Optimizing Interactive Development of Data-Intensive Applications

Matteo Interlandi Sai Deep Tetali [*] Muhammad Ali Gulzar Joseph Noor
Tyson Condie Miryung Kim Todd Millstein
University of California, Los Angeles
{miterlandi, saideep, gulzar, jnoor, tcondie, miryung, todd}@cs.ucla.edu

Abstract

Modern Data-Intensive Scalable Computing (DISC) systems are designed to process data through batch jobs that execute programs (e.g., queries) compiled from a high-level language. These programs are often developed interactively by posing ad-hoc queries over the base data until a desired result is generated. We observe that there can be significant overlap in the structure of these queries used to derive the final program. Yet, each successive execution of a slightly modified query is performed anew, which can significantly increase the development cycle. VEGA is an Apache Spark framework that we have implemented for optimizing a series of similar Spark programs, likely originating from a development or exploratory data analysis session. Spark developers (e.g., data scientists) can leverage VEGA to significantly reduce the amount of time it takes to re-execute a modified Spark program, reducing the overall time to market for their Big Data applications.

Categories and Subject Descriptors H.2.4 [*Information Systems*]: Database Management—query processing, parallel databases

General Terms Languages, Performance, Theory

Keywords Query Rewriting, Incremental Evaluation, Spark, Interactive Development, Big Data

1. Introduction

Data scientists report spending the majority of their time writing code to ingest data from several sources, transforming it into a common format, cleaning erroneous entries, and performing exploratory data analysis to understand its structure [22]. These tasks span the development cycle of a Big Data application. For instance, data cleaning and exploratory data analysis are typically performed in an *iterative process*: programmers start with an initial query and iteratively improve it until the output is in a desired form.

Despite this common pattern, existing Data-Intensive Scalable Computing (DISC) systems, such as Apache Hadoop [3] and Apache Spark [4], do not optimize for these scenarios. Rather, they run each query refinement anew, ignoring the work done in previous executions. Due to the immense scale of today's datasets and the aforementioned steps involved, developing Big Data applications is very time consuming; it is not uncommon for data scientists to wait hours, only to find that they should have filtered some unforeseen outliers. A common fallback approach is to develop against a sample of the data. However, this approach is incomplete in that it does not guard against outliers not in the sample.

Our goal is to support interactive development of data-intensive applications by optimizing the execution of query refinements. There have been several prior works on optimizing data-intensive applications, but they do not meet this need. Some works can provide large speedups in the face of changes to the input data, for example via a form of *incremental computation* [25, 26, 29] or targeted optimizations for *recurring workloads* [23, 28] and *iterative* (*recursive*) *queries* [9, 26, 31]. These approaches all assume that the query itself is unchanged. Other systems provide the ability to cache and reuse the results of sub-computations [15, 24]. In an interactive development setting, these systems would allow unchanged portions of a query to reuse old results. However, any parts of a query that are downstream of a code change must still be executed from scratch, thereby limiting the ability to obtain interactive speeds.

In this paper we introduce VEGA: an Apache Spark framework that automatically optimizes a series of similar Spark programs, likely originating from a development or exploratory data analysis session. VEGA automatically reuses materialized intermediate results from the previous

[*] Now at Google, Inc.

SoCC '16, October 05-07, 2016, Santa Clara, CA, USA.
© 2016 ACM. ISBN 978-1-4503-4525-5/16/10. . . $15.00.
DOI: http://dx.doi.org/10.1145/2987550.2987565

run when executing the new version of a program. As a starting point, we can reuse the results from the latest materialization point before any code modification, as prior cache-based systems would do [15, 24]. VEGA significantly improves upon this baseline by automatically *rewriting* the dataflow to push the code modifications as late as possible, thereby allowing the execution to start from a later materialization point. This optimization is driven by an analysis that determines when and how two successive operations can be reordered without changing program semantics.

