SparkCruise: Handsfree Computation Reuse in Spark

Abhishek Roy, Alekh Jindal, Hiren Patel, Ashit Gosalia, Subru Krishnan, Carlo Curino

Microsoft

{abhishek.roy,alekh.jindal,hirenp,ashit.gosalia,subru,carlo.curino}@microsoft.com

ABSTRACT

Interactive data analytics is often inundated with common computations across multiple queries. These redundancies result in poor query performance and higher overall cost for the interactive query sessions. Obviously, reusing these common computations could lead to cost savings. However, it is difficult for the users to manually detect and reuse the common computations in their fast moving interactive sessions. In the paper, we propose to demonstrate SparkCruise, a computation reuse system that automatically selects the most useful common computations to materialize based on the past query workload. SparkCruise materializes these computations as part of query processing, so the users can continue with their query processing just as before and computation reuse is automatically applied in the background — all without any modifications to the Spark code. We will invite the audience to play with several scenarios, such as workload redundancy insights and pay-as-you-go materialization, highlighting the utility of SparkCruise.

PVLDB Reference Format:
Abhishek Roy, Alekh Jindal, Hiren Patel, Ashit Gosalia, Subru Krishnan, Carlo Curino. SparkCruise: Handsfree Computation Reuse in Spark. *PVLDB*, 12(12): 1850-1853, 2019.
DOI: https://doi.org/10.14778/3352063.3352082

1. INTRODUCTION

Cluster-as-a-service is emerging as a popular choice for interactive data analytics in the cloud. It allows an analyst to quickly spin up a cluster for her interactive session and then tear it down once she is done. Unfortunately, such analytics often incurs redundancies in the query workload, i.e., parts of the computations are unknowingly duplicated across multiple queries. Our analysis from production Azure HDInsight [1] workloads at Microsoft reveal that more than 50% of the queries have overlaps, referred to as *common subexpressions* henceforth, with one or more other queries in the same session. Likewise, an analysis over the publicly available TPC-DS benchmark shows that more than 90% of the queries have common subexpressions, even after excluding the scan operators. Materialized views is a common technique in database literature for optimizing multiple queries by storing pre-computations (materialized views) that could speed up query execution [4]. However, creating, reusing, and maintaining materialized views has been a major pain for database users. This is even more challenging in cluster services where reusing computations within a fast moving session is very hard. At the same time, reducing the total operational costs of these interactive sessions is crucial.

In our recent work, we described automatic computation reuse for a job service environment at Microsoft [6]. In this demonstration, we extend that to cluster-as-a-service environments and describe a handsfree computation reuse infrastructure, coined SparkCruise, for Spark processing engine in Azure HDInsight clusters. The goal is to create smarter cluster instances that can self-tune the data processing systems for the customers. Our key ideas include: (i) instrumenting the Spark application log with subexpression identifiers, called *signatures*, that could be later used to identify common subexpressions (ii) analyzing the Spark query plans to collect the common subexpressions and using runtime metrics to reason about their utility for reuse, (iii) generating query *annotations* and providing them as feedback for future Spark queries, and, (iv) additional Spark optimizer rules for materializing and reusing common subexpressions in an online manner. In the rest of the paper, we first present a detailed description of SparkCruise (Section 2), then we show an evaluation of SparkCruise over the TPC-DS benchmark (Section 3), and finally we describe several demonstration scenarios (Section 4).

2. SparkCruise OVERVIEW

We consider a recurring query workload [6], denoted as $W = \{Q_1, Q_2, \ldots, Q_N\}$ where Q_i is a SparkSQL query. We consider materializing the logical subexpressions of the queries in W. Let $S(Q_i)$ denote the set of subexpressions present in query Q_i and $S = \{S_1, S_2, \ldots, S_N\}$ be the set of all subexpressions in W. Then, the common subexpressions between any two queries Q_i and Q_j are the set of subexpressions in $S(Q_i) \cap S(Q_j)$. For example, queries Q_1 and Q_2 in Figure 1 share the subexpression $\sigma_{Price>100}(Order)$.

