SparkCruise: Workload Optimization in Managed Spark Clusters at Microsoft

Abhishek Roy
Microsoft
abhishek.roy@microsoft.com

Alekh Jindal
Microsoft
alekh.jindal@microsoft.com

Priyanka Gomatam
Microsoft
priyanka.gomatam@microsoft.com

Xiating Ouyang*
University of Wisconsin-Madison
xouyang@cs.wisc.edu

Ashit Gosalia
Microsoft
ashit.gosalia@microsoft.com

Nishkam Ravi
Microsoft
nishkam.ravi@microsoft.com

Swinky Mann
Microsoft
swinky.mann@microsoft.com

Prakhar Jain*
Databricks
prakhar.jain@databricks.com

ABSTRACT

Today cloud companies offer fully managed Spark services. This has made it easy to onboard new customers but has also increased the volume of users and their workload sizes. However, both cloud providers and users lack the tools and time to optimize these massive workloads. To solve this problem, we designed SparkCruise that can help understand and optimize workload instances by adding a workload-driven feedback loop to the Spark query optimizer. In this paper, we present our approach to collecting and representing Spark query workloads and use it to improve the overall performance on the workload, all without requiring any access to user data. These methods scale with the number of workloads and apply learned feedback in an online fashion. We explain one specific workload optimization developed for computation reuse. We also share the detailed analysis of production Spark workloads and contrast them with the corresponding analysis of TPC-DS benchmark. To the best of our knowledge, this is the first study to share the analysis of large-scale production Spark SQL workloads.

PVLDB Reference Format:
Abhishek Roy, Alekh Jindal, Priyanka Gomatam, Xiating Ouyang, Ashit Gosalia, Nishkam Ravi, Swinky Mann, and Prakhar Jain. SparkCruise: Workload Optimization in Managed Spark Clusters at Microsoft. PVLDB, 14(12): 3122 - 3134, 2021.
doi:10.14778/3476311.3476388

1 INTRODUCTION

Spark [4] is a widely popular data processing platform that is used for a variety of analytical tasks, including batch processing, interactive exploration, streaming analytics, graph analytics, and machine learning. At Microsoft, Azure HDInsight [26] offers managed Spark clusters that allow users to start processing their data processing without worrying about managing the underlying infrastructure. However, once the data processing tasks are deployed as production workflows, users are largely responsible for optimizing their workloads to achieve better performance at lower costs. This is fast emerging as a major pain in cloud data services, more so due to the lack of DBAs in the cloud environments [21], leading to several recent efforts for building new platforms and features that optimize end to end workloads in the cloud [22, 27, 29, 30].

Performance improvements in Spark have come a long way over the last decade. It started with the initial proposal of resilient distributed datasets (RDDs) in 2010 for improving the performance of multiple parallel operations by reusing a working set of data [33]. Later, Shark [31] was proposed in 2013 to run declarative Hive queries (SQL-on-Hadoop) interactively using the Spark processing backend. The Shark project evolved into Spark SQL [13] for doing relational data processing along with a query optimizer, called Catalyst, in 2015. While Catalyst was rule-based in the beginning, query costing and cost-based query optimization was later added to Spark in 2017 [1]. Most recently, given the broader realization that it is often hard to make the right query optimization choices at compile time [16], adaptive query execution was introduced in Spark in 2020 [3]. However, given the breadth of applications and deployment scenarios that are typically seen in modern data processing systems like Spark, it is still hard to pre-build the right set of optimizations in the system itself. This has led to a new wave of thinking to *instance optimize* a data processing system to a given workload [23, 30]. In fact, the presence of hundreds of configurations in current Spark codebase aligns with the above line of thinking that a Spark deployment could be tuned to different workload needs. Unfortunately, it is incredibly hard, if not impossible, to manually tune these configs or adapt the system to a given workload. Interestingly, modern cloud deployments of data processing systems offer an unprecedented opportunity to observe and learn from large volumes of workloads. As a result, we could build a workload-driven feedback loop to automatically (and continuously) tune the system from the workloads seen at hand.

We presented our overarching vision on how to improve cloud query engines in [21]. In this paper, we describe SparkCruise, the next big step in optimizing Spark workloads that we have built for

Proceedings of the VLDB Endowment, Vol. 14, No. 12 ISSN 2150-8097.
doi:10.14778/3476311.3476388

* Work done while at Microsoft.

Spark clusters in Azure HDInsight. SparkCruise exposes a workload optimization platform that leverages massive cloud workloads and provides a feedback loop to the Spark engine for improving performance and reducing costs. We demonstrated an early version of the SparkCruise system earlier [28]. Since then we have added newer techniques for plan log collection, introduced a scalable telemetry pipeline that runs daily, explored data cleaning and integration techniques to improve the quality of our common workload representation, analyzed production workloads to characterize the Spark workloads in HDInsight, provided a notebook for customers to derive insights from their own workloads, and pushed one concrete feature for automatic computation reuse all the way to production. We describe the overall system design and extensibility of SparkCruise, the opportunities for compute reuse in production Spark workloads, the deployment of SparkCruise in HDInsight, and the experiences from our production journey.

Our key contributions can be summarized as follows:

- We present the SparkCruise platform for adding workload-driven feedback loop in Spark, and discuss how it transforms Spark engine from optimizing one query at a time to optimizing end to end workloads. (Section 3)
- We describe a query plan telemetry pipeline for collecting anonymized Spark SQL plans with low overheads and at production scale. (Section 4)
- We introduce a denormalized workload representation for Spark that combines both the compile-time and run-time characteristics of the workload and could be used for a variety of optimization features. We discuss the data quality challenges in creating this workload representation and show cleaning techniques to overcome them. (Section 5)
- We present detailed insights from production Spark workloads at Microsoft, including distributions of inputs, applications, queries, operators, cardinalities, selectivities, and plan shapes such as width and height. (Section 6)
- We describe a workload insights notebook that we have built and released for customers in HDInsight to discover insights from their own workloads. (Section 7)
- Finally, we drill down into automatic computation reuse as a concrete workload optimization in Spark that we have built and released for customers in HDInsight. We discuss the reuse mechanisms and various online and offline policies for view selection and materialization. (Section 8)

2 SPARK BACKGROUND

The Spark data processing platform supports a variety of analytical applications including batch or interactive analytics over structured or unstructured data, streaming analytics over constantly arriving data, graph analytics over linked data, iterative machine learning algorithms, and the newer data science applications. Structured data processing, in particular, has increasingly gained enterprise level adoption in the last few years with several large companies running their key ETL workloads using Spark. This has resulted in several trends. First, Spark has become the most active Apache project that is visited on GitHub [18], with a vibrant open source community of 83 committers [19] and numerous meet-ups around the world [12], Second, there is in-house Spark development at several large enterprises such as LinkedIn [24], Facebook [11], and IBM [8], and Third, there are managed Spark services from all major cloud providers, including Amazon Web Services [5, 9], Microsoft Azure [6, 7, 26], and Google Cloud [10].

At Microsoft, Azure HDInsight allows customers to *run popular open source frameworks — including Apache Hadoop, Spark, Hive, Kafka, and more* [26]. Essentially, it abstracts the complexities in setting up and maintaining the cluster, and providing a more managed experience for customers to quickly get started with their analytical tasks. For Spark, this means that users can leverage the latest Spark distributions, easily configure their cluster for different application needs, and monitor and tune the performance and costs. As a result of this better Spark infrastructure experience, we find a large fraction of HDInsight customers running their recurring ETL workloads. Others prominent use of HDInsight Spark is for interactive notebooks that have become very popular for ad-hoc analysis. Interestingly, workload optimization is relevant to both these usage types: for saving total costs in ETL workloads and for reducing the time to insights in interactive workloads.

Spark workloads are made up of applications, each of which consist of one or more queries running in the Spark session. Multiple applications can run in parallel on the same cluster. We focus on Spark SQL queries, i.e, all analytics that compile down to Spark dataframes and go through the Catalyst query optimizer, while ignoring the programs written directly against the RDDs. This is because declarative Spark SQL workloads are more amenable to characterization and feedback in the query optimizer layer (without affecting the user expectation on how the programs should be executed, as with RDDs), not to mention they also form the majority of our workloads.

In the remainder of the paper, we first provide an overview of SparkCruise, our workload optimization platform for Spark, before describing each of its components and discussing the features we have shipped in HDInsight.

3 SPARKCRUISE OVERVIEW

SparkCruise adds a workload-driven feedback loop to Spark to *instance optimize* its performance for a given workload. Figure 1 shows the overall architecture. As mentioned before, we focus on Spark SQL queries that run through the Catalyst query optimizer and that users expect the system to optimize, as opposed to RDD programs that are almost like physical execution plans handcrafted by the users. There are four sets of components in Figure 1 that are worth highlighting and we discuss them below.