In addition to the rewriting optimization, VEGA employs a complementary technique that adapts the prior work on incremental computation mentioned above (i.e., [25, 26, 29]) to our setting. Specifically, VEGA can perform an incremental computation rather than ordinary re-execution of the operations downstream of the modified portion of the program, thereby computing only data changes (*deltas*) relative to the previous execution. We detail how VEGA determines when such incremental computation is more profitable than ordinary computation.

We have implemented VEGA both at the Spark SQL level (referred to as VEGA SQL) as well as at the RDD transformation level (VEGA RDD), in order to optimize programs written directly in Spark. Thanks to the high-level semantics of Spark SQL, query rewriting performed by VEGA SQL is completely transparent to the user, i.e., no additional information is required from the programmer. VEGA RDD instead trades off transparency for additional optimizations: VEGA RDD comes with a specifically tailored API enabling rewrites that leverage the lower-level physical plan information provided by the RDD representation. VEGA RDD also supports the complementary incremental computation optimization.

Experimental evaluations show that VEGA is able to achieve up to three orders-of-magnitude performance gains by reusing work from prior Spark program executions for several real-world scenarios. Figure 1 previews the performance of VEGA SQL compared to normal Spark SQL for re-executing a modified query that measures how many links in the Common Crawl dataset [2] point to a certain domain; the modification refines the query by returning only links that point to Wikipedia pages. In the case of Spark SQL, the modified program is executed from scratch, whereas VEGA SQL is able to rewrite the modification to operate over the output of the previous execution. As a result, the response time of the re-executed query is significantly lower than native Spark SQL. A more complete description of this experiment is given in Section 5.

Contributions. To the best of our knowledge, VEGA is the first DISC systems approach to explicitly support optimizations for iterative program (query) development. The paper makes the following contributions over the state of the art:

- Two techniques to support interactive development of data-intensive applications: a form of query rewriting that pushes code modifications to later dataflow stages in order to avoid re-computing upstream operations; and an adaptation of incremental computation to avoid a full re-computation of downstream operations.
- A library that implements these techniques to speed up the evaluation of updates to both Spark SQL queries and Spark programs.
- A set of experiments over real-world use cases showing the effectiveness of the approach.

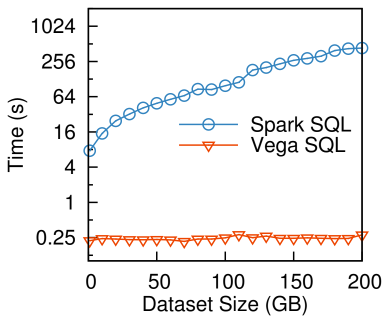


Figure 1. VEGA SQL compared against the Spark SQL base case, in which the re-execution of the changed program takes place without reusing previous results.

Organization. The paper is organized as follows: Section 2 briefly introduces Spark, Spark SQL, and describes our approach for optimizing the execution of modified programs. Section 3 defines VEGA's optimization techniques, and Section 4 describes the VEGA SQL and VEGA RDD implementations on top of Spark. The experimental evaluation is detailed in Section 5. Lastly, Section 6 covers related work and Section 7 concludes the paper.

2. Overview

This section provides a brief background on Apache Spark. It then describes an example Spark program, which we use to informally overview our two techniques—*plan rewriting* and *incremental computation*—for optimizing the execution of modified code.

2.1 Apache Spark

Spark is a platform for executing data-parallel computations on Resilient Distributed Datasets (RDDs) [33] that reference data in distributed storage e.g., HDFS. The RDD abstraction provides *transformations* (e.g., map, reduceByKey, filter, groupBy, join, etc.) and *actions* (e.g., count, collect) that operate on the reference partitioned data. A typical Spark program executes a series of transformations ending with an action that returns a result value (e.g., the record count of an RDD, a collected list of records referenced by the RDD) to the *driver program*.