The goal of SparkCruise is to detect the most useful common subexpressions in W to materialize, automatically materialize them as part of query processing, and subsequently reuse them in future queries. Figure 2 shows the system architecture. It includes components for log ingestion, analysis, and feedback deployment. SparkCruise is

Proceedings of the VLDB Endowment, Vol. 12, No. 12
ISSN 2150-8097.
DOI: https://doi.org/10.14778/3352063.3352082

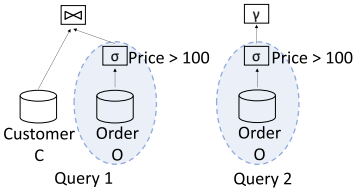


Figure 1: Common subexpression between two queries.

plug-and-play with Apache Spark using just the configurations, without requiring any code changes in Spark. Below we describe the key features of SparkCruise.

2.1 Application Log Instrumentation

We implemented an event listener for SparkSQL that logs the query plans at the end of query execution. We further annotated the query plans with identifiers, called *signatures* for each subexpression. We calculate two signatures, a recurring and a strict signature for each subexpression. The recurring signature ignores the time varying information such as literal values and dataset version numbers, and is used to find recurring subexpressions in the workload. The strict signature includes all plan information to precisely identify a subexpression instance. We provide the actual signature implementation as an external library and could provide different implementations to identify subexpressions differently. Signatures help to identify common subexpressions across a query workload, in contrast to the *semantic hash* that is already present in SparkSQL and is used for matching equivalent subexpressions while planning a given query. Our custom event listener collects the annotated query plans logs (in JSON format) in the application logs, which get persisted on the HDInsight cluster. Once we have the application logs with the annotated query plans, we analyze them as described below.

2.2 Workload Analysis

SparkCruise periodically analyzes the query workload from the application logs. We parse the annotated query plans, namely the parsed, the optimized, and the physical query plans, in the application log and iterate over the subexpressions of the optimized (logical) query plans. Considering the logical subexpressions allows us to cover common expressions more generally, however, we link the logical subexpressions with their corresponding runtime metrics from the physical query plans. Note that attributing runtime metrics to logical expressions could be hard in many cases, since the logical and the physical plans may not always correspond. Still, our best effort linking was able to link more than 90% of plans in our workloads. We collect the subexpressions, along with the runtime metrics, into a subexpressions table. Each row in this table represents a subexpression and all the compile-time and runtime features associated with it. We also preserve query metadata, such as application identifier, with each subexpression. The resulting flat, denormalized subexpressions table turns out to be very useful to run the subexpression selection algorithms. We describe these below.

2.3 Subexpression Selection

We now select the subexpressions to materialize from the subexpressions table generated above. The goal is to select

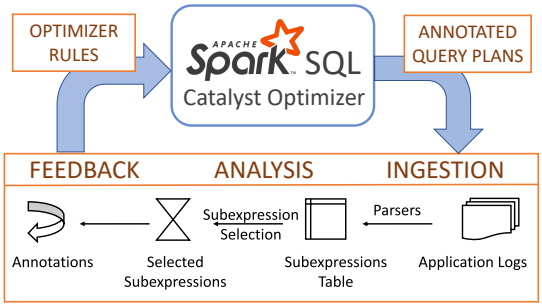


Figure 2: SparkCruise Architecture.

the smallest set of common subexpressions that offers the most utility for reuse. The subexpression selection problem as well as the more general view selection problem has been widely studied in the literature [5, 7]. However, since Spark sessions are interactive in nature, and hence short-running compared to batch processing, we want to provide heuristics that are effective and yet run very fast, in order of a few seconds. These heuristics allow users to make trade offs such as storage space and runtime savings. Larger workloads can of course use more sophisticated subexpression selection algorithms from the literature, e.g., BigSubs [5]. We discuss two of our heuristics below.

The first heuristic uses the repeat frequency to select the *top-K* frequent subexpressions. It also restrict the number of selected subexpressions from each query. This limits the overhead of materializing subexpressions in any given query. The second heuristic weighs the frequency of subexpressions by their height in the query plan. As a result, this heuristic selects common subexpressions that are closer to root level.