First, SparkCruise provides an elaborate query plan telemetry that captures Spark SQL query plans in a scalable manner. This includes an additional plan log listener to collect plans in JSON format, adding identifiers called signatures at each node in the query plan, anonymizing the plans from any personally identifiable information (PII), and collecting the resulting log in both structured and semi-structured format with varying degree of retention. The query plan telemetry is enabled simply via a configuration change and once collected it could be used for a variety of further analysis by both the service provider and well as the customer themselves.

Second, the workload collected above goes through a set of pre-processing to generate a common workload representation that

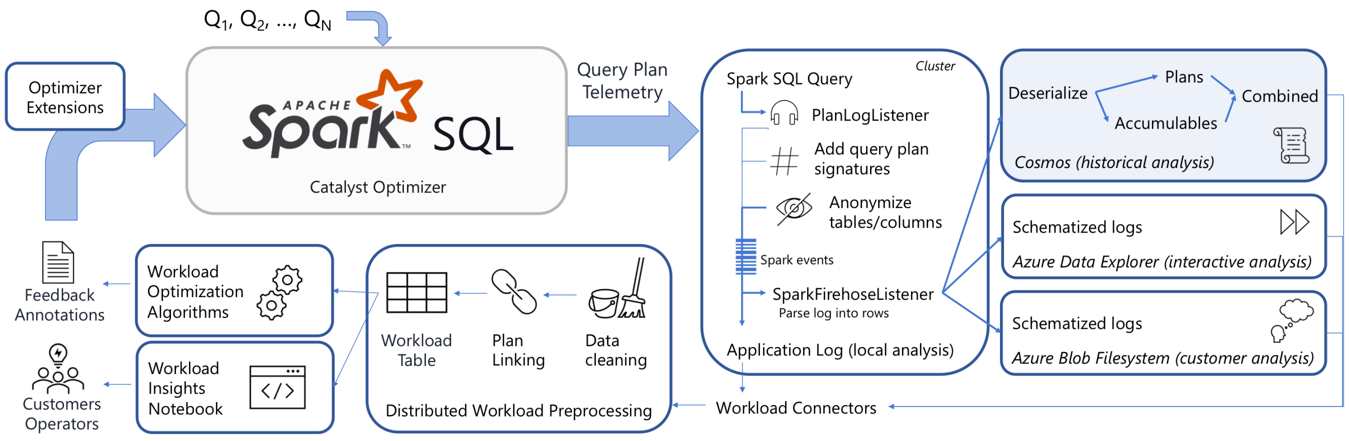


Figure 1: SparkCruise Architecture

could be used for running the actual workload optimization algorithms. This includes linking the operators in the logical and physical query plans with the stages in the Spark jobs execution, and cleaning the workload logs for missing or invalid values. The resulting workload is represented as a denormalized workload table. The workload preprocessing is run over large volumes of workloads in a distributed manner, i.e., using Spark itself for scaling out the preprocessing tasks. Furthermore, the preprocessing is done once and shared across all of the downstream analyses and optimizations.

Third, the workload table generated above could be used to derive better understanding and insights. Such insights are helpful for both the service operators as well as the customers. SparkCruise provides a workload insights notebook to quickly analyze the workload table generated for their workloads. The notebook also comes with a number of pre-canned queries to easily capture the shape, size, performance, and cost of the workload.

Finally, the workload table is also used to run optimization algorithms such as materialized view selection [20]. By providing a common workload representation consisting of both the compile-time and run-time characteristics, SparkCruise democratizes the development of newer workload optimization algorithms based on a variety of opportunities, e.g., learning cardinality models from past workloads. We serialize the optimization output into a feedback file and provide that to be loaded into the Catalyst optimizer for future optimization. The actual optimization action is performed by adding extra optimizer rules using the Spark extensions API, thus turning it into a self-tuning system.

In the following sections, we delve into each of the above four components of SparkCruise in more detail.

4 QUERY PLAN TELEMETRY

Workload optimizations in SparkCruise are rooted in analyzing and improving the Spark SQL query plans. Therefore, collecting Spark SQL query plans is at the core of SparkCruise. Our design requirements for adding this observability are five-fold:

(1) We do not want to make changes in the core Spark codebase. This is because we want the query plan telemetry to be easily collected with open-source Spark as well as with multiple deployments in different Microsoft products.
(2) Collecting plan telemetry should have minimal overheads to avoid impact on query performance. This means we should be reusing existing telemetry events and only add additional information wherever required.
(3) We need to identify patterns in the query plans in order to learn from past workloads and apply them in future queries. This requires to annotate query plans with signatures that could be used to identify and match interesting patterns.
(4) Query plans could contain sensitive information, particularly in the column and table names. Therefore, we need to scrub all such information and protect customer privacy.
(5) Finally, we need to support multiple scenarios where the query plans could be leveraged, from local debugging within the cluster to global workload analysis by the cloud provider.

Note that in contrast to the recently described Diametrics benchmarking platform at Google [15], SparkCruise captures a rich set of query plans (not just the SQL queries) along with associated metadata (run-time statistics to learn from the past behavior of those query plans), without copying the customer data. This is because in contrast to Diametrics, SparkCruise is not limited to internal customers and workloads and so access to customer data is not possible without explicit customer approval.

In the remaining of this section, we first describe query plan listener, plan annotations, and plan anonymization, before describing the telemetry pipeline.

4.1 Plan Listener

The first step in adding workload optimization capability to Spark is to collect the workload traces. Spark already collects telemetry at the level of both applications and tasks, however, the query plan-level details are incomplete. Specifically, the event logs contain Spark SQL query plans in text format, same as the one output by the EXPLAIN command, that is hard to parse and consume later on. Furthermore, many of the lines in the text plans are trimmed in case they are too long. So we need a more reliable format for query plan telemetry. Fortunately, the LogicalPlan object in Catalyst contains a method to serialize a query expression in JSON format. We have leveraged this by implementing a custom listener that logs the Spark query plans in JSON. The listener is invoked at the end of every query execution (i.e., SparkListenerSQLExecutionEnd event). Our listener emits the JSON plans from all four stages, namely the parsed plan, analyzed plan, optimized plan, and the physical execution plan. Later on, the workload optimization algorithms ingest workload traces and provide feedback to the query optimizer.

The above plan listener has very low overhead, e.g., in the order 0.5 seconds for TPC-DS queries. Still, we discovered that customers sometimes end up machine generating very large Spark SQL plans in their production workloads. These large plans could contain 10s of thousands of nodes in the parsed query plans, that lead to out of memory errors in one specific case. Therefore, we added additional checks to limit the plan logging for extremely large query plans. For example, if parsed and analyzed plans are found to be too big then we attempt to log only the subsequent plans (optimized, physical) as they are typically smaller than the initial plans.

4.2 Plan Annotations

Given that the goal of SparkCruise is to learn from the past Spark workloads and apply feedback to future queries, we want to identify patterns in query plans for future feedback. Therefore, we annotate every node in the query plan with identifiers, called *signatures*, that can be used for providing targeted feedback to future queries. Signatures are recursive hashes of the query plan nodes that capture both the node-level details as well as the query plan structure. We could further decide on which nodes and what levels of details in those nodes to include in the signature hash, thus capturing different kinds of query plan patterns. We describe the two kinds of signatures that we provide by default, although, our design allows to easily add newer signatures: (1) *Strict Signatures* capture the complete information at the node level and its children. At the leaf level it also includes the dataset version. (2) *Recurring Signatures* also capture the information at the node level and its children. However, it ignores the literal values and the dataset version.

4.3 Plan Anonymization

SparkCruise only analyzes workload metadata, with no access to customer data. Still, the filenames, table names, and column names could potentially contain keywords or identifiers relevant to different customer businesses. Therefore, SparkCruise anonymizes Spark SQL query plans by obfuscating the column and table names. Furthermore, we also obfuscate any literal values in the query predicates (filter or join predicates) to avoid leaking any customer identifiable information. We apply the same obfuscation to parsed, analyzed, optimized, and physical plans. However, we preserve the column reference ids that track columns within a query plan from leaf to the root of the plan.

4.4 Telemetry Pipeline

We feed the annotated and anonymized query plans into a telemetry pipeline consisting of several end points, each for different set of scenarios, as shown in Figure 1. The annotated and anonymized query plans are emitted as Spark events that are captured by the SparkFirehoseListener and converted into rows, with one row for each event and all plans and other metadata as JSON values in that row. We then push these event rows into several backends: (1) an optional user defined Azure Blob Filesystem location for users to run their own workload analysis later on, (2) Azure Data Explorer tables for interactive analysis, typically by the service operators, with a smaller retention window, and (3) Cosmos storage (compressed formats) for historical analysis over larger time windows. The above Spark events are also collected in the application log on the local cluster, for any real-time debugging or analysis by the users themselves.

5 WORKLOAD REPRESENTATION

In this section, we describe the steps to transform raw events into a shared workload representation. A shared workload representation removes the time consuming step of data collection and integration from each optimization algorithm. We also explain the relational format of the workload representation and why it has been widely adopted by downstream optimization algorithms.