RDDs are *immutable* and RDD transformations are *coarse-grained*: i.e., applied in bulk over all the items in the target RDD. In the driver program, Spark *lazily* evaluates

RDD transformations by returning a new RDD reference specific to that transformation operation; essentially building a query plan. Any action executed by the driver triggers the execution of the query plan referenced by the action. To execute a query plan, Spark compiles the transformations into a dataflow (or DAG) of *stages*. Spark groups transformations that can be pipelined (i.e., results are passed one-to-one between transformations) into a single stage. A *shuffle step* is used to re-partition the data between stages. The final stage is responsible for executing the action and returning the result to the driver program. The stage DAG represents the physical plan, which is passed to the Spark scheduler for execution. The Spark scheduler is responsible for evaluating each stage: a stage is executed before downstream dependent stages are scheduled i.e., Spark batch executes the stage DAG. To execute a "runnable" stage, the Spark scheduler will launch *tasks* that perform the operations of the stage on input data partitions. Intermediate stage inputs and outputs are materialized in the Spark Block Manager.

Spark SQL. Originally developed as Shark [32], Spark SQL enables queries over structured data on Spark, using the familiar declarative SQL language or DataFrame API. Spark SQL comes with an optimizer framework called Catalyst, which represents expressions (e.g., selection predicates, attribute projections, join conditions) as *trees* and supports rules that can manipulate them. The Spark SQL compiler and optimizer leverage Catalyst for query analysis, logical plan optimization, and physical plan generation (i.e., to a Spark program).

2.2 Running Example

Our running example leverages a dataset made available by the NYC Open Data Project. Calls to the non-emergency service center are monitored, and related metadata is saved into a database. An excerpt of this CALLS database, containing data ranging from 2010 to 2015, is publicly available [1].

```
1  case class Calls(id:String,hour:Int,agency:String,...)
2  format = new SimpleDateFormat("M/d/y h:m:s a")
3  input = sc.textFile("hdfs://...")
4  calls = input.map(_.split(",")).map(r =>
5      Calls(r(0),format.parse(r(1)).getHours,r(2),...)
6  calls.registerTempTable("calls")
7  hist = sqlContext.sql("
8  SELECT agency, count(*)
9    FROM calls
10   JOIN (
11     SELECT hour
12       FROM calls
13       GROUP BY hour
14       HAVING count(*) > 100000
15   ) counts
16   ON calls.hour = counts.hour
17   GROUP BY agency")
18 hist.show()
```

Figure 2. Spark SQL program generating calls distribution per agency during busy hours. The inner query is used to detect the busy hours, i.e., when the number of incoming calls exceed 100k.

Assume that a service manager is interested in knowing *the agencies that received the most calls during busy hours*, where an hour is considered busy if more than 100*k* calls were received in total. The Spark SQL program in Figure 2 can be used to answer this query. The schema of the CALLS dataset is defined in line 1. Line 3 loads the content of the specified path from HDFS into the input RDD. The data is in CSV format; lines 4-5 parse and load the data into the calls DataFrame. Lines 8-17 contains a Spark SQL query that generates a histogram of calls received by agencies during busy hours. The inner query (lines 11-14) identifies the busy hours. The outer query joins the inner query result with the calls DataFrame to produce call records during busy hours. The group-by operation generates the final distribution containing the number of calls (during busy hours) received by each agency. The evaluation of the query defining hist is triggered by the show action (line 18) which prints the result. The same Spark SQL query can be represented by the logical query plan depicted in Figure 3 (executed bottom-up).

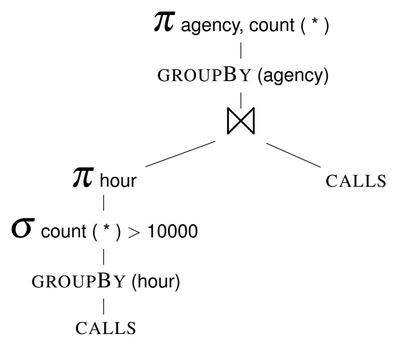


Figure 3. Logical plan for the program of Figure 2.

It turns out that this dataset has a subtle bug: calls that were not assigned a creation date are given a default hour of zero, indicating that the call occurred during the midnight hour. The discovery of this bug could motivate the following revision to the inner query, with the goal of removing skewed (midnight) entries:

```
SELECT hour
FROM calls
WHERE hour <> 0
GROUP BY hour
HAVING count(*) > 100000
```

Resubmitting the overall program, with the revised inner sub-query, will execute from scratch in Spark SQL. Our goal in VEGA is to do better by leveraging work done by the previous execution. Next, we introduce two techniques that VEGA uses for this purpose.