2.4 Feedback Loop

By default, Spark runs the SparkSQL queries identically every single time they appear on the same inputs. With SparkCruise, we add a feedback loop to learn on how things went in the past and improve the query plans in the future. Specifically, we identify the common subexpressions to materialize, from our analysis over the past workload, and supply it as feedback for future queries to the query optimizer. We provide this feedback in the form of *query annotations*, where each annotation has the recurring signature of the subexpression and the corresponding materialize and reuse command. These annotations can be served from a local file or from a web-service, as also described in the CloudViews system [6]. Since we are working in a cluster environment in HDInsight, we use the local file system to serve feedback. SparkCruise provides additional optimizer rules for SparkSQL to load the query annotations and apply the feedback during query optimization. We describe this mechanism below.

2.5 Optimizer Extensions

SparkCruise provides optimizer extensions for SparkSQL to materialize and reuse common subexpressions in an online fashion. We want to reuse the common subexpressions within a Spark session and so the first queries in the session that hit each of the selected subexpressions materialize the result of the subexpression. Subsequent queries in the session can directly read from the materialized subexpressions without computing them again. To perform these operations, we use the Spark extensions points added in Spark-18127 [3]. These extensions allow SparkCruise to

complement the Spark optimizer with two additional rules for modifying the logical query plans, as described below.

The first rule, *Online Materialization*, materializes the output of a subexpression when: (i) the feedback contains an annotation with the same recurring signature as that of the subexpression, and (ii) the subexpression has not been materialized already. We also synchronize multiple queries hitting the same rule such that only one of the queries materializes a given subexpression. The materialize operation triggers a smaller SparkSQL query for the subexpression, followed by reusing the materialized subexpression for the remainder of the same query. Once a subexpression is materialized, we also preserve its strict signature for matching future reuse. The second rule, *Computation Reuse*, replaces a subexpression with the scan of the materialized subexpression when: (i) the recurring signature of the subexpression is present in the feedback, and (ii) the strict signature of the subexpression matches the strict signature of the materialized subexpression. Note that the optimizer always tries to apply the materialize rule before applying the reuse rule. However, both rules can be applied multiple times in the same query plan.

2.6 Configurations

SparkCruise provides a few configurations, including setting the frequency to analyze the application log, selecting the subexpressions to materialize for a given storage budget, policies for evicting the materialized subexpressions, etc. The materialized subexpressions could be stored either on the local HDFS in the HDInsight cluster or on a global blob store like the Windows Azure Storage Blob (WASB). Azure HDInsight cluster provides the mechanics to use either of them. The local HDFS on the cluster offers faster read and write times by virtue of using local SSDs. However, the lifetime of the local HDFS is same as that of the cluster and so materialized subexpressions are not persisted when the cluster is restarted. Local HDFS also has a limited storage capacity, depending on the machine configurations, while WASB offers a permanent and practically unlimited storage capacity. In our future work, we plan to optimally place the materialized files in the different storage levels, including main-memory, local HDFS, and WASB. Currently, we support materialization in CSV and the Parquet formats.

3. EVALUATION

We now present an evaluation of SparkCruise on TPC-DS benchmark using an Azure HDInsight Spark cluster with the default configuration. The cluster was equipped with four data nodes of `D13v2` instance type. We generated the TPC-DS dataset for a scale factor of 10, resulting in a total size of tables of 6.8GB after Parquet compression. We executed the SparkSQL implementation of 95 queries from the TPC-DS benchmark [2], using the command line SparkSQL interface. We measured the end to end wall clock times of each query and report the average over three runs. First, we ran all queries, one after the other, without feedback to measure the baseline performance. Then, we ran the workload analysis and provided the feedback using SparkCruise.

Our analysis identified 398 common subexpressions in the entire workload, appearing with an average frequency of 5.5 times. Scan, of course, are common in general, but our analysis also showed 103 filter, 58 join, and 19 aggregate subexpressions to be in common. Using the frequency-based heuristic described in Section 2.3, we selected 17 subexpressions to materialize. Note that this is a very conservative set of subexpressions to materialize. Figure 3 shows the ratio of the runtime with SparkCruise to that with the baseline. We observe that the running time of some queries (in red bars) has increased, because it is the first query that hit a common subexpression and has to pay the materialization cost. However, once common subexpressions are materialized, the running times of the majority of the queries (in green bars) improve as they can read directly from the materialized file. Overall, we see the total running time reducing by approximately 30%. These improvements are due to the fact that the 17 selected subexpressions were materialized by only 11 queries, while they were reused by 89 queries.