5.1 Plan Linking

At the end of query execution, SparkCruise collects different plans related to query processing, namely parsed plan, analyzed plan, optimized plan, physical plan, and executed plan. Each plan in this list is derived from the previous plan. However, Spark does not preserve the provenance information between individual nodes of the different plans. This makes it impossible to get the runtime cardinalities and costs for logical operators. Specifically, SparkCruise applies view materialization and reuse based on signatures computed on optimized logical plan, but the cost-based view selection needs to consider the cost of logical operators as well. So, we attempt to link the nodes from different plans during workload preprocessing.

We include two plan linking algorithms for logical and physical query plans: *(1) Top-down heuristic based* - This method starts from the root node and links nodes from two query plans using a set of predefined heuristics. For example, this method skips the Exchange operator that is present in physical plans but not in logical query plans. Similarly, a logical Join operator can be converted into Exchange, Sort, and SortMergeJoin physical operators. This method uses lexical similarity when matching nodes in logical plans and their corresponding physical implementations such as Aggregate and HashAggregate. *(2) Bottom-up similarity based* - We extended Cupid [25], a generic schema matching system for SQL tables, to match query plans. Cupid uses weighted lexical similarity and structural similarity score for every pair of sub-trees in logical and physical plans to find the best match between nodes. We use the same set of rules for lexical similarity as the top-down heuristic method. From our experience, we have found (1) to be preferable, especially as it is easier to debug. Going ahead, we would like to maintain the provenance information inside the Spark optimizer itself to avoid this post-processing linking of plans.

The next linking between nodes in Spark physical plans and executed plans involves a simple tree traversal. Figure 2 shows an example of the links between optimized logical plans and the executed plan. After the plans linked together, we can assign signatures computed on optimized logical plans to the cost of the

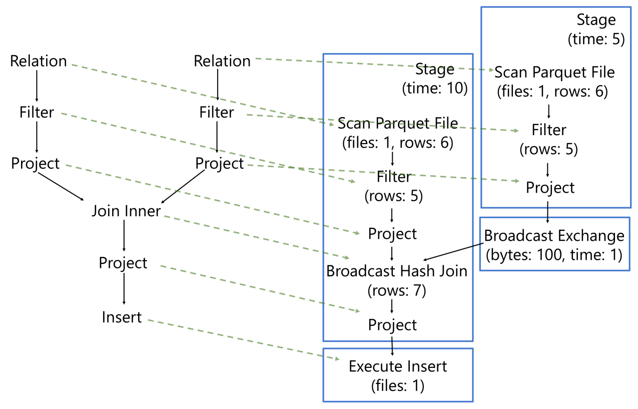


Figure 2: Linking of nodes in optimized logical plan (left) and executed plan (right).

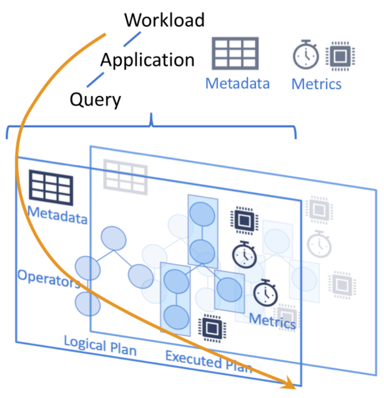


Figure 3: Linking different entities in Spark workloads.

corresponding physical operator. This allows us to develop cost-based workload optimization algorithms and apply feedback during query optimization using signatures.

5.2 Data Cleaning

At the end of query execution, Spark reports runtime metrics such as cardinality, time, and memory. However, these metrics are mostly at the stage level (multiple operators combined together) and few metrics are reported for individual operators. Thus, even after linking query plans we do not have metrics for most of the logical operators. To solve this problem we perform a few data imputation steps to assign metrics to individual operators. We describe two imputation strategies for row cardinalities and running time –

(1) Spark provides runtime cardinality metrics at different nodes in the query plans, generally for operators where cardinality changes such as scan, filter, aggregates, and joins. To get the cardinalities for remaining operators, we perform a post-order traversal of query plans and, if cardinality is missing, then we copy the cardinalities from the child node. For some operators such as Join, in case of missing cardinality, we take the maximum cardinality among all child nodes.

(2) Spark reports the running time for every stage in query plan. We divide the stage-level running time among the operators with missing running times in the stage node . With the imputed running time per operator, we can calculate the serial time cost of a subexpression in plan as the sum of times from individual operators. The serial time of subexpressions along with output cardinality is used by the cost-based view selection algorithms to select views with high savings potential and low materialization costs.

5.3 Workload Table

Workload table is the foundation for workload optimization algorithms. The workload table combines different entities in Spark applications, such as application metadata, query metadata, metrics, query plans, and annotations to create a tabular representation of the workload. In this section, we will describe the steps performed to create the workload table from raw events.

As explained in Section 4.4, the anonymized telemetry data can be stored in multiple locations (from storage accounts to databases) depending on the type of analysis. The collected telemetry is transformed to workload table by the Workload Parser Spark job. The Workload Parser job has connectors to read and write the data from different sources. Workload Parser can independently process the telemetry belonging to each Spark application. This allows the Workload Parser job to scale with the number of applications in the workload.

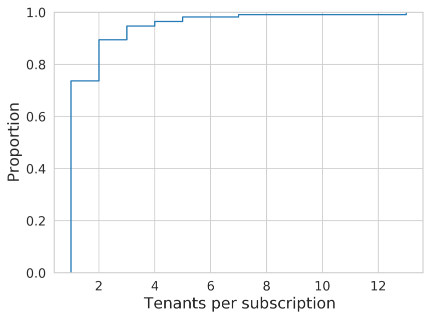
The Workload Parser job recreates the hierarchy of entities in a Spark workload. Figure 3 shows the typical entities in a Workload that include Applications, Queries, Plans, Metadata, and Metrics. These entities are parsed from the events in JSON format. For example, the preorder traversal of query plans is serialized in JSON format and the Workload Parser job recreates the query plan graph from the serialized format. The Workload Parser also performs the necessary plan linking and data imputation steps. Then, these linked entities in the workload are exported as a denormalized workload table. The workload table has one row per physical operator. Each row of the workload table contains the details of physical operator, application-level metadata, query-level metadata, linked logical operator details, and compile-time and run-time statistics. Table 1 shows a subset from the workload table. There is a lot of repeated metadata information in the workload table. We have found that having a single denormalized workload table with all the available information removes the need for complex data processing steps by downstream workload optimization algorithms. After this step, the workload table is available for processing by workload optimization algorithms and for visualization via Workload Insights Notebook.

6 PRODUCTION INSIGHTS

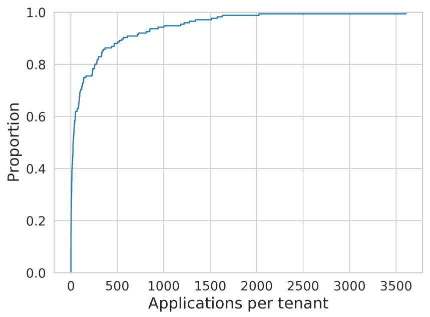
In this section, we show insights from production Spark workloads at Microsoft with the goal to understand the volume, shapes, sizes, costs, and other aspects of production Spark SQL queries. Later in the next section, we contrast these with TPC-DS queries, a popular benchmark for analytical workloads. We consider a large subset of daily workloads from HDInsight consisting of 176 clusters from 114 Azure subscriptions, and consisting of 34,834 Spark applications with 349,366 Spark SQL query statements (a mix of

Table 1: Illustrating a subset of attributes and rows from the workload table for TPC-DS workload. Each row in workload table corresponds to a physical operator. The columns include application-level metadata, query-level metadata, linked logical operator details (e.g., Signatures), compile-time (e.g., EstCard, RowLen) and run-time statistics (e.g., ActualCard, OpTime).

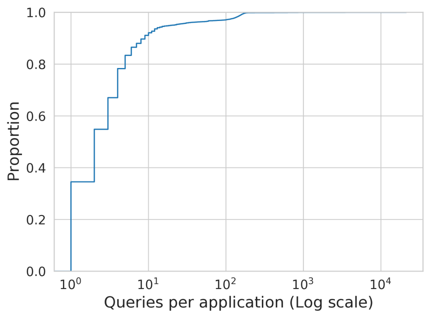
AppName	AppTime	QueryID	QueryTime	OpName	OpID	Logical	StrictSignature	EstCard	ActualCard	RowLen	OpTime
tpcds-q19	76360	1	49336	HashAggregateExec	3	Aggregate	4399905266039605409	0	2788	64	21733
tpcds-q61	106706	1	73540	FilterExec	42	Filter	7691322187098102260	250	71	420	2
tpcds-q85	95508	1	60751	BroadcastHashJoinExec	21	Join	18446744071904119269	6485337	1265878	124	18223



(a) CDF of tenants per subscription

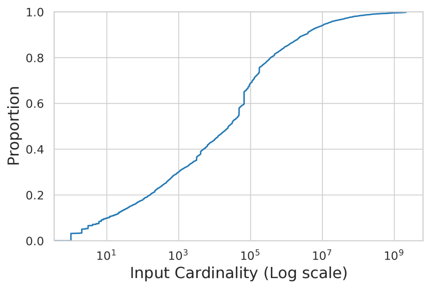


(b) CDF of applications per tenant

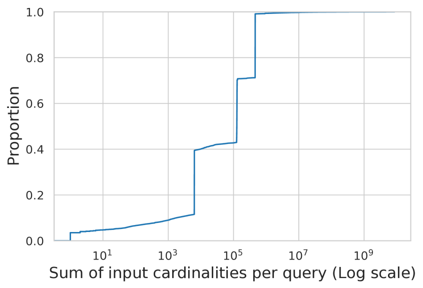


(c) CDF of queries per application

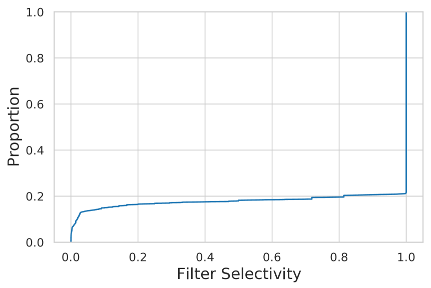
Figure 4: Distribution of tenants, applications, and queries among HDInsight customers.



(a) CDF of input cardinalities



(b) CDF of sum of input cardinalities per query



(c) CDF of filter selectivities

Figure 5: Input and filter cardinalities for Spark workloads.

streaming and non-streaming queries) and a total 1,438,411 query sub-expressions. Note that we already segregate all catalog queries, and so the above workload consists of purely the DML operations. We applied the same workload steps to this workload as described in the previous section, i.e., plan linking and data cleaning, and derive several insights from the resulting workload table. To the best of our knowledge, this is the first study showing detailed analysis from large-scale production Spark SQL workloads.

6.1 Subscriptions, Tenants, Applications

HDInsight customers with Azure subscriptions can create one or more tenants (or clusters). Each tenant can run multiple applications, and application can have multiple Spark SQL queries. Figure 4 shows the CDFs for the number of tenants per subscriptions, application per tenants, and Spark SQL queries per application. We can see that while most customers work with just one tenant per subscription, about 30% have multiple tenants for each subscription (see Figure 4a). We see a similar skew in the number of applications per tenant (see Figure 4b). Many tenants have very few applications, yet almost 30% tenants have 100*s* of applications per tenant and 5% have 1000*s* of applications per tenant. Finally, the number of queries per application demonstrates the largest skew (see Figure 4c). Most applications have less than 10 queries and only 5% have queries in double digits. It is interesting to note that the bulk of workload consists of shorter (in number of queries) applications.

6.2 Inputs

We now analyze the size of the inputs that are processed by the Spark SQL queries. Figure 5 shows the CDFs for input table cardinalities, per-query input cardinalities, and cardinalities after any filtering. From Figure 5a, we see that just over 10% of the input tables have more than $1M$ rows. This is interesting because Spark is often preferred for more interactive query processing and so many of the inputs processed by Spark are small. In fact, almost

30% of the inputs tables have less than $1K$ rows. Interestingly, if we consider the sum of all input table rows processed by each query, then Figure 5b shows that almost 60% of the queries process $10K$ or more rows. Thus, even though individual tables might be smaller, Spark processes several inputs in most queries. Finally, Figure 5c shows the filter selectivities in Spark SQL queries. We observe that 80% of the queries have selectivity of more than 20%, i.e., very few queries have highly selective filters. Many filters in the workload are passthrough like null checks on each record.

6.3 Operators

We now dig into the operators seen in our Spark SQL workloads. Figures 6a and 6b shows the distribution of logical and physical Spark SQL operators in the workload (Note that we skip the leaf level scans for Figure 6, and InputAdapter and WholeStageCodegen operators for Figure 6b). While Project is the most frequent logical operator, consisting of 27% of all operators, interestingly, Aggregate is the second most popular operator, comprising of more than 18% of all operators. This is followed by Filters, Limits, and Joins. Note that updates are very rare here. While the logical operators give a sense of what of kind of queries users are writing, physical operators (see Figure 6b) give a sense of how those queries are processed. We see that hash aggregation and shuffle are the two most common physical operators, followed by filter and projection respectively. This gives us an idea of what operations to optimize when trying to improve the overall performance and reduce costs. Indeed, internal teams at Microsoft have found such insights useful to make data-driven decisions when building new features.

Figure 7 shows the distributions of different logical and physical operators per-query. From Figure 7a, we see that most queries (almost 80%) have multiple joins, while some queries (around 20%) have many more projections, filters, and aggregates (up to 10 per query). From Figure 7b, we see more than 80% queries have 2 or more broadcast hash join operators. This is also true for broadcast exchange operators. The prevalence of broadcast operators indicates that many joins have small inputs on one side, that results in broadcasting that side. This is further confirmed in Figure 8a that shows that almost 90% of the joins are executed as broadcast hash joins, as compared to just over 8% that are executed as sort merge joins. Finally, Figure 8b shows the sizes of the intermediate data that gets shuffled. Surprisingly, 95% of queries shuffle less than $1MB$ of data. This is because many of these shuffles are aggregate shuffles that only move the partial aggregates around. Few percent of shuffles still move more than $100GB$ of data. Overall, data movement is not really a concern for this type of workload.

6.4 Queries

Let us now look at the plan shapes of the Spark SQL queries. Figure 9a shows the distribution of the number of operator per query. We can see that almost all queries have less than 20 operators. Furthermore, roughly 80% have 5 or less operators. Note the this is in contrast to batch processing systems like SCOPE [14] that allows users to write a sequence of SQL-like statements in a script and executes them as a single giant DAG of operators. Spark SQL queries on the other hand consists of much smaller operator DAGs. Figure 9b further illustrates the distribution of plan depth and width in Spark SQL queries. We see that 90% of queries have a depth of 10 or less. Plan width, on the other hand, ranges from 1 to 4 operators.

7 INSIGHTS NOTEBOOK

Given that generating and analyzing the workload table could reveal several interesting insights, we have made this process easier for customers by making a *Workload Insights Notebook* available in HDInsight [1]. Key features of this notebook include generating the workload table, analyzing this table using PySpark queries, pre-canned queries that could help get started quickly, e.g. identify workload shape, size, recurrence, etc. The notebook also shows prospective cost/benefits of workload optimization features such as statistics on selected views and their costs if they were reused. Users can now run the same analysis, as shown in the previous section, on their own local workloads. They can even use the notebook medium to write their own exploratory queries. Thus, the workload insights notebook can help users understand their specific workload and make data-driven decisions on how to optimize it.

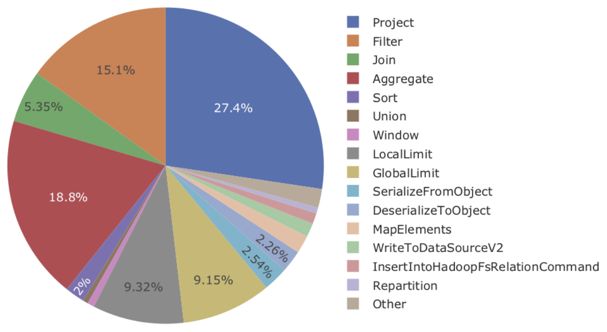
Below we revisit some of the insights discussed in the previous section, and contrast them with corresponding insights on TPC-DS ($1TB$ scale), a popular benchmark for decision support systems. The goal is to illustrate how different workloads can lead to very different insights for different customers.

Inputs. Interestingly, the input table sizes and the inputs sizes after filtering have very similar distributions in TPC-DS (see Figures 10a–10c) as also seen in the production workloads (see Figures 5a–5c). The filter selectivity, in particular, shows a strikingly similar plateau at around 20%, i.e., a large fraction of filters select almost everything.

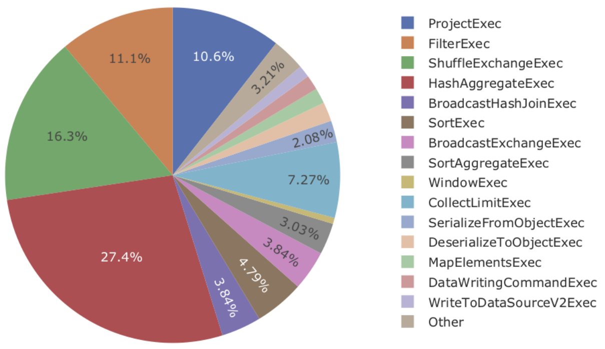
Operators. Figure 11 shows the logical and physical operator distribution in TPC-DS. Surprisingly, these are very different from the operator distributions seen in production queries in Figure 6. Considering logical operators, we see that the fraction of Projects go up from 27% to 40%, while the percentage of Aggregates come down by almost $3x$, from 18.8% to 6.5%. TPC-DS also has more than 3 times the joins than in production workloads, from 5.3% to 18.9%. The fraction of Filters though remains relatively comparable (15.1% and 23.9%). Considering the physical operators, we see that while hash aggregate was the most frequent in production workloads, it comes after projection, filter, and shuffle exchange in TPC-DS. Similarly, filter and shuffle exchange swap their places in relative frequencies. Finally, Figure 11c shows the distribution of join operator implementations in TPC-DS, compared to those shown in production workloads in Figure 8a. We can see that while broadcast hash join still remains the dominant join implementation, the percentage of sort merge join goes up $3x$ from 8.7% in production workloads to 25.3% in TPC-DS. Thus, we see that different workloads can have very different characteristics and the workload insights notebook can help customers understand their workloads better.

Queries. Figure 12 shows the shuffle and plan sizes in TPC-DS queries. Interestingly, we see that the fraction of large shuffles is higher in TPC-DS compared to production workloads; 30% of shuffles move more than $100GB$ in TPC-DS, compared to only percentages in production workloads. While the actual data movement

[1]https://github.com/Azure-Samples/azure-sparkcruise-samples/blob/main/SparkCruise/WorkloadInsights_HDI_v0.4.ipynb

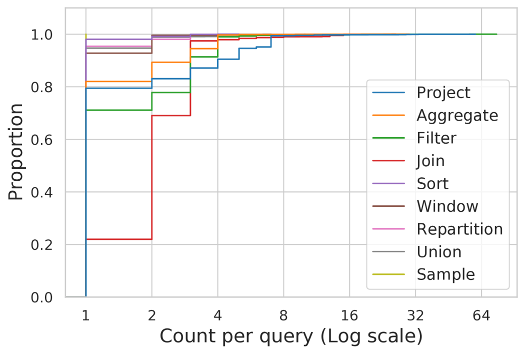


(a) Frequency of logical operators

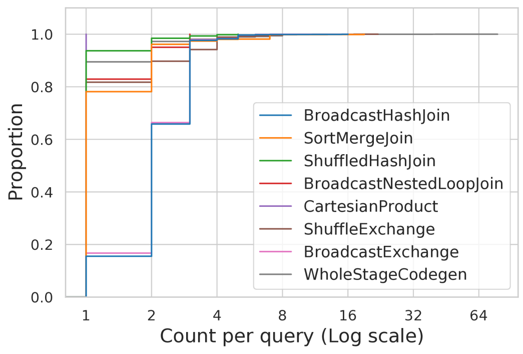


(b) Frequency of physical operators

Figure 6: Frequency of operators in Spark workloads.

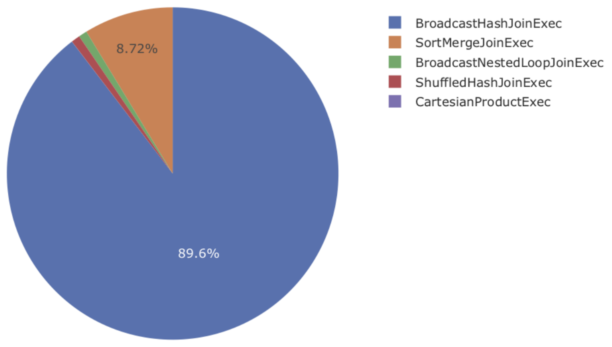


(a) Logical operators per query

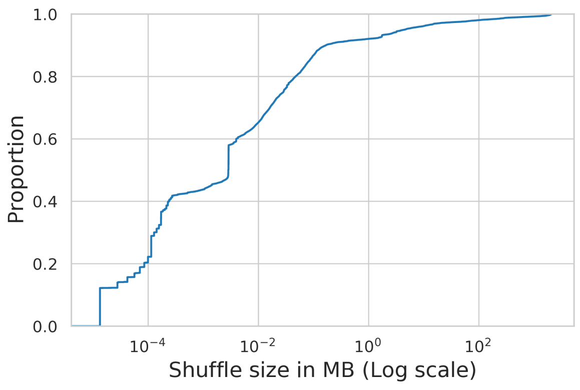


(b) Physical operators per query

Figure 7: Frequency of per-query operators in Spark workloads.



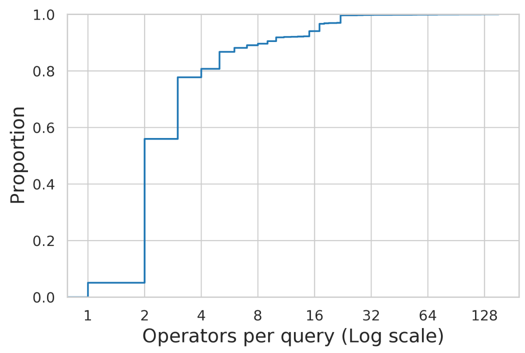
(a) Different join implementations



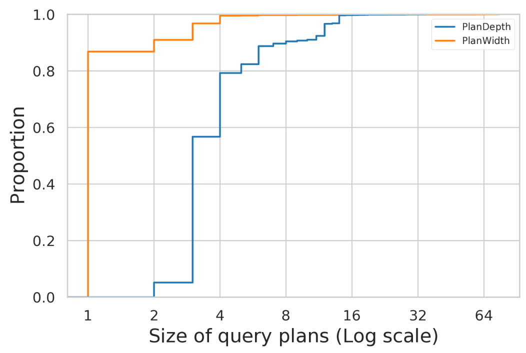
(b) Shuffle sizes

Figure 8: Distribution of join operators and shuffle sizes in Spark workloads.

size can vary with the TPC-DS scale factor, it is interesting to see that the shuffles sizes in TPC-DS are much more prominent. Figure 12b shows the distribution of the number of operators per query. We see that TPC-DS queries are much larger, with 20% queries having 50 or more operators compared to just 4 or more operators for production workloads. Likewise, we found many more projects, filters, and joins per query in TPC-DS. Finally, Figure 12c shows

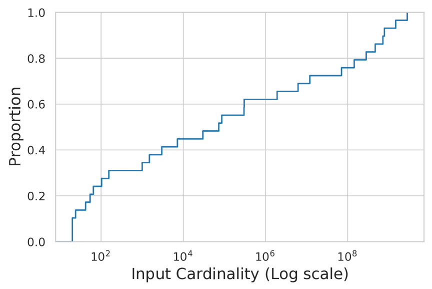


(a) CDF of operator count per query

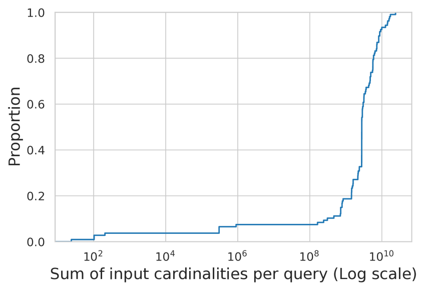


(b) CDF of plan depth and width

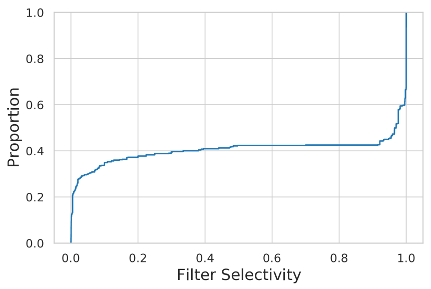
Figure 9: Optimized logical plan sizes in Spark workloads.



(a) CDF of input cardinalities

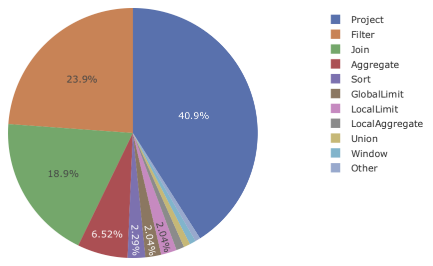


(b) CDF of sum of input cardinalities per query

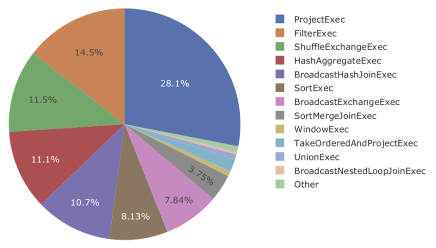


(c) CDF of filter selectivities

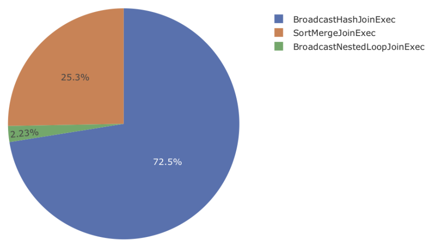
Figure 10: Input and filter cardinalities in TPC-DS workload.



(a) Frequency of logical operators



(b) Frequency of physical operators



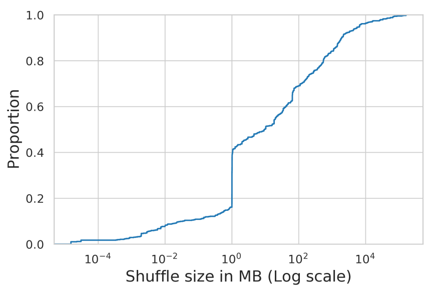
(c) Different join implementations

Figure 11: Frequency of operators in TPC-DS workload.

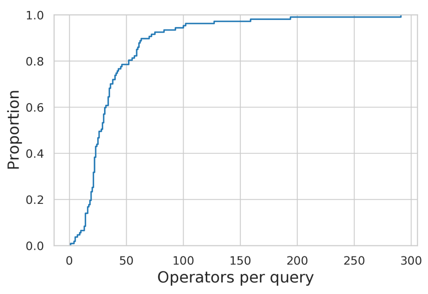
distribution of query plan depth and width. Again, we find deeper (depth 18 or more for 20% of the queries) and wider (width of 10 or more for 20% of the queries) plans in TPC-DS.

Redundancies. Figure 13 shows common subexpressions grouped by their root operators. These common subexpressions are candidates for view selection. We see that in TPC-DS workload, 58% of Filters are repeated more than once with an average repeat frequency of 5.69%. Not surprisingly, the percentage of common subexpressions decrease among operators that are closer to the root level of query plans. For example, only 12% of joins are repeated across queries with an average repeat frequency of 3.92%. However, since joins are generally expensive, they might be preferred over more frequent filters by the view selection algorithm.

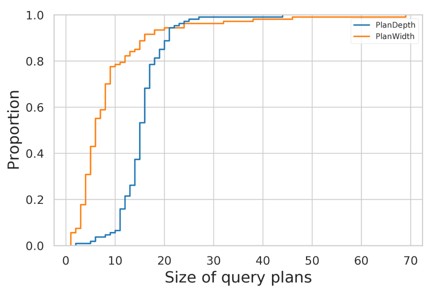
To summarize, carefully studying the query workload characteristics can reveal interesting insights for better understanding and potential optimizations, and workload insights notebook can help customers achieve those insights quickly.



(a) Shuffle sizes



(b) CDF of operator count per query



(c) CDF of plan depth and width

Figure 12: Shuffle and plan sizes in TPC-DS workload.

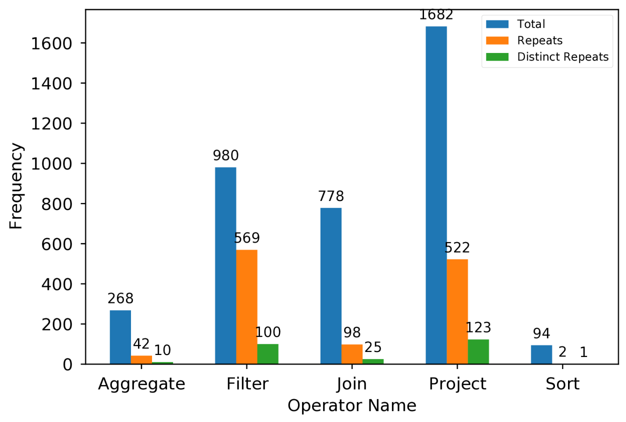


Figure 13: Repeat frequency of operators across queries in TPC-DS workload.

8 AUTOMATIC COMPUTATION REUSE

In this section, we describe a concrete workload optimization, namely reusing common computations across a variety of Spark SQL workloads, that we have built and made available for HDInsight customers to try out on their workloads [2]. Customers can first analyze their workloads and identify opportunities for reuse, they can then run the view selection algorithms and generate feedback for their workloads, and finally future Spark SQL queries automatically pick up that feedback for reusing common computations. While TPC-DS results and reuse mechanisms were highlighted in the earlier system demonstration [28], we quickly recap them below for completeness. The more interesting aspect in this paper, however, is our discussion over three workload scenarios, derived from our production experiences, and the conclusions on whether or not they are likely to benefit from computation reuse.

8.1 Workloads

SparkCruise supports both recurring and ad-hoc query workloads for computation reuse. Recurring workloads could consist of repetitive queries that are executed periodically with new input datasets and query parameters. For recurring workloads, we can run workload optimizations on the past query logs and generate feedback for future queries. However, there is another growing class of applications that use Spark for ad-hoc or exploratory analysis [2]. These workloads are non-recurring and we need online algorithms that can adapt with new queries. To illustrate the opportunity in popular workloads, Figure 13 shows the repeat frequency of operators in TPC-DS workload. We can see that numerous aggregates, filters, joins, projections, and even sorts repeat across queries. In fact, more than half of the filters repeat across queries, while 16% of the aggregates and 13% of the joins also repeat across multiple queries. Thus, analytical workloads typically have significant opportunities for computation reuse. This has been also shown in several earlier works in this area [20, 22, 32].

[2]https://docs.microsoft.com/en-us/azure/hdinsight/spark/spark-cruise

8.2 View Feedback

Once the workload table is generated via the workload representation process described in previous sections, SparkCruise provides view selection algorithms to select the most useful computations to materialize and reuse in a cost-based manner. The workload table contains strict and recurring signatures that capture the details of the subexpressions, or candidates for computation reuse. The strict signatures are used by the view selection algorithm to detect common computations across queries. The view selection algorithm consists of a set of heuristic rules to select views with high reuse potential and low materialization cost. The list of selected selected views are uploaded to a shared location via a feedback file or service. This list contains the recurring signatures and some metadata associated with each view. The recurring signatures enables SparkCruise to detect future occurrences of the same view patterns when the workloads are submitted again with new inputs or parameters.

8.3 Reuse mechanics

We have extended the Spark optimizer with two rules — Online Materialization and Computation Reuse that can read the feedback and modify the logical query plans. These two rules transform the traditional Spark query optimizer to a workload-aware optimizer. The Online Materialization rule materializes the results of a subquery if the workload-based feedback exists for the subquery and it has not been materialized. Similarly, the Computation Reuse replaces the sub-query with the scan of the materialized view to avoid its computation cost. Figure 15 shows automatic materialization of a common join by Query 42 and the subsequent reuse of the materialized view by Query 52 of TPC-DS benchmark.

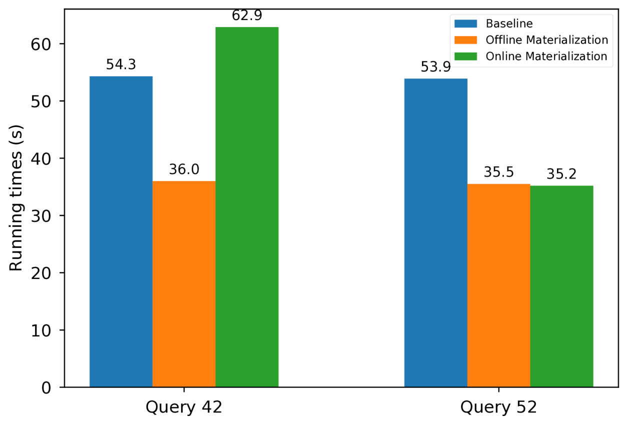


Figure 14: Query performances with offline and online materialization.

8.4 Case Studies

We presented our early results on TPC-DS benchmark in [28]. In the remaining section, we describe how SparkCruise operates under three different workload conditions. Materialized views generally have three phases: (1) view selection, and (2) view materialization, and (3) view matching and reuse. Traditionally, view selection and materialization are considered to be offline (i.e., interesting views are selected and materialized when the system is idle before the actual workload arrives), and view matching/reuse could be done either manually by having users modify their queries or automatically by the system. In SparkCruise, we only consider automatic view matching and reuse, since it is not trivial for users to make manual changes to their queries. However, we consider different permutations of view selection and materialization being offline or online.

8.4.1 Offline view selection, offline view materialization. For recurring workloads, we can run the view selection algorithm in an offline manner using the recurring signatures of query subexpressions. The view selection algorithm can then select the most promising recurring signatures for reuse and that are expected to remain consistent even with some changes in the workload. Furthermore, recurring workloads could have a slack between the arrival of data and workload start time, e.g., a log data is cooked and prepared daily and there are enough spare cycles during the day before the cooked data starts getting consumed by other queries. In such cases, we can use consider using offline cycles to materialize the views in advance. Every query that contains the view can benefit from reading the materialized view instead.

Figure 14 shows the the running time of TPC-DS queries 42 and 52 when both use a pre-materialized view to speed up their processing times, by 34% each compared to the baseline.

8.4.2 Offline view selection, online view materialization. Instead of materializing views offline in spare cycles, we can select views offline but materialize them in an online fashion, as part of query processing. The first query to hit the candidate view pays the overhead to materialize, but the subsequent queries can directly scan the materialized view. Figure 15 illustrates the query plans of the materialization subquery by TPC-DS query 42 and its subsequent reuse by Query 52. Figure 14 shows the the running time of Query 42 that materializes a view has increased, but interestingly, the overall workload time is still improves by 9.3%, since the reduction in running time of Query 52 is significant.

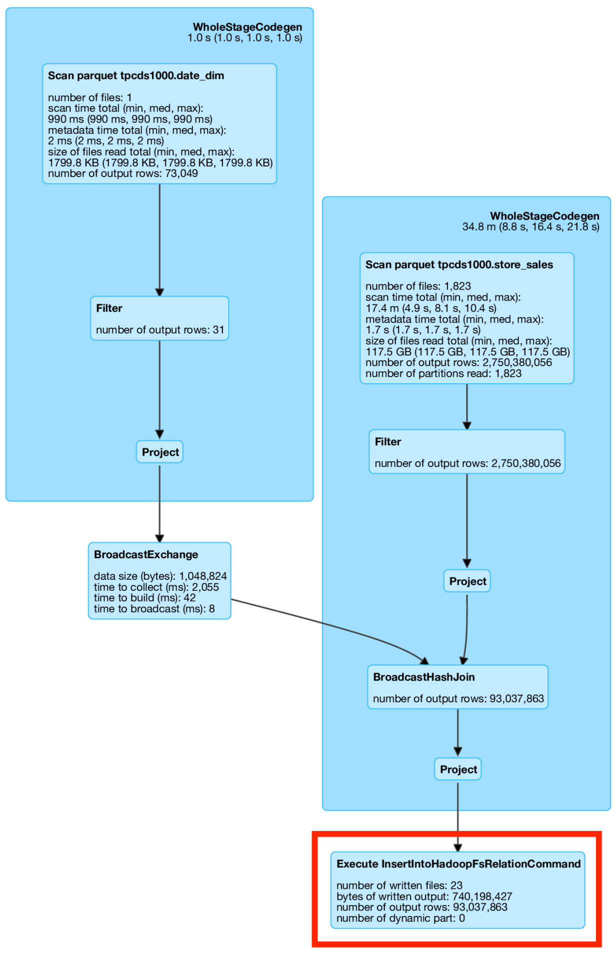
Online view materialization does not require offline cycles to materialize views. This makes a good fit for serverless systems where users spin up cluster on-demand, or for recurring workloads that are scheduled immediately after the input becomes available — typical scenarios in a production setting.

8.4.3 Online view selection, online view materialization. The scenarios considered so far target only the recurring workloads. Another class of workload that is growing is interactive/notebook applications that run as a session. In this scenario, we do not have prior runs of the complete workload. However, we have information from the previously executed queries within the same session. We utilize this information from these executed queries, to select views for materialization and reuse for future, yet unseen queries in the same session. The following changes are required in SparkCruise to handle this online view selection.

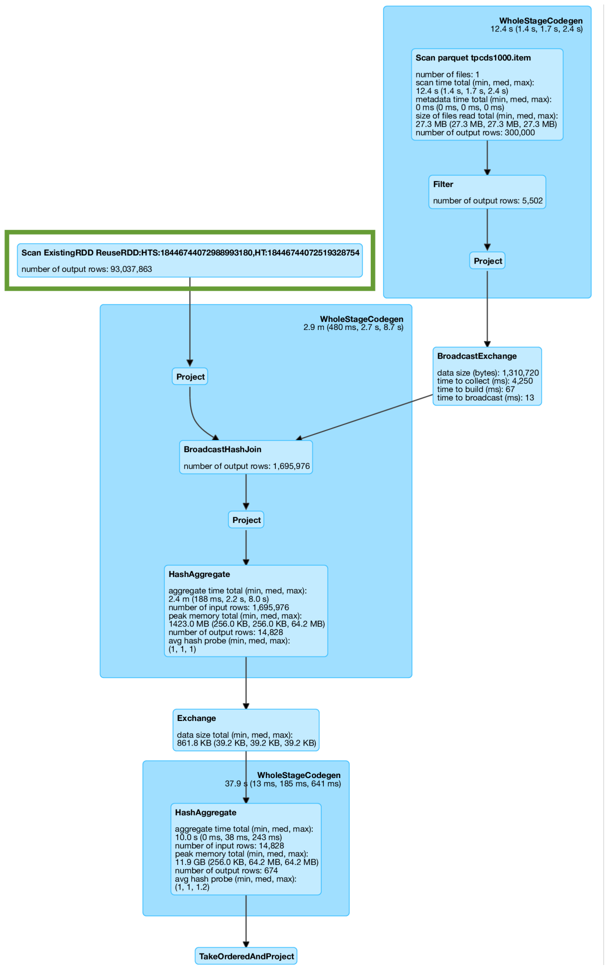
First, we incrementally update the workload table with telemetry from the last executed query. Then, we run view selection algorithm after a fixed number of queries to select views for future queries. The optimizer rules will apply the updated feedback to materialize and reuse views in an online fashion as before. We implemented a new online view selection algorithm for this scenario. The algorithm, like BigSubs [20], maintains a bijective graph between queries and the subexpressions seen so far. As new queries and subexpressions arrive they are added to the graph. The probability of a new subexpression being selected depends on its score (either frequency or utility) and interaction with existing materialized subexpressions (e.g., if it is contained within another subexpression). Similarly, the subexpressions can be evicted from the materialized set if the storage budget is exceeded.

We ran this new algorithm on TPC-DS, with queries executed serially one after another. For TPC-DS we observed that it was difficult to tune the algorithm because many subexpressions were repeated only few times, so any delay in selection would reduce the future opportunity for reuse in the same session. And if we aggressively select views, then the materialization cost exceed the reuse benefits. So we considered an alternative benchmark IDEBench [17], which is designed for interactive data exploration. We used the benchmark to generate one denormalized table of 500M rows and 40 queries. Figure 16 shows the ratio of running time with online view selection and baseline where we cache results from both common subexpressions and queries. Overall, the algorithm was able to reduce the running time by 9.6%. This could be further improved by predicting the future query properties, e.g. drill-down queries. Note that the online view selection algorithm could be used in conjunction with offline view selection to handle a mix or recurring and ad-hoc workloads.

In summary, we see that while different workload scenarios can have different amount of speedups, computation reuse is applicable quite broadly and can improve user experience and reduce costs.


WholeStageCodegen
1.0 s (1.0 s, 1.0 s, 1.0 s)
Scan parquet tpcds1000.date_dim
number of files: 1
scan time total (min, med, max):
990 ms (990 ms, 990 ms, 990 ms)
metadata time total (min, med, max):
2 ms (2 ms, 2 ms, 2 ms)
size of files read total (min, med, max):
1799.8 KB (1799.8 KB, 1799.8 KB, 1799.8 KB)
number of output rows: 73,049
Filter
number of output rows: 31
Project
BroadcastExchange
data size (bytes): 1,048,824
time to collect (ms): 2,055
time to build (ms): 42
time to broadcast (ms): 8
WholeStageCodegen
34.8 m (8.8 s, 16.4 s, 21.8 s)
Scan parquet tpcds1000.store_sales
number of files: 1,823
scan time total (min, med, max):
17.4 m (4.9 s, 8.1 s, 10.4 s)
metadata time total (min, med, max):
1.7 s (1.7 s, 1.7 s, 1.7 s)
size of files read total (min, med, max):
117.5 GB (117.5 GB, 117.5 GB, 117.5 GB)
number of output rows: 2,750,380,056
number of partitions read: 1,823
Filter
number of output rows: 2,750,380,056
Project
BroadcastHashJoin
number of output rows: 93,037,863
Project
Execute InsertIntoHadoopFsRelationCommand
number of written files: 23
bytes of written output: 740,198,427
number of output rows: 93,037,863
number of dynamic part: 0

(a) Materialization of join operator output by Query 42.


WholeStageCodegen
12.4 s (1.4 s, 1.7 s, 2.4 s)
Scan parquet tpcds1000.item
number of files: 1
scan time total (min, med, max):
12.4 s (1.4 s, 1.7 s, 2.4 s)
metadata time total (min, med, max):
0 ms (0 ms, 0 ms, 0 ms)
size of files read total (min, med, max):
27.3 MB (27.3 MB, 27.3 MB, 27.3 MB)
number of output rows: 300,000
Filter
number of output rows: 5,502
Project
Scan ExistingRDD ReuseRDD:HTS:18446744072988993180,HT:18446744072519328754
number of output rows: 93,037,863
WholeStageCodegen
2.9 m (480 ms, 2.7 s, 8.7 s)
Project
BroadcastExchange
data size (bytes): 1,310,720
time to collect (ms): 4,250
time to build (ms): 67
time to broadcast (ms): 13
BroadcastHashJoin
number of output rows: 1,695,976
Project
HashAggregate
aggregate time total (min, med, max):
2.4 m (188 ms, 2.2 s, 8.0 s)
number of input rows: 1,695,976
peak memory total (min, med, max):
1423.0 MB (256.0 KB, 256.0 KB, 64.2 MB)
number of output rows: 14,828
avg hash probe (min, med, max):
(1, 1, 1)
Exchange
data size total (min, med, max):
861.8 KB (39.2 KB, 39.2 KB, 39.2 KB)
WholeStageCodegen
37.9 s (13 ms, 185 ms, 641 ms)
HashAggregate
aggregate time total (min, med, max):
10.0 s (0 ms, 38 ms, 243 ms)
number of input rows: 14,828
peak memory total (min, med, max):
11.9 GB (256.0 KB, 64.2 MB, 64.2 MB)
number of output rows: 674
avg hash probe (min, med, max):
(1, 1, 1.2)
TakeOrderedAndProject

(b) Reuse of materialized join output by Query 52.

Figure 15: The figure shows query plans of a materialization sub-query (left) and the reuse of materialized output by a subsequent query (right).

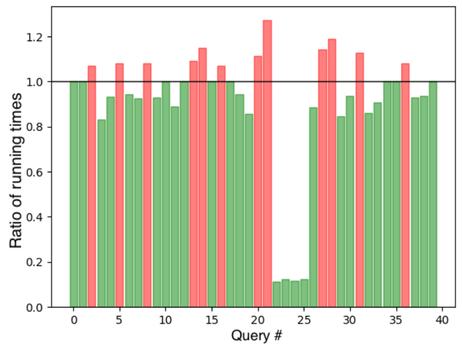


Figure 16: Online view selection and materialization on IDEBench.

9 CONCLUSIONS

In this paper, we presented the SparkCruise platform for adding workload-driven feedback loop to Spark query engine. Our key advancements are — (1) a scalable telemetry pipeline to collect plan information from production Spark deployments, (2) preprocessing stages to transform raw telemetry events into a tabular representation of workload that is more easy to analyze and derive insights from, (3) workload-based optimization algorithms to enable computation reuse, and (4) feedback mechanism to make Spark query optimizer workload-aware. We also took this system all the way to production in Azure HDInsight with the automatic computation reuse feature. Our experiences in production influenced the design choices for the system, such as preserving privacy of the users, having low overheads, and implementing all of the steps without modifying Spark code. The scale of Azure HDInsight enabled us to analyze production Spark SQL queries and contrast the differences with common benchmarks. There are many more optimization opportunities across workloads. Going ahead, we plan to build more workload-aware algorithms on top of SparkCruise platform. These workload-aware algorithms will enable us to instance optimize the Spark performance for a given workload and will reduce the total cost for our users.

REFERENCES

[1] 2017. *Cost Based Optimizer in Apache Spark 2.2.* Retrieved February 26, 2021 from https://databricks.com/blog/2017/08/31/cost-based-optimizer-in-apache-spark-2-2.html

[2] 2019. *Koalas: Easy Transition from pandas to Apache Spark.* Retrieved February 26, 2021 from https://databricks.com/blog/2019/04/24/koalas-easy-transition-from-pandas-to-apache-spark.html

[3] 2020. *Adaptive Query Execution: Speeding Up Spark SQL at Runtime.* Retrieved February 26, 2021 from https://databricks.com/blog/2020/05/29/adaptive-query-execution-speeding-up-spark-sql-at-runtime.html

[4] 2021. *Apache Spark.* Retrieved February 26, 2021 from https://spark.apache.org

[5] 2021. *Apache Spark on Amazon EMR.* Retrieved February 26, 2021 from https://aws.amazon.com/emr/features/spark/

[6] 2021. *Azure Databricks.* Retrieved February 26, 2021 from https://azure.microsoft.com/en-us/services/databricks

[7] 2021. *Azure Synapse Analytics.* Retrieved February 26, 2021 from https://azure.microsoft.com/en-us/services/synapse-analytics

[8] 2021. *CODAIT - Open Source.* Retrieved February 26, 2021 from https://www.ibm.com/opensource/centers/codait/

[9] 2021. *Databricks on AWS.* Retrieved February 26, 2021 from https://databricks.com/product/aws

[10] 2021. *Databricks on Google Cloud.* Retrieved February 26, 2021 from https://cloud.google.com/databricks

[11] 2021. *Scaling Apache Spark at Facebook.* Retrieved February 26, 2021 from https://www.slideshare.net/databricks/scaling-apache-spark-at-facebook

[12] Apache. 2021. *Spark Community.* Retrieved February 26, 2021 from https://spark.apache.org/community.html

[13] Michael Armbrust, Reynold S Xin, Cheng Lian, Yin Huai, Davies Liu, Joseph K Bradley, Xiangrui Meng, Tomer Kaftan, Michael J Franklin, Ali Ghodsi, et al. 2015. Spark sql: Relational data processing in spark. In *Proceedings of the 2015 ACM SIGMOD international conference on management of data.* 1383–1394.

[14] Ronnie Chaiken, Bob Jenkins, Per-Åke Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. 2008. Scope: easy and efficient parallel processing of massive data sets. *Proceedings of the VLDB Endowment* 1, 2 (2008), 1265–1276.

[15] Shaleen Deep, Anja Gruenheid, Kruthi Nagaraj, Hiro Naito, Jeff Naughton, and Stratis Viglas. 2020. DIAMetrics: benchmarking query engines at scale. *Proceedings of the VLDB Endowment* 13, 12 (2020), 3285–3298.

[16] Amol Deshpande, Zachary Ives, and Vijayshankar Raman. 2007. *Adaptive query processing.* Now Publishers Inc.

[17] Philipp Eichmann, Carsten Binnig, Tim Kraska, and Emanuel Zgraggen. 2018. IDEBench: A Benchmark for Interactive Data Exploration. arXiv:arXiv:1804.02593

[18] Apache Software Foundation. 2020. *Annual Report FY2020.* Retrieved February 26, 2021 from http://www.apache.org/foundation/docs/FY2020AnnualReport.pdf

[19] Apache Software Foundation. 2021. *Apache Committee Information.* Retrieved February 26, 2021 from https://projects.apache.org/committee.html?spark

[20] Alekh Jindal, Konstantinos Karanasos, Sriram Rao, and Hiren Patel. 2018. Selecting subexpressions to materialize at datacenter scale. *Proceedings of the VLDB Endowment* 11, 7 (2018), 800–812.

[21] Alekh Jindal, Hiren Patel, Abhishek Roy, Shi Qiao, Zhicheng Yin, Rathijit Sen, and Subru Krishnan. 2019. Peregrine: Workload optimization for cloud query engines. In *Proceedings of the ACM Symposium on Cloud Computing.* 416–427.

[22] Alekh Jindal, Shi Qiao, Hiren Patel, Zhicheng Yin, Jieming Di, Malay Bag, Marc Friedman, Yifung Lin, Konstantinos Karanasos, and Sriram Rao. 2018. Computation reuse in analytics job service at microsoft. In *Proceedings of the 2018 International Conference on Management of Data.* 191–203.

[23] Tim Kraska, Mohammad Alizadeh, Alex Beutel, H Chi, Ani Kristo, Guillaume Leclerc, Samuel Madden, Hongzi Mao, and Vikram Nathan. 2019. Sagedb: A learned database system. In *CIDR.*

[24] LinkedIn. 2021. *Analytics Platforms.* Retrieved February 26, 2021 from https://engineering.linkedin.com/teams/data/analytics-platform-apps/analytics-platforms

[25] Jayant Madhavan, Philip A. Bernstein, and Erhard Rahm. 2001. Generic Schema Matching with Cupid. In *Proceedings of the 27th International Conference on Very Large Data Bases (VLDB '01).* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 49–58.

[26] Microsoft. 2021. *Azure HDInsight.* Retrieved February 26, 2021 from https://azure.microsoft.com/en-us/services/hdinsight/

[27] Parimarjan Negi, Matteo Interlandi, Ryan Marcus, Mohammad Alizadeh, Tim Kraska, Marc Friedman, and Alekh Jindal. 2021. Steering Query Optimizers: A Practical Take on Big Data Workloads. In *Proceedings of the 2021 ACM SIGMOD International Conference on Management of Data.*

[28] Abhishek Roy, Alekh Jindal, Hiren Patel, Ashit Gosalia, Subru Krishnan, and Carlo Curino. 2019. Sparkcruise: Handsfree computation reuse in spark. *Proceedings of the VLDB Endowment* 12, 12 (2019), 1850–1853.

[29] Tarique Siddiqui, Alekh Jindal, Shi Qiao, Hiren Patel, and Wangchao Le. 2020. Cost models for big data query processing: Learning, retrofitting, and our findings. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data.* 99–113.

[30] Chenggang Wu, Alekh Jindal, Saeed Amizadeh, Hiren Patel, Wangchao Le, Shi Qiao, and Sriram Rao. 2018. Towards a learning optimizer for shared clouds. *Proceedings of the VLDB Endowment* 12, 3 (2018), 210–222.

[31] Reynold S Xin, Josh Rosen, Matei Zaharia, Michael J Franklin, Scott Shenker, and Ion Stoica. 2013. Shark: SQL and rich analytics at scale. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of data.* 13–24.

[32] Haitao Yuan, Guoliang Li, Ling Feng, Ji Sun, and Yue Han. 2020. Automatic view generation with deep learning and reinforcement learning. In *2020 IEEE 36th International Conference on Data Engineering (ICDE).* IEEE, 1501–1512.

[33] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. 2010. Spark: Cluster computing with working sets. *HotCloud* 10, 10-10 (2010), 95.