments in "big data" storage solutions gear towards moving data processing closer to where the data resides, reducing unnecessary movement and speeding up data processing considerably. Computational storage is an emerging trend where a comparatively large amount of data processing occurs inside the storage layer. Examples of new devices exposing flash storage internal computing power include Samsung's SmartSSD [1], NGD Systems [2], and ScaleFlux [3]. This new functionality signals performance improvement opportunities for I/O heavy workloads containing operations amenable to being completed near the storage source. One of the most critical types of database analytics – OLAP – well exemplifies this type of opportunity. It is typically very I/O intensive and contains quite a few building blocks that may be seamlessly moved to, or executed by, a computational storage device.

Offloading is not a new concept. Network processors, GPUs and recently machine learning specialized processors are widely used to accelerate specific compute kernels while freeing CPU resources. We will show that the offloading of many more time-consuming operations from the host CPU to storage improves both workload performance and system efficiency. The immediate benefit, of course, is a sizeable decrease in I/O volume. This reduction in I/O leads to less host resource utilization, which not only improves performance of individual queries, but also increases server capacity. Besides database operations, other frequent operations that can be executed near the storage device include encryption and compression.

Database analytics workloads are especially read-intensive. It is not uncommon for I/O reads to take 90% or more of the total execution time. Offloading some of that to storage reduces I/O bandwidth along with other host resource usage, and may improve performance considerably. Furthermore, SSDs have an internal bandwidth that is much higher than that which is exposed to the host computer through existing channels (SAS, SATA, PCI-E, etc.) [4], which means that computational storage has a large amount of untapped potential to exploit.

This paper discusses the expected performance benefits of offloading some important basic database operations – namely Scan, Filter and Project – to computational storage. We evaluate the performance estimate model using TPC-DS workload and two database engines running on Hadoop clusters: SPARK-SQL and Presto.

This paper is organized as follows: after covering previous computational storage database offloading work, we explain the OLAP workload selection, and the configuration of our two clusters. In Section IV we dive into TPC-DS characteristics and examine the overall performance from running on the two Hadoop clusters, which have been the focus of our experimentation. In Section V, we explain our modeling methodologies, and in Section VI we describe and analyze results from that modeling. Specifically, we show how a substantial speed-up from computational storage optimization can depend on multiple factors. Finally, we briefly discuss other SQL building blocks amenable to computational storage pushdowns, and conclude.

2 Previous Work

Most previous work on pushing SQL functions down to computational storage concentrate on specific functions of a specific Database Engine. Summarizer [5] modifies the existing NVMe command interface to implement four operations: initialize variables or set queries; read data and execute computation; read data and filter – the selection case; and the transfer of the output results to the host. From their case studies, using 3 TPC-H queries and a very small scale factor (100MB – 0.1SF), we determined that they could do similarity joins as well. They compare different degrees of computation offloading for these three queries. The authors show that somewhat complex computations can be carried out near storage, and briefly discuss the data integration problem: how to combine data from different formats and sources. They concentrate on one specific integration problem: similarity join, and describe the heuristics they use. Left unanswered is the bigger issue on how to integrate truly distinct formats.

YourSQL [6] is based on MariaDB. YourSQL allows for complex query operations to be offloaded to a smart SSD in the form of an ISC task. That paper spends the bulk of its time talking about optimizer heuristics. One very interesting observation, from the authors' performance analysis is that while your typical SQL application – OLTP or OLAP – cannot exhaust an NVMe bandwidth, its near-storage implementation can.

Biscuit [7] is what YourSQL uses to enable its computational storage operations. It provides the user application with C++ APIs. The user's SSD-side C++ program with Biscuit APIs, called an SSDlet, is loaded in the device. A host-side program invokes and coordinates execution of the SSDlet tasks using libsisc; communication is done by linking input and output ports to specific tasks. Here they also claim that the APIs used to access files are nearly identical to standard libraries.

ExtraV [8] is IBM's effort at computational storage for graph processing based on their CAPRI [9]. This paper describes an FPGA prototype that executes common graph traversal functions near the device. It works like virtual memory for graph applications, as it provides the host with the illusion that the entire graph lives in memory, while it is actually partly stored and compressed in an SSD. The authors have stated that graph processing is mostly done in memory, either in single servers or clusters, and that it cannot be done efficiently when graphs grow beyond the available memory.

PG-Strom [10] is an accelerator for PostgreSQL that offloads part of the SQL workload to a GPU. Supports Joins and Aggregates. However, by the time of that publication [10] all data fed to the GPU came from main memory (not storage).

Neteeza was the first successful product to use FPGAs as computational storage computing accelerators for analytics data engines. It does not require any software installation or tuning. Just plug and play. Neteeza database engine is based on Postgres [11], and implements four functions in its FPGA engine: Compress, Project, Restrict and Visibility. Francisco [12] claims that Neteeza's engine decompresses data at wire speed. Project and Restrict operations filter out columns and rows, respectively, based on the parameters in the SELECT and WHERE clauses of a query. The Neteeza Visibility engine is focused on database integrity, and therein, filters out rows that should not be seen by the query, such as any rows being inserted by a transaction that has not yet committed.

Computational storage has also attracted interest beyond SQL and database applications. For example, REGISTOR [24] is an FPGA platform applying regex search, on-the-fly, to any file being transferred from an SSD to the host; INSIDER [25], also an FPGA-based drive controller, exposes a virtual file system with embedded programmability, allowing programmers to push down operations customized to the application's specific needs.

3 Workload and setup

Here, we explain the TPC-DS benchmark, as well as the two cluster configurations used in the experiments described. Moreover, we describe the two database engines (SPARK-SQL and Presto), and explain the rationale behind using the Parquet file format to offload SQL operations to computational storage.

3.1 TPC-DS

"The TPC Benchmark DS (TPC-DS) is a decision support benchmark that models several generally applicable aspects of a decision support system" [13].

TPC-DS contains 24 tables, organized as a snowflake schema. It contains 6 very large FACT tables, and many small DIMENSION tables. Furthermore TPC-DS is comprised of 99 queries, each one representing a different business question. So, even though this is an artificial benchmark, it tries to mirror real-life applications. Schema is scalable, with the smallest being 1GB and the largest 100TB. The 1GB dataset is used for QA only. Performance is measured in Queries per Hour @ Scale Factor (QphDS@SF), and must include multiple tests (pertaining to power, throughput, and data maintenance). In this study, we consider a subset of the power test. For a more detailed explanation of the TPC-DS benchmark, we refer the reader to [14].

TPC-DS has been around since 2007, but did not catch up until recently and after a major re-write, with the first published official report dated March 2018 (Cisco) [15]. As of January 2020, there are only six official reports published. Nonetheless, subsets of TPC-DS are heavily used informally by the industry to demonstrate up and coming trends [16] [17]. TPC-DS is one of many Transaction

Processing Performance Council (TPC) benchmarks [18], and as such covers enough general OLAP cases to be useful to practitioners.

Because FACT tables are orders of magnitude larger than DIMENSION tables, we will gravitate towards queries that are "FACT table Scan heavy," as opposed to queries that are "DIMENSION table Scan heavy."

3.2 Test Configuration

Two clusters are used in this paper, and since they are configured to run SPARK-SQL and Presto, we refer to them simply as the SPARK-SQL cluster and the Presto cluster. Each has eight data nodes with different hardware. The detailed configuration is listed in Table I. Both engines use Hive Metadata, and the Parquet file format.

SPARK-SQL is Apache Spark's high-level tool for structured data processing [19]. It is an in-memory, distributed, RDBMS that understands SQL and a Dataset API (available in Java and Scala). User applications interface with SPARK-SQL via a command-line module, JDBC or ODBC. SPARK-SQL also supports reading and writing data stored in an existing Apache Hive installation.

Table 1: CLUSTER CONFIGURATION

		SPARK-SQL	Presto
Data Node Hardware	CPU	Intel(R) Xeon(R) Gold 6152 CPU @ 2.10GHz	Intel(R) Xeon(R) CPU E5-2699 v4 @ 2.20GHz
	Memory	256GB	256GB to 1024GB
	Local Storage	2x NVMe SSD 3.2TB	3x NVMe SSD 1.6TB
Software Stack	OS	Linux Kernel 4.13.0	Linux Kernel 4.x.x
	SPARK-SQL/Presto	2.3.0	0.205
	Hadoop	2.7.3	2.9.0
	Hive	1.2.1000	1.2.2
	HDFS Replication	1	
TPC-DS	Scale Factor	10000	
	Storage Format	Paquet	

Presto is a distributed SQL query engine designed to query large data sets distributed over one or more heterogeneous data sources [20]. Presto provides a CLI interface, and query processing (parser, planner, scheduler), but will use data and metadata provided by other software components (HBase, Hive, MySQL, etc.). Presto interacts with these other components via connectors, and this is its claim to fame as it is possible to combine multiple, different data sources into one query seamlessly. There is no need for very expensive ETL (Export-Transform-Load) datasets in order to analyze them.

Similar to classic massively parallel processing (MPP) DBMS [21], Presto is a distributed system that runs on a cluster. Presto client submits SQL statements to a master daemon coordinator. Using metadata from connectors, the coordinator parses the query, generates the plan, and then schedules and coordinates how it is executed by the workers. Workers get data from connectors, execute assigned tasks, and deliver results to the client. All processing happens in memory, and data is pipelined across the network between different stages.

Parquet is an open source columnar file format that was designed to be used with OLAP systems [22]. The Parquet file format is READ optimized, as inserts or updates can be expensive operations. It was inspired by the "Dremel" paper [23], and is extensively used in the Hadoop ecosystem. Furthermore, each Parquet disk file contains the table's schema. This feature resolves the issue of the device being aware of the table metadata, a requirement for any computational storage processing. Furthermore, existing Parquet readers are capable of projecting and filtering certain data types using statistics provided in metadata. Implementing some functionality in a computational storage device is complementary and in addition to the existing pushdown capabilities of Parquet.

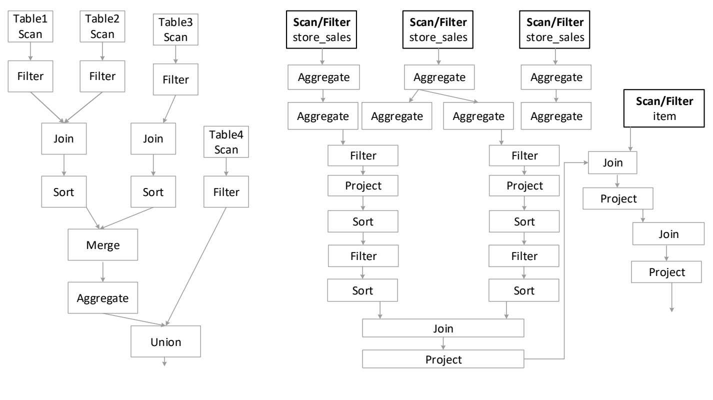
TPC-DS queries are downloaded from the TPC website and results were verified against sample output from the TPC. All queries run sequentially as a single test job. Before each query, the memory cache is cleared. In addition,

- SPARK-SQL is restarted before every query
- Presto is restarted before the job

4 TPC-DS Characterization

In this section, we discuss the many stages (or fragments) of the execution plan generated by the query optimizer. Next, we show SPARK-SQL and Presto query runtime results for TPC-DS. We list them side by side to show that they behave differently in order to illustrate and explain the different speed-ups that one might see for the same query executed with different engines. Next, we examine the concept of Scan Ratio, and how we use it to characterize and rank queries.

4.1 Typical TPC-DS query plan



(A) Generic SQL query plan (B) SPARK-SQL plan for Query 44

Figure 1: SQL query plans

SQL query plans are composed of basic building blocks. They form an execution tree. Each building block typically focuses on one specific operation, and is scheduled by a SQL engine. How these building blocks are assembled dictates query performance. Main building blocks include: Scan, Filter, Project, Aggregate, Sort, Join, Merge, Union. Figure 1 (A) illustrates a typical query sequence, with building blocks being executed from top to bottom. Figure 1 (B) is the building block sequence created by the SPARK-SQL planner for TPC-DS Query 44.

The functionality of some building blocks includes:

- Scan: Read database content from storage to compute host memory and apply any needed transformations
- Filter: Filter table rows in memory with giving criteria
- Project: Select table columns in memory
- Join: Combine two tables based on given criteria

4.2 Performance of all queries

Figure 2 shows the runtime for all TPC-DS queries for SPARK-SQL and Presto. With 10TB dataset, SPARK-SQL completes 91 and Presto completes 61 queries. Both database engines store all intermediate results in memory, and the queries that failed incurred an "out-of-memory" error. The query runtime has a wide range from less than a minute to many hours. We have not matched our cluster hardware configurations for SPARK-SQL and Presto, as it is not our goal to compare the performance between them. The point of this paper is to showcase a subset of the many different system parameters influencing the potentially substantial speed-up afforded by computational storage devices. We demonstrate that even though computational storage can provide impressive speed-ups, the benefits vary significantly depending on many other parameters such as table size, selectivity, query plan, etc.

In the following sections, we will select five queries from each cluster based on system characterization of the queries and potential offload benefits, and provide further analysis of each.

4.3 Scan Ratio

Scan Ratio is defined as the total CPU time spent on a database Scan operation, divided by total CPU time consumed by the query. The CPU time is reported by query planner from database engine. This time is not the wall clock time and should not be confused with query runtime.

For TPC-DS queries, the Scan Ratio ranges from near 0% to ~93% on SPARK-SQL and up to nearly 100% for Presto. In Figure 3, queries are sorted by their Scan Ratio, from left to right. Q9, with the highest Scan Ratio, is furthest to the right. Notice that most CPU intensive queries have a small Scan Ratio, but not all. Some complex queries, such as Q44, are both compute and I/O intensive.

High Scan Ratio does not necessary mean the query reads more data from storage, it only indicates that time spent on I/O is higher relative to other query operations. For example, Query 45 has a total disk read of ~1.3TB, its Scan Ratio is only 2.99%. But for Query 9, which has the highest Scan Ratio of ~93%, total disk read is only ~105GB. Although the total query runtime difference is not large (Q9, 212.36 sec, Q45, 176.02 sec.), the CPU cycles spent on non-I/O operations caused the Scan Ratio to be lower for Q45. A high Scan Ratio indicates that a query is a strong candidate for computational storage optimization, since its I/O operations are likely to be in its critical path, while a low Scan Ratio indicates that operations other than I/O are the bottleneck.

5 Offloading Model

Here, we explain how we selected each plan stage to be offloaded to computational storage, followed by a detailed description of the model methodology used with both database engines. Notice that the methods are somewhat different, which we chose to do in order to cover more aspects of the offloading process.

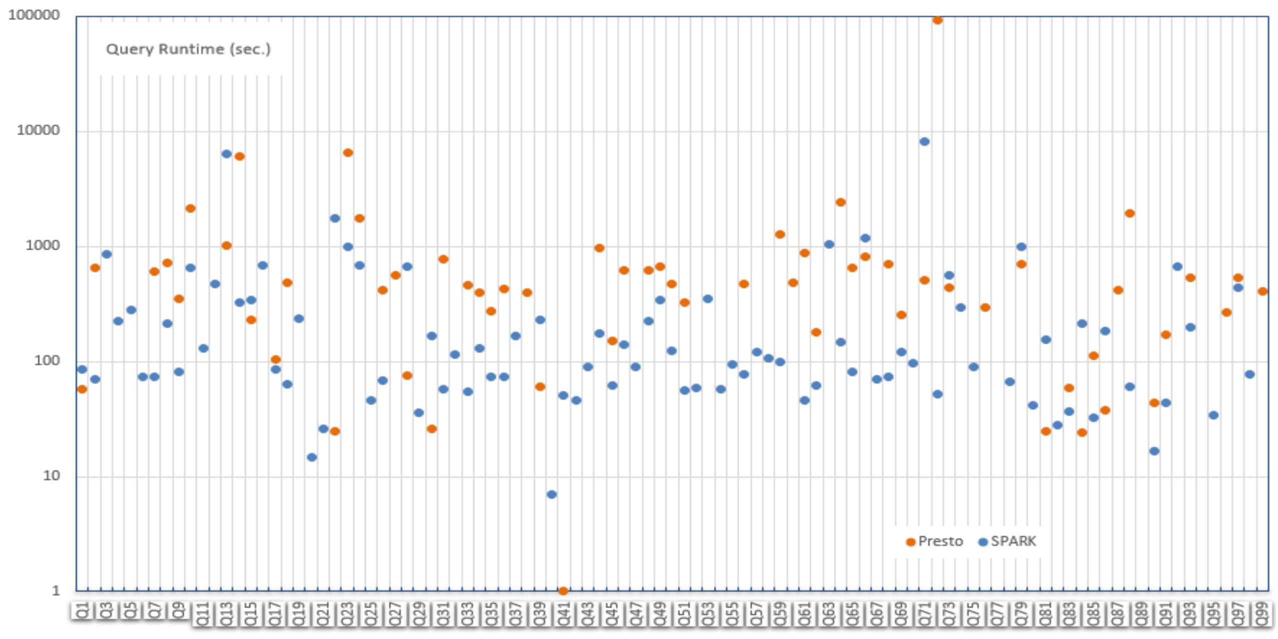


Figure 2: TPC-DS runtime query comparison

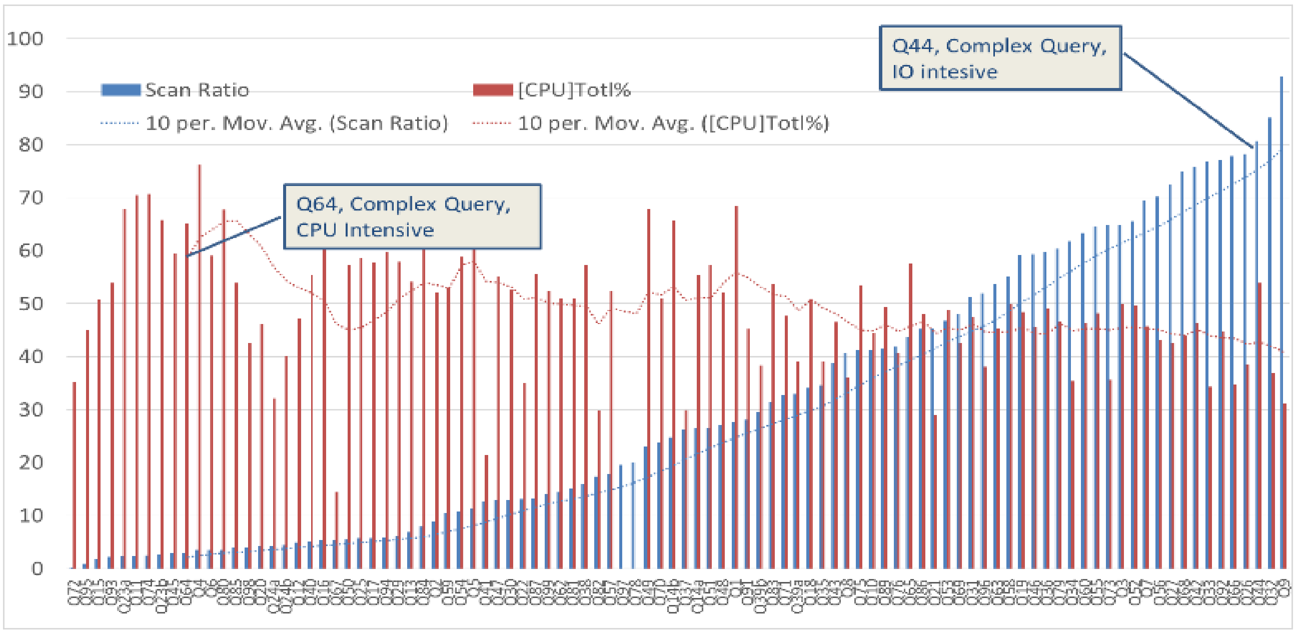


Figure 3: TPC-DS Scan Ratio and CPU utilization

5.1 Offloading components (or kernels)

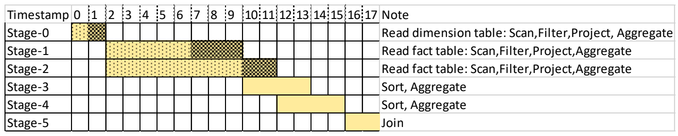
In this section, we exploit opportunities to offload operations from host to computational storage. In order to execute a query, data flows from the leaves of the plan to the root. Usually the leaves contain some form of SCAN operation: table rows and columns are read in (usually from disk, unless this data was previously cached). The SCAN operation usually includes some sort of data transformation, from the format on disk to the one understood by the database engine. Once a table is scanned (or sometimes while the table is being scanned), rows may be filtered or projected. Next plan steps may contain aggregates, sorts, joins, window functions, or other advanced data transformations. Operations near the leaves will generally be "easier" to push down to computational storage. Basic SCANs, FILTERs, and PROJECTIONs may happen with virtually no change to the database engine query plan. More aggressive push down optimizations are possible, but require the cooperation of the database engine, and re-factoring of the query plan.

For example, in Figure 1 (B), we observe this pattern in both FACT table and DIMENSION table I/O. By combining "Scan," "Filter" and "Project" into a new building block, we can estimate the performance benefit of offloading this new building block ("Scan/Filter") to computational storage. Regardless, with "Scan/Filter" offloading, the SPARK-SQL plan for Query 44 still looks the same.

5.2 SPARK-SQL model methodology

The performance estimate model for SPARK-SQL is based on how the database engine plan is executed – in stages with dependencies. We assume there is no resource limitation on the number of stages that can be executed concurrently.

For example, Figure 4 shows a generic query that involves 3 tables, 1 DIMENSION table and 2 FACT tables. Stage-0 reads the content of the DIMENSION table, while reading FACT tables happens in Stage-1 and Stage-2. Then, Stage-3 and 4 sort the results from Stage-1 and 2. The results are subsequently passed to Stage-5 for the final Join operation.



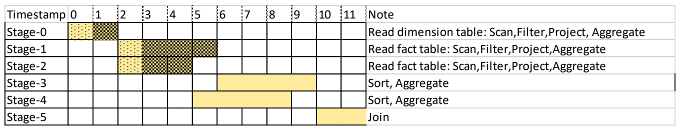
Timestamp	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	Note
Stage-0	░	▓																	Read dimension table: Scan,Filter,Project, Aggregate
Stage-1			░	░	░	░	░	▓	▓	▓									Read fact table: Scan,Filter,Project,Aggregate
Stage-2			░	░	░	░	░	░	░	░	▓	▓							Read fact table: Scan,Filter,Project,Aggregate
Stage-3											🟨	🟨	🟨	🟨					Sort, Aggregate
Stage-4													🟨	🟨	🟨	🟨			Sort, Aggregate
Stage-5																	🟨	🟨	Join

Figure 4: Query Stage Scheduling

First 3 stages (0, 1 and 2) include Scan/Filter/Project operations as marked with light dot shade in Figure 4. The time spent on the operations are 1, 5 and 8 seconds respectively, and could be offloaded to computational storage. The offloaded execution time is calculated as:

- Reserve 1 second for offloading-related handshaking. The reserved time is an arbitrary number.
- Assumes that the Filter runtime on the device is at wire speed and can be omitted. This is an optimistic assumption that provides an upper bound for our analysis. The actual Filter runtime depends on compute/IO capabilities of the device, and can be further improved with pre-processing in the device.
- Time-of-result data transfer between the device and the host as calculated based on the device Read bandwidth specification; in this paper, 3GB/sec has been used.