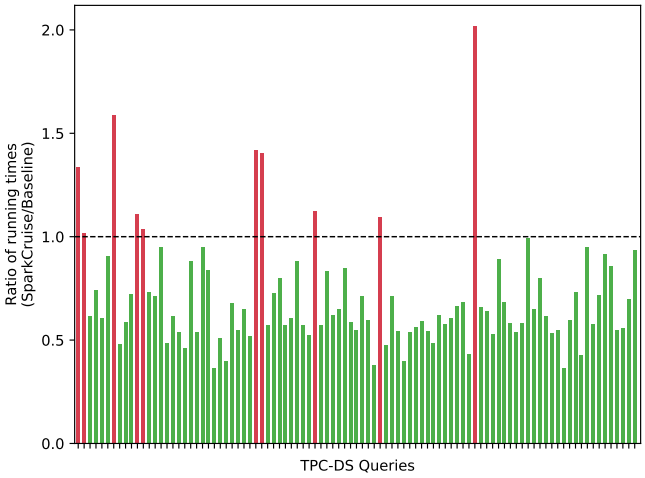


Figure 3: Running time ratio on TPC-DS workload.

4. DEMONSTRATION

The goal of this demonstration is to provide actionable insights from a SparkSQL workload and to explore the performance of our system under recurring and changing workloads. We will run the demonstration of our system on Azure HDInsight Spark cluster. The system will be pre-loaded with TPC-DS dataset. The audience will be provided with a query console, the ability to tune the system parameters, and access the performance metrics. We will invite the audience to play with SparkCruise for the following four scenarios. (i) analyzing common subexpressions from a SparkSQL application log, (ii) seeing the online materialization and reuse in action, (iii) tweaking the subexpression selection heuristics for trading between cost and performance, and (iv) observing how the system behaves with changing workloads. We have created a graphical user interface, as illustrated in Figure 4(a), to interact in each of the scenarios. We describe these scenarios in detail below.

4.1 Workload Redundancy Insights

A major challenge in big data analytics is that neither the data analyst nor the system administrator understand the query workload very well. Therefore, our first scenario is to help the audience understand the redundancies (the common subexpressions) in a query workload. The audience can analyze the pre-generated TPC-DS application log and see the overall number of common subexpressions and their

reuse utility. They can further dig into per-operator common subexpressions to get a sense of which operations are most common among different queries in the workload. The audience is free to issue different ad-hoc queries on the TPC-DS dataset, using the SparkSQL console, and analyze the correspondingly generated application log. The workload analysis takes less than 5 seconds for all TPC-DS queries, so we expect to provide an interactive experience. Workload redundancy insights helps a user in understanding the cost and benefits of enabling SparkCruise on her cluster.

4.2 Pay-as-you-go View Materialization

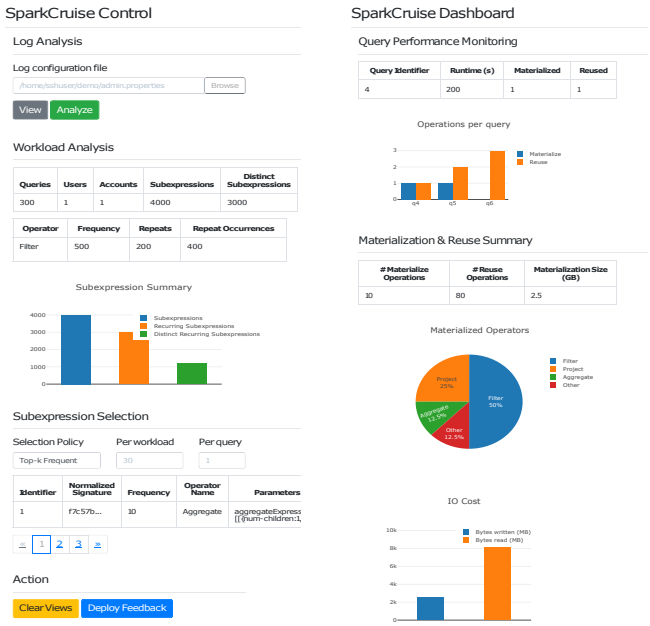
Data analysts want to get started as soon as new data arrives and not wait for long data preparation steps. As a result, there is no offline phase available to materialize common subexpressions and speed up a query workload. Therefore, in this scenario, we invite the audience to start issuing a set of queries, that have been already analyzed from the past application logs, and observe how (i) common subexpressions selected by our analysis are materialized as part of the query processing, and (ii) the materialized common subexpression automatically gets reused. The audience can change the arrival times of the queries or even reorder the queries, and the view materialization happens in a pay-as-you-go manner when the first query hits a common subexpression. The audience can see the overheads of materializing the subexpressions. Note that a query can both materialize and reuse subexpressions, and therefore the cost of a query can be lower even if it materializes a selected subexpressions. Throughout this scenario, the workload analysis is triggered at regular intervals in the background, while the demonstration attendees keep querying their data without any offline phase to create materialized views. Selected subexpressions from the analysis are fed back to optimizer and materialized as part of query processing.