2.3 Query Plan Rewriting

The revised logical plan for the inner query (left branch in Figure 3) will be:

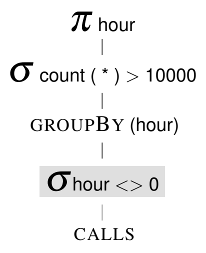


Figure 4. Modification to the logical plan of Figure 3.

Catalyst—the Spark SQL query optimizer and planner—will plan the added selection predicate close to the source calls dataset (as shown in Figure 4) to exploit early pruning. In contrast, the VEGA query rewrite technique will try to "push" the introduced selection predicate as late as possible in the query plan, allowing maximal reuse of materialized intermediate results from the previous execution. In the above example, VEGA recognizes—by analyzing the logical query plan expression in Catalyst—that the added where condition *commutes* with the inner group-by operation, since that operation does not modify the hour field of any record. Similarly, the new where condition also commutes with the subsequent count, projection, and join operations. However, the new where condition cannot be pushed past the GROUPBY operation because it groups over a different key (i.e., agency).

Therefore, assuming that the join result from the previous execution was materialized, the following logical plan will produce the same result as re-executing the modified query.

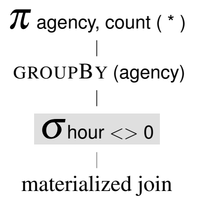


Figure 5. Optimized logical plan.

The performance gains from this rewrite are significant. As we will show in Section 5, rewritten queries can deliver up to three orders-of-magnitude performance improvement w.r.t. the base case of re-running from scratch.

2.4 Beyond Logical Optimizations

As we will describe in the next section, the above rewrite technique works at both the Spark SQL level (via the logical query plan) and at the physical RDD level. VEGA further leverages incremental evaluation at the RDD level to speed up Spark program re-execution. Our approach leverages prior work on handling *incremental data changes* for efficient support of view maintenance [6, 16] and iterative queries [8, 9, 11, 26]. Specifically, we treat the output of a new or modified RDD as a change to the input data for the downstream operators, relative to that of the prior program execution. Specifically, the output of the new RDD is represented as a *delta*, a pair of multisets $\Delta = (\Delta_+, \Delta_-)$ consisting of record insertions and deletions, respectively. This then allows us to employ the *delta rules* [16] approach of executing incremental versions of the downstream operators, which now take deltas as input and produce deltas as output.

Consider again the logical plan of Figure 5. The related physical plan (executing top-down) will look as follows:

```
1            → materialized join
2            →  FILTER (hour != 0)
3            → MAP (agency, 1)
4            → SHUFFLE
5            → REDUCE (agency, SUM)
```

Figure 6. Physical plan for the logical plan of Figure 5.

When we execute the newly introduced FILTER transformation at line 2 on the previously saved join results, we produce a Δ consisting of an empty Δ_+ and a Δ_- that contains all records that do not pass the added filter. Downstream transformations are then executed incrementally, taking deltas as input and producing output deltas. For example, the MAP on line 3 will produce a Δ_- record for each of the incoming ones. The REDUCE transformation on line 5 similarly uses the input Δ records to revise its results from the previous execution; note that this assumes the REDUCE results from the prior execution were materialized. Therefore, there is a space cost associated with incremental evaluation that is common across all incremental systems [8, 11, 16, 26]. These costs will be further explored in Section 3.3.

3. The VEGA Optimizations

This section describes our two approaches to optimizing query re-execution. We introduce a simple formal model of an execution workflow, which is an abstraction of both the logical and physical plans shown in the previous section. We use this formal model to precisely define our plan rewriting and incremental computation optimizations.

3.1 The Program Model

We model the dataflow of a program as a sequence of *transforms*, each of which is a data-parallel function such as map, filter, and reduce. Please see the previous section for the semantics of such functions in Spark. Consider a dataflow composed of n transforms $T_1 \to T_2 \to \ldots \to T_n$, where the input to transform T_i is the output of T_{i-1}.[1] The output of a transform is a multiset. We assume that a user has already executed the program, and that the output of transform T_i is represented as O_i. The user observes the final output O_n and decides to add a new transform δT_α after T_k, with ($1 \leq k < n$). The revised dataflow is $\ldots \to T_k \to \delta T_\alpha \to T_{k+1} \to \ldots T_n$. We handle multiple inserted transforms one at a time. Dele-

[1] W.l.o.g. we omit the input dataset for transform T_1.

U (other) $\rightarrow$	Filter		Map		Shuffle	Reduce	Join
T (target) $\downarrow$	Key	Value	Key	Value			
Filter(Key)	T	T	Filter $(T.f \circ U.m^{-1})$	T	T	T	T
Filter(Value)	T	T	T	Filter $(T.f \circ U.m^{-1})$	T	None	Filter $(T.f \circ U.col_1)$
Map(Key)	None	T	Map $(U.m \circ T.m \circ U.m^{-1})$	T	$\begin{cases} T \text{ if } T.m \text{ invertible} \\ \text{shuf} \circ T \text{ otherwise} \end{cases}$	$\begin{cases} T \text{ if } T.m \text{ invertible} \\ U \circ \text{shuf} \circ T \text{ otherwise} \end{cases}$	None
Map(Value)	T	None	T	Map $(U.m \circ T.m \circ U.m^{-1})$	T	T if $T.m$ distributes over $U.r$	Map $(T.m \circ U.col_1)$

Table 1. Rewrite rules for commuting Spark transformations.

tion and modifications of transorms are discussed at the end of the section.

As noted in the previous section, our approach depends on the reuse of materialized intermediate results from a previous execution. By default, VEGA retains stage inputs (i.e., shuffle outputs), and full job outputs; other materialization points can be specified by the programmer. Retaining results at input stage boundaries incurs minimal I/O overhead (cf. Section 5.1) since this intermediate data is already materialized by Spark shuffle. VEGA retains these materialized results (possibly on disk) beyond the lifetime of a given job for possible reuse in speeding up a subsequent query execution.

3.2 Logical Plan Rewriting

Recall again the revised dataflow $T_1 \rightarrow \ldots \rightarrow T_k \rightarrow \delta T_\alpha \rightarrow T_{k+1} \rightarrow \ldots T_n$ after a new transform δT_α has been added. Let O'_j be the output produced by transform T_j in the revised dataflow. Clearly when $1 \leq j \leq k$, then $O_j = O'_j$. Therefore, we need only re-execute starting from the last materialization point in $T_1 \rightarrow \ldots \rightarrow T_k$. The goal of query plan rewriting is to go beyond this by enabling a later materialization point to be used instead, without changing the final result of the program.

The key idea of query plan rewriting is to identify a new transform $\delta T'_\alpha$ (possibly equivalent to δT_α) such that $T_{k+1} \rightarrow \delta T'_\alpha$ is equivalent to $\delta T_\alpha \rightarrow T_{k+1}$ (i.e., they produce the same output when given the same input). Repeated applications of this idea cause the newly introduced transform to move farther downstream, modified as necessary to maintain the semantics of the original program. This process can be repeated until either we reach the last materialization point or we encounter a transform that the newly introduced transform cannot move past.

VEGA currently only supports pushing filters and maps past other transformations, because we found that these are the main operations that are added/modified iteratively in workflows. We have developed a set of rules for pushing these two kinds of transformations past other transformations without changing program behavior. Next we describe these rules in their full generality; Section 4 describes how these rules are employed in the context of our VEGA SQL and VEGA RDD implementations.

Commutativity Rules. Table 1 presents the rules that drive our program rewriting optimization. Each row contains a "target" transform T, which is added to the workflow, and each column contains a transform U that is directly after T in the workflow. The cell in the table for T/U contains the code