With these assumptions, the example execution time can be reduced from 18 seconds to 12 seconds (see Figure 5).



Timestamp	0	1	2	3	4	5	6	7	8	9	10	11	Note
Stage-0	░	▓											Read dimension table: Scan,Filter,Project, Aggregate
Stage-1			░	▓	▓	▓							Read fact table: Scan,Filter,Project,Aggregate
Stage-2			░	▓	▓								Read fact table: Scan,Filter,Project,Aggregate
Stage-3							🟨	🟨	🟨	🟨			Sort, Aggregate
Stage-4						🟨	🟨	🟨	🟨				Sort, Aggregate
Stage-5											🟨	🟨	Join

Figure 5: SPARK-SQL Offload Model

As the most fundamental step in building the estimate model, we need to know the time spent for Scan/Filter/Project on each SPARK-SQL query stage. Fortunately, with SPARK-SQL the log file provides the following key logging information (Figure 6):

- $MeasWClockTime_{Stage}$: The Stage wall clock runtime
- $ThrTime_{AllOperations}$: Total execution time for the stage from all execution threads. This is not wall clock time
- $ThrTime_{S,F,P}$: The execution time break down for Scan, Filter, Project

With the above information, the estimated time spent on Scan/Filter/Project can be calculated as

$$EstWClockTime_{S,F,P} = MeasWClockTime_{Stage} * \frac{ThrTime_{S,F,P}}{ThrTime_{AllOperations}}$$

In addition to Scan time, we also consider the following:

- $WClockTime_{Setup}$ -- The time to initialize computational storage for offloading. We always assume one second for the estimation calculation.
- $WClockTime_{DataTransfer}$ -- The time required to transfer the results from the offloading device back to the host. It is calculated based the Read bandwidth of the computational device. In our model, the Filtered result is usually less than 0.5% of the results that are unfiltered. It would take only a fraction of a second to read back to the host, therefore we ignored it this time.
- With Parquet format, we assume that no Project operation or Project time is omitted.

With the above assumption, the estimated stage runtime with offloading for SPARK-SQL is calculated as:

$$EstWClockTime_{Stage} = WClockTime_{Setup} + EstWClockTime_{S,F,P}$$

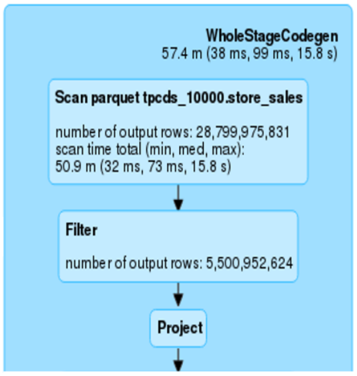
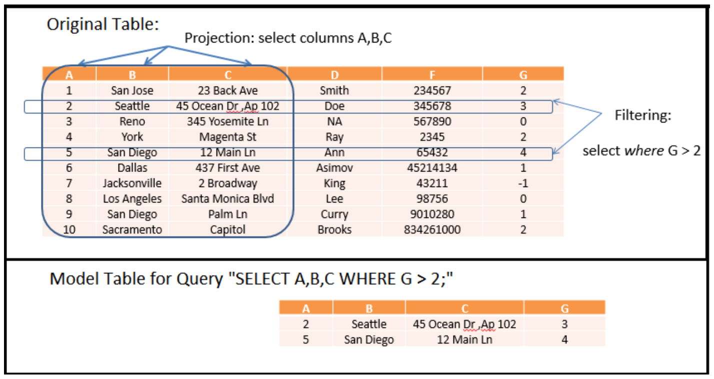

WholeStageCodegen
57.4 m (38 ms, 99 ms, 15.8 s)
Scan parquet tpcds_10000.store_sales
number of output rows: 28,799,975,831
scan time total (min, med, max):
50.9 m (32 ms, 73 ms, 15.8 s)
Filter
number of output rows: 5,500,952,624
Project

Figure 6: One SPARK-SQL Query Stage with Statistics

5.3 Presto model methodology

To model push down benefits of Scan/Filter/Project operations, we create and populate smaller tables we call "model tables." These "model tables" contain only the rows and columns that would be selected by a computational storage engine executing the Scan/Filter/Project operations defined by the query. We repeat the query using the model table, and compare results against the same query using the original tables – see Figure 7. For Presto, both original and model queries generate the same query plan. Similar to our SPARK-SQL model, the performance difference is the upper bound of the speed-up that a computational storage device would yield, because this model assumes that the storage device would be capable of filtering and projecting rows and columns at wire speed. However, if we take into consideration the higher internal flash storage bandwidth [4], this is a realistic approximation of the expected speed-up.


Original Table:
Projection: select columns A,B,C
Filtering:
select *where* G > 2
Model Table for Query "SELECT A,B,C WHERE G > 2;"

A	B	C	D	F	G
1	San Jose	23 Back Ave	Smith	234567	2
2	Seattle	45 Ocean Dr ,Ap 102	Doe	345678	3
3	Reno	345 Yosemite Ln	NA	567890	0
4	York	Magenta St	Ray	2345	2
5	San Diego	12 Main Ln	Ann	65432	4
6	Dallas	437 First Ave	Asimov	45214134	1
7	Jacksonville	2 Broadway	King	43211	-1
8	Los Angeles	Santa Monica Blvd	Lee	98756	0
9	San Diego	Palm Ln	Curry	9010280	1
10	Sacramento	Capitol	Brooks	834261000	2

A	B	C	G
2	Seattle	45 Ocean Dr ,Ap 102	3
5	San Diego	12 Main Ln	4

Figure 7: Presto Offload Model.

6 Offloading Evaluation

Here, we describe in detail the query selection process, and give a high-level view of the results obtained by the modeling of both database engines. Furthermore, we present side-by-side analysis of the expected speed-up for a few selected queries.

6.1 The queries

In this study, we picked five queries from each configuration for deep analysis. The queries were selected based on where they fall on the different quadrants of the Scan Ratio versus a CPU utilization chart (see Figure 8) to cover a wider range of characteristics. Because we focus on offloading Scan/Filter/Project, we want queries that are I/O intensive and show high selectivity when filtering and projecting FACT tables. That is, we look for queries of the "needle in the haystack" variety. Three of the queries (Q9, Q44, and Q75) are found in both studies, while the other two are found exclusively in either SPARK-SQL or Presto. We chose this approach because, due to their different architecture and optimizer, interesting queries in one environment are not necessarily interesting, or possible, in the other. For example, Presto cannot execute Q4 (out-of-memory error).

Using the chart in Figure 8, we selected the following five SPARK-SQL queries for analysis: Queries 9 and 44 have high Scan/Filter ratio; Query 4 has high CPU utilization; Query 72 has the longest runtime and higher CPU utilization; and Query 75 has a balanced Scan/Filter ratio and CPU utilization.

Based on our analysis of the query plans generated by Presto, we believe that Query 44, Query 49, and Query 76 are the natural candidates for near storage FILTER because they are the ones that filter out the largest portions of FACT tables. Furthermore, these are all SCAN heavy queries (Figure 8). Another two, Query 9 and Query 75, are on the "Top Ten" Presto list both in terms of Scan operations and complexity (number of fragments, or stages), and are queries that appear in the SPARK-SQL study.

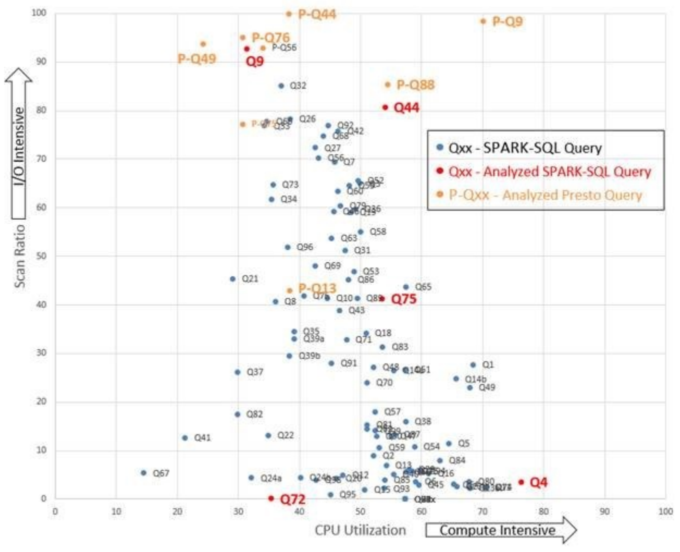


Figure 8: SPARK-SQL Scan Ratio vs CPU utilization

6.2 Characterization and performance summary – SPARK-SQL

Table 2 summarizes the characterization and offloading estimation for the SPARK-SQL queries identified above. We did not find any cluster NVMe Read bandwidth bottleneck. The highest peak Read bandwidth is ~2GB/sec. for Query 75, which is less than 3GB/sec of the NVMe Read bandwidth specification.

For some queries, the CPU utilization can become the bottleneck at several stages. In the presence of a computational storage device, the CPU utilization should benefit from Scan/Filter offloading, but we did not explore this topic for SPARK-SQL, and in our model we assumed that CPU utilization is unchanged by Scan/Filter offloading.

The benefit of Scan/Filter offloading ranges from no speed-up to ~8.17x in query runtime. Scan Ratio, CPU utilization and SQL execution plan all contribute to this speed-up, and will be analyzed in detail in sub-section 6.4 below.

Table 2: SPARK-SQL COMPUTATIONAL STORAGE MODEL RESULTS

TPC-DS Query ID		Query 9	Query 44	Query 4	Query 72	Query 75
Query Characterization	# of SPARK-SQL Stages	31	9	19	14	19
	# of FACT Table Used	1	1	3	2	6
	# of DIMENSION Table Used	1	1	1	7	2
	# of FACT Table Scan	15	3	3	2	9
	# of DIMENSION Table Scan	1	1	2	9	3
	# of Filter	16	8	9	9	9
	# of Project	16	11	22	19	28
	# of Aggregate	30	7	12	2	8
	# of Sort	0	4	18	4	14
	# of Join (All Join Types)	0	3	17	10	19
System Metrics	Runtime (sec.)	212	89	849	8205	298
	Scan Ratio	93%	81%	4%	0%	41%
	Total Data Read (GB)	1,276	605	940	168	1,182
	Average CPU Utilization	31%	54%	76%	35%	53%
	Peak CPU Utilization	94%	99%	100%	95%	100%
	Average NVMe Bandwidth (MB/s)	146	292	257	5	258
	Peak NVMe Bandwidth (MB/s)	1,365	1,625	1,008	1,026	1,918
Offloading Estimation	Runtime with Scan/Filter Offloading (sec.)	26	25	780	8190	144
	Speedup from offloading Scan/Filter	8.17x	3.61x	1.09x	1.00x	2.07x

6.3 Characterization and performance summary – Presto

Figure 9 shows speed up for 10 queries we modeled, including the 5 selected queries we analyze in detail. Presto speed-up from computational storage modeling varies from no speed-up for queries that are not I/O bound, to an impressive 59.3x for Query 44. Let's look at how this happened. In Table 3, we list the primary characteristics and system metrics for each selected query. For Presto, CPU utilization is never above 80% busy for the queries tested. Most issues arise from less than optimal query plans, and the queries that failed ran out of DRAM memory.

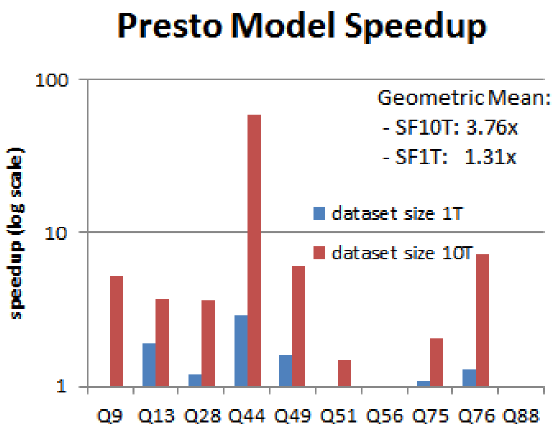


Figure 9: Summary Presto Speed-up

Table 3: PRESTO COMPUTATIONAL STORAGE MODEL RESULTS

	Query ID	Query 9	Query 44	Query 49	Query 75	Query 76
Query Characterization	# of Presto Fragments*	31	18	19	44	17
	# of FACT Tables Used	1	1	6	6	3
	# of DIMENSION Tables Used	1	1	1	2	2
	# of FACT Table Scan	15	4	6	12	3
	# of DIMENSION Table Scan	1	2	3	12	6
	# of ScanFilterProject FACT Table	**0**	**4**	**6**	**0**	**2**
	# of ScanFilterProject DIMENSION Table	1	0	3	6	0
	# of Window functions	0	2	6	0	0
	# of Joins	15	5	6	19	6
	10TB dataset	**original**	**original**	**original**	**original**	**original**
System Metrics	Runtime(seconds)	338	1126	719	1555	314
	Scan Ratio	99%	100%	94%	77%	95%
	Total Data Read (GB)	14	63	86	118	47
	Average CPU%	70	38	24	31	31
	10TB dataset	**model**	**model**	**model**	**model**	**model**
Model	Runtime(seconds)	64	19	125	821	43
	Total Data Read (GB)	9	0.12	19	81	0.75
	Average CPU%	47	9	41	43	19
	Runtime Speedup Original/Model	5.28x	59.3x	6.1x	2.05x	7.3x

* Equavilant to Stage from SPARK-SQL

6.4 Characterization and performance details – SPARK-SQL and Presto

Now let's examine the SQL plan, CPU, and I/O for the seven queries described in section 6.1 above.

Query 9 is the Highest Scan Ratio SPARK-SQL query, and one of the highest for Presto too (see Figure 8). Both SPARK-SQL and Presto follow the same query plan for Q9: They scan and filter the same FACT table 15 times with differing filter values. An Aggregation operation follows each Scan/Filter, the results are used for final joining with a DIMENSION table. All 15 FACT table Scan/Filter stages start simultaneously and are followed by their Aggregation stage (see Figure 10). Within the Scan/Filter stage, the first half involves most I/O operations (Scan), and the second half mostly performs the Filter function. Although all 15 Scan/Filter stages start at the same time, because of SPARK-SQL executor limitations, not all workers get scheduled immediately. Some stages have to wait for resources. This is reflected in the I/O chart (Figure 10). The I/O bandwidth peaks at the beginning. As the stages start Filter operation, CPU get busier and I/O bandwidth decreases. As the stages complete and release resources to the next waiting stage, I/O bandwidth goes up and CPU utilization goes down. We see this I/O spike after four Scan/Filter stages complete.

Figure 11 illustrates the offloading performance estimate of computational storage with SPARK-SQL. Blue bars show the measured stage execution time and red bars show the estimated stage execution. Stage dependency is unchanged.

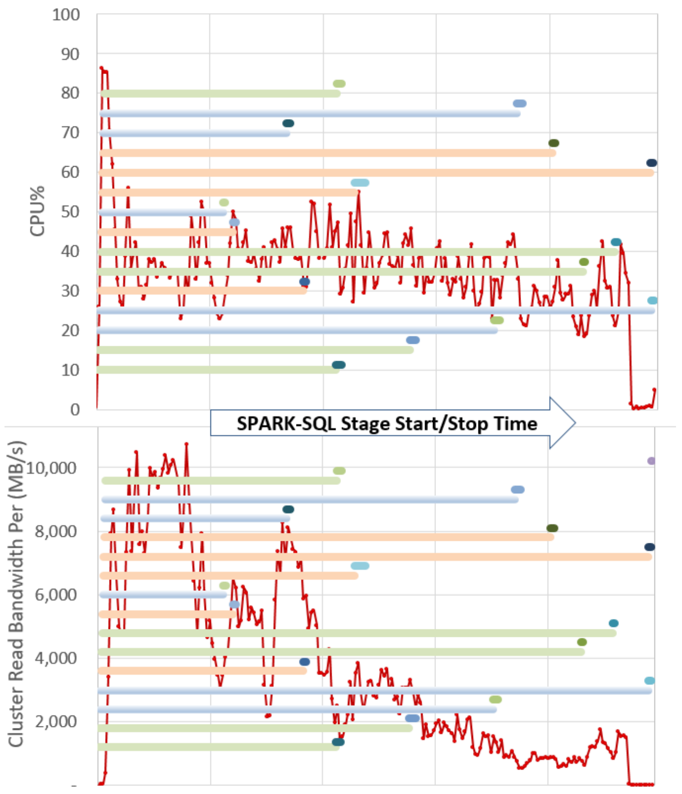


Figure 10: SPARK-SQL Query 9 CPU Utilization and Cluster Read Bandwidth

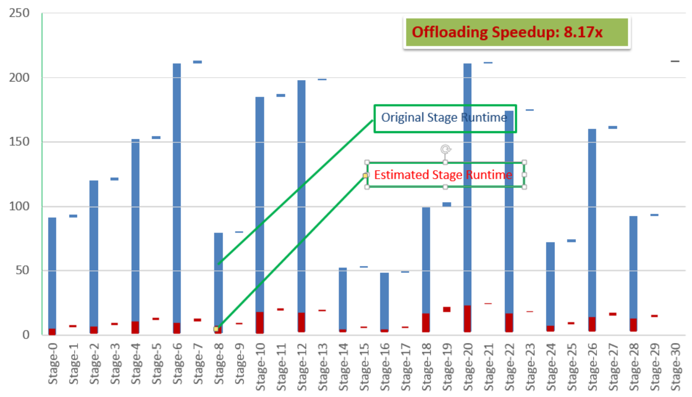


Figure 11: Query 9 SPARK-SQL Stage Breakdown with estimation

In our study, we only model Scan, Filter and Projection, but Query 9 also stands to benefit from Aggregate Pushdown, since the 15 scans result in 15 single, aggregated values. Because the Presto schema partitions table store_sales by ss_quantity, Query 9 does not significantly benefit from FILTER. The gains observed at the higher scale factor happen because of an artifact of the Presto model process. For Query 9 with 10TB dataset, the total I/O ratio between the original query and the Presto model is comparatively small: 1.54x. This I/O savings is not enough to justify the 5.3x speed-up observed at 10TB (Table 3). Our hypothesis is that this was caused by the modeling, which generated five smaller tables – while the model reads from five different tables (each three sets of workers reads from one table), the original query reads 15 concurrent times from one table. To reinforce this point, notice that the I/O savings ratio of the second largest Presto speed-up -7.3x for Query 76 at 10TB, is 62.66x (see Table 3).

However, Query 44, is the one query displaying dramatic speed-ups from Presto modeling. This happens because we have a lot of filtering that is increasing at scale. Query 44, which scans FACT table store_sales four times, is filtering rows where column ss_store_sk is equal to 2. With 1TB dataset, Query 44 uses only 0.15% of store_sales rows, and with 10TB dataset, it uses only 0.13% of store_sales rows. For both SPARK-SQL and Presto, Query 44 is a High Scan Ratio query with a CPU-intensive query operation. Notice that Presto and SPARK-SQL plans for Query 44 are different (Figure 12). The SPARK-SQL plan is smart enough to see a repeat subquery, execute it only once, and to broadcast the small dimension table. Presto plan is not scalable, and benefits immensely from the I/O savings afforded by the computational storage speed-up.

Query 44 plan	SPARK-SQL	Presto
# Stages/Fragments	9	18
# Fact Table Scans	3	4
# Dimension Table Scan	1	2
# Joins	5	5
# Sorts	4	0

Figure 12: Query 44 plan compare

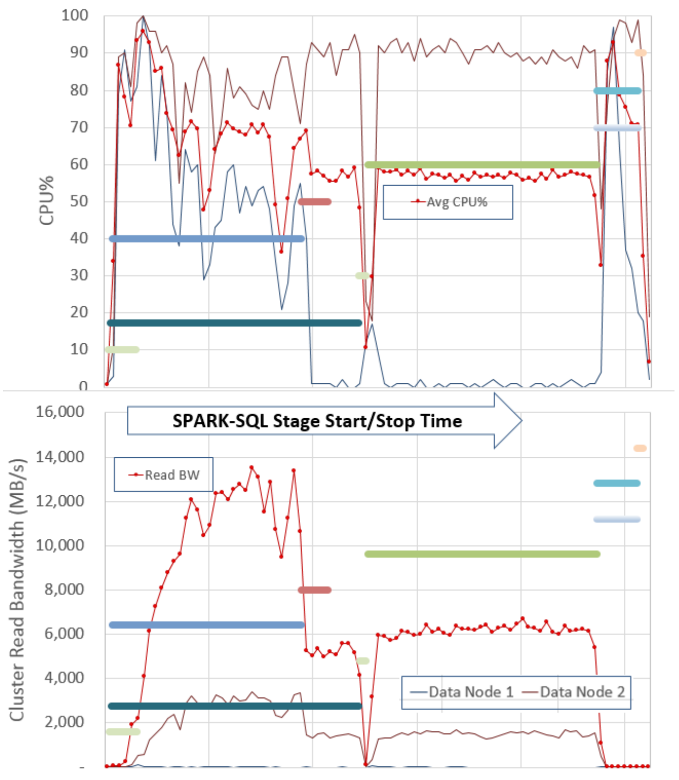


Figure 13: SPARK-SQL Query 44 CPU Utilization and Cluster Read Bandwidth

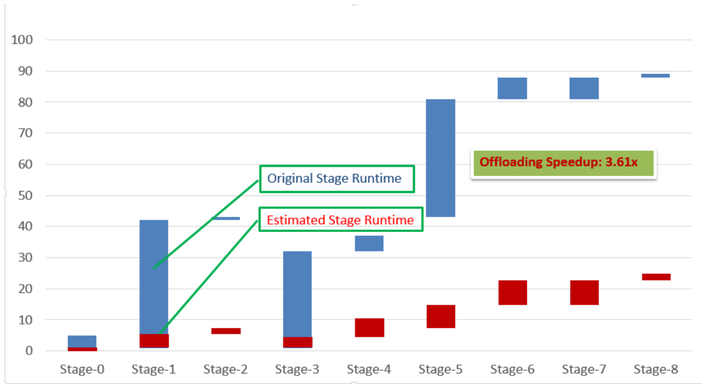


Figure 14: Query 44 SPARK-SQL Stage breakdown with estimate

For SPARK-SQL, Query 44 has other CPU intensive operations, such as Sort and Join, and its average cluster CPU utilization is at ~54%, but because of SPARK-SQL's worker scheduling, not all Data Nodes are utilized. See, for example, in Figure 13, the 3rd fact table Scan/Filter in Stage 5 only uses up to four Data Nodes. Data Node D9 is idling while D10 is nearly saturated. We do not explore the offloading impact on CPU cycles for SPARK-SQL, but moving Filter operation to computational storage should relieve Data Node CPU utilization and further improve performance. SPARK-SQL speedup for Q44 is 3.61x (Figure 14).

For Presto, Query 49's model response time is 6.1x faster than the original query, our third best result. Response time went from 12+ minutes to 2+ minutes (see Table 3). Query 49 reads in 4.5 times more bytes than its model, and this savings impacts both response times and CPU utilization, which becomes more efficient with the model: average CPU busy % went from 24 with the original query to 41 with the model.

SPARK-SQL Query 75 is a balanced query with all six FACT tables being used plus two DIMENSION tables. All FACT table Scan/Filter processing can benefit from computational storage offloading, but some stages are CPU bottlenecked (see Figure 15), and the SPARK-SQL speed-up for this query is 2.07x (Figure 16). Similarly, Presto Query 75 scans all six FACT tables, but there is no filter opportunity, just projection. Still, even though there is no speed-up for Presto, at 1TB we see excellent speed-up at 10TB: query response time went from 26 minutes to 13+ minutes. This result shows that the Parquet reader used by Presto may not be adequately implementing projection, while the Spark Parquet reader is doing so.

For Presto, Query 75 behavior is similar to Query 9. Both queries display no speed-up with the 1TB dataset, but modest gains with the 10TB dataset. Query 75 shows less speed-up than Query 9 at 10TB. From Figure 17, we see another interesting pattern: both the original and model show a barrier around three minutes before query completion, when all CPU and I/O utilization for all servers is near zero. This moment is identified by a vertical green bar in Figure 17. The elapsed time gain from the model happens before that barrier — the original query runs for about 23 minutes while the model runs for about 10 minutes. From Figure 17, we see that the model is handling less I/O both before and after the barrier, but no elapsed time gain is observed after the barrier. Query 75 total I/O ratio between the original and the model is only 1.5x.

Query 4 is SPARK-SQL's most CPU-intensive query. It uses three FACT tables and one DIMENSION table with many Sort, Join and other operations. These operations saturate cluster compute resources and the CPU becomes the bottleneck (see Figure 18). Presto cannot execute Query 4 with the 10TB dataset – it gets an "out of memory" error. Because most query runtime is spent on CPU-intensive, non-I/O operations, the Scan/Filter offloading benefit is limited to 9% as shown in Figure 19.

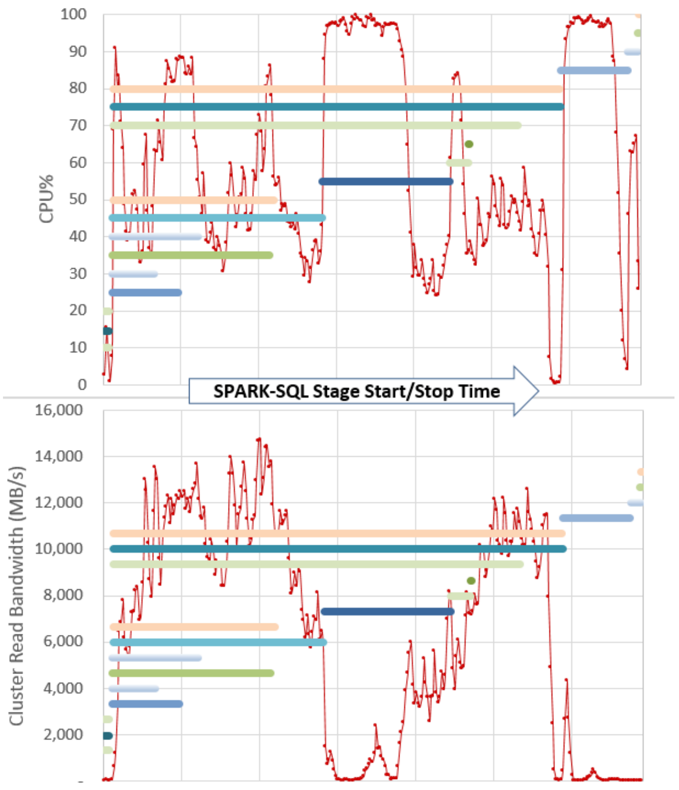


Figure 15: SPARK-SQL Query 75 CPU Utilization Cluster Read Bandwidth

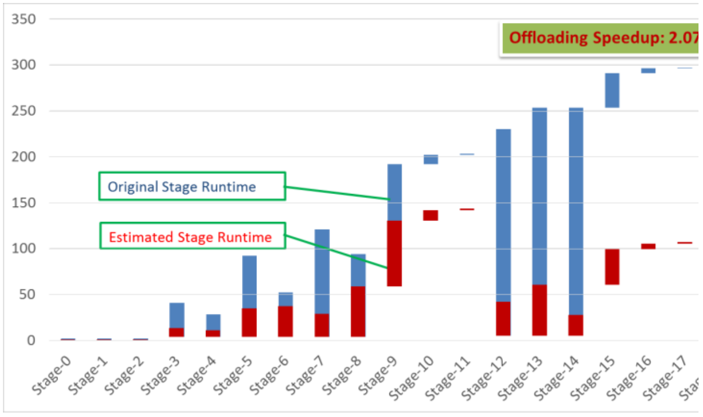


Figure 16: Query 75 SPARK-SQL Offloading estimate

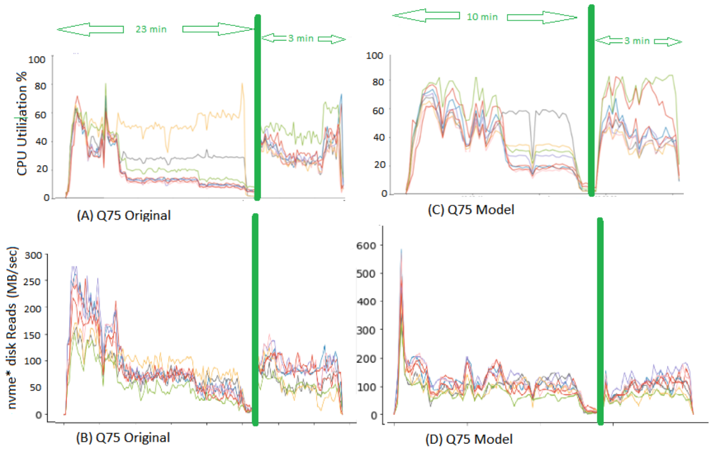


Figure 17: Query 75 Presto CPU and I/O activity

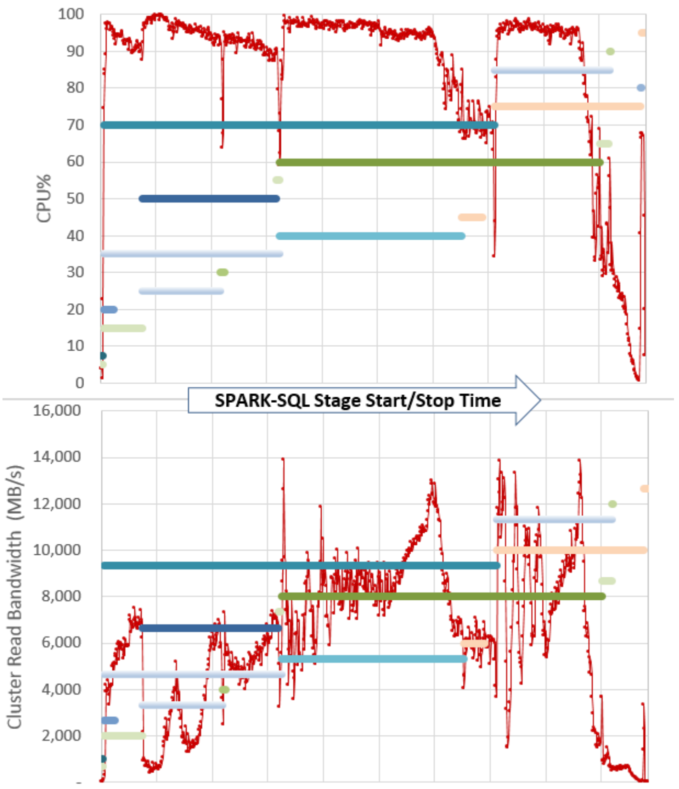


Figure 18: SPARK-SQL Query 4 CPU Utilization and Cluster Read Bandwidth

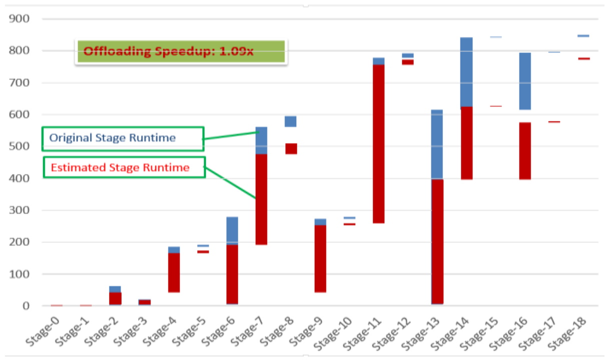


Figure 19: Query 4 SPARK-SQL Stage breakdown and estimate

Presto Query 76 filters and scans the three largest FACT tables (store_sales, web_sales, and catalog_sales). Furthermore, the selectivity is significant: only 4.50% of table store sales, 0.03% of web_sales, and 0.50% of catalog_sales are used after the filter operation. Consequently, we see excellent speed-ups for Presto at both scale factors. At 10TB, query response time went from 5+ minutes to 43 seconds, with the total I/O ratio between the original and the model an impressive 62.66x (see Table 3).

Query 72 has the longest runtime of all TPC-DS queries. It has 10 Join operations, and they are scheduled almost sequentially by SPARK-SQL within a single stage. Compared to the total runtime, the time spent on I/O counts only a small fraction. Because offloading is applied only on I/O, for this query, we observed no performance gain when offloading Scan/Filter.

Overall, for the computational storage operations being considered, everything is impacted by the selectivity of each filter and projection operation yield. And those yields can be substantial. For example, Query 44 reads 530 times more bytes than its model.

8 Thoughts on offloading other components

SCAN, FILTER, and PROJECTION are SQL operations that can be easily pushed down to computational storage. They are the proverbial "low hanging fruit." There are other operations that also wisely might be pushed down to computational storage, though some require cooperation from the database engine. For example, some aggregates, such as SUM, COUNT, MIN, MAX, are amenable to being pushed down even in a distributed environment. Other aggregates, such as AVERAGE and MEAN, can be partially pushed down, and would require active participation of the database engine. Furthermore, some JOINs, such as broadcast-join, can be pushed down. In the case of TPC-DS, for example, if dimension table DATE_DIM was replicated for all storage devices and its JOIN operations to fact tables were pushed down, this could potentially benefit 90% of the workload (89 queries) that scans and joins DATE_DIM.

9 Conclusion

This paper characterizes an Online Analytical Processing (OLAP) benchmark, TPC-DS, when implemented with a read-optimized, columnar Parquet format in the Hadoop ecosystem. We experimented with two database engines: SPARK-SQL and Presto. Furthermore, we modeled performance gains from pushing a few SQL building blocks to a computational storage device using Parquet, without any cooperation from the database engine. We showed that these gains can be substantial, but are not universal. Queries with high selectivity on the leaves of their plan with the largest tables benefit the most from such optimization. Queries with low selectivity in their SCAN operations, even if they are scan-heavy, see more modest performance gains per our modeling. Notice, however, that our models do not consider the cost to decompress and decode data from a storage format to an internal database format. It is worth noticing that scan-heavy operations may benefit significantly from performing decompression and decoding in storage, even if they present little or no filter opportunities.

Our main contribution is estimating the expected speedup from pushing down a few SQL building blocks (SCAN, FILTER, and PROJECT operations) to computational storage when using optimized, columnar Parquet format files. We demonstrate that these operations are not only universal and simple to offload, but that they may be implemented with little or no software changes for most database engines. As SmartSSD and other near storage computing technologies become available, we will see new opportunities and significant speedups for big data analytics and data mining.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their comments and suggestions to improve our earlier draft.

REFERENCES

[1] Samsung SmartSSD: https://samsungatfirst.com/smartssd/ Accessed August, 10,2019.

[2] NGD systems: https://www.ngdsystems.com/ Accessed August 10, 2019.

[3] ScaleFlux: http://www.scaleflux.com/ Accessed October 1, 2019.

[4] SIMMS https://www.simms.co.uk/tech-talk-2/sas-sata-or-pcie-know-your-interface/ Accessed 8/15/2019.

[5] G. Koo, et al. "Summarizer: Trading Communication with Computing Near Storage" MICRO'17, Oct 14-18, 2017, Boston, MA, USA.

[6] I. Jo, et al. "YourSQL: A High-Performance Database System Leveraging In-Storage Computing" Proceedings of the VLDB Endowment, Vol. 9, No 12, pp. 924-935, August 2016.

[7] B. Gu, et al. "Biscuit: A Framework for Near-Data Processing of Big Data Workloads" ISCA, Seoul, Korea, pp. 153-165, June 2016.

[8] J. Lee, et al. "ExtraV: Boosting Graph Processing Near Storage with a Coherent Accelerator", Proceedings of the VLDB Endowment, Vol. 10, No. 12, pp. 1706-1717, August 2017.

[9] J. Stuecheli, B. Blaner, C. Johns, M. Siegel. "CAPRI: A coherent accelerator processor interface". IBM Journal of Research and Development, 59(1):7:1{7:7, January 2015.

[10] K. Kohei, "GPCPU Accelerates PostgreSQL", DB Tech Showcase, Tokyo, Japan, November 2014.

[11] "Postgres Derived Databases", Documentation at https://wiki.postgresql.org/wiki/PostgreSQL_derived_databases. Accessed 6/12/2018.

[12] P. Francisco "IBM PureData System for Analytics Architecture" IBM White Paper, 2014.

[13] TPC Benchmark DS Standard Specification Version 2.10.1. www.tpc.org/tpc_documents_current_versions/pdf/tpc-ds_v2.10.1.pdf Accessed May 13, 2019.

[14] M. Poess, et al. "Analysis of TPC-DS the first standard benchmark for SQL-based big data systems", Proceedings of the 2017 Symposium on Cloud Computing, Santa Clara, CA, USA, pp. 573-585, September 2017.

[15] TPC-DS Top Results. www.tpc.org/tpcds/results/tpcds_advanced_sort.asp Accessed May 13, 2019.

[16] T. Ansley "Accelerating the Apache Hadoop 3.1-based Distribution Ecosystem with Flash Storage" www.micron.com/about/blog/2018/july/accelerating-the-apache-hadoop-based-distribution-ecosystem-with-flash-storage July 31, 2018.

[17] A. Thapliyal "Azure HDInsight Performance Benchmarking: Interactive Query, Spark and Presto" azure.microsoft.com/en-us/blog/hdinsight-interactive-query-performance-benchmarks-and-integration-with-power-bi-direct-query/ December 20, 2017.

[18] Transaction Processing Performance Council website www.tpc.org

[19] Apache Spark Documentation 2.4.3. spark.apache.org/docs/latest/ Accessed 8/6/2019.

[20] Presto Hive Connector. prestodb.io/docs/current/connector/hive.html Accessed 6/1/2018.

[21] Presto Documentation. prestodb.io/docs/current/overview.html Accessed 4/5/2018.

[22] B. Braams, "Predicate Pushdown in Parquet and Apache Spark" Master's Thesis. Univ. of Amsterdam. December, 2018.

[23] S. Melnik, S. et al. "Dremel: interactive analysis of web-scale datasets". Proceedings of the VLDB Endowment 3.1-2 (2010), pages 330-339.

[24] S. Pei, J. Yang, Q. Yang "REGISTOR: A Platform for Unstructured Data Processing Inside SSD Storage" SYSTOR, June 4-8, 2018, Haifa, Israel.

[25] Z. Ruan, T. He, J. Cong "INSIDER: Designing In-Storage Computing System for Emerging High-Performance Drive" USENIX ATC 2019, Renton, WA, USA.