4.3 Trading Materialization Cost and Utility

The goal of SparkCruise is to provide a handsfree experience in reusing common subexpressions. Still, for expert users, we allow to trade between materialization costs and utility via a set of subexpression selection policies. Example policies include considering subexpressions appearing that appear at least n times, or subexpressions with at least t runtime, or consider at most k subexpressions per query, or consider a total storage budget of B. These individual polices provide the tradeoffs between storage budget, performance improvements, cost per subexpression, and utility per query. The audience can select one of the provided subexpression selection policies and tune its parameters, e.g., the n, t, k, and B above, using the console in Figure 4(a). They can also combine multiple policies and discover combinations that work the best for different sets of queries. As before, the audience can use our pre-generated TPC-DS application log or issue new ad-hoc queries using the SparkSQL console.

4.4 Handling Workload Changes

SparkCruise relies on the past workload being a strong indicator of the future. This works very well for enterprise workloads that are predictable and recurring in nature, i.e., same queries arriving at a fixed frequency over changing the data. However, each instance of a query comes with some minor changes. One common pattern is to update the literal values in the queries, e.g., the user can set the



(a) Workload Analysis Console. (b) SparkCruise Dashboard.

Figure 4: SparkCruise Web Interface

calendar day to increase by one on each day. In this case, any two instances of the workload will have their selection expression has changed. SparkCruise can successfully handle this scenario, since we use the recurring signatures for analysis and strict signatures for materialization and reuse operations. In the case of more dramatic changes in the workload, we do not incur any wasted cost for materialization, since new queries do not contain the previously selected common subexpressions. SparkCruise adapts the selected subexpressions to these new changes in the workload simply by re-running the analysis and deploying the feedback periodically. We will invite the audience to change one or more queries in the workload, either in the literal values or in the query structure itself. The audience will be able to see that SparkCruise handles the former case perfectly while in the latter case it does not incur any materialization overhead until the next analysis/feedback is available. The audience can monitor the performance under changing workloads on SparkCruise dashboard in Figure 4(b).

5. REFERENCES

[1] Azure HDinsight. `https://azure.microsoft.com/en-us/services/hdinsight/`. Accessed: 2019-03-06.

[2] Spark SQL performance tests. `https://github.com/databricks/spark-sql-perf`. Accessed: 2019-03-06.

[3] S. Agarwal. [SPARK-18127] Add hooks and extension points to spark. `https://issues.apache.org/jira/browse/SPARK-18127`, April 2017. Accessed: 2019-03-06.

[4] R. Chirkova, J. Yang, et al. Materialized views. *Foundations and Trends® in Databases*, 4(4):295–405, 2012.

[5] A. Jindal, K. Karanasos, S. Rao, and H. Patel. Selecting subexpressions to materialize at datacenter scale. *PVLDB*, 11(7):800–812, 2018.

[6] A. Jindal, S. Qiao, et al. Computation reuse in analytics job service at Microsoft. *SIGMOD*, 2018.

[7] P. Michiardi, D. Carra, and S. Migliorini. In-memory caching for multi-query optimization of data-intensive scalable computing workloads. *DARLI-AP Workshop*, 2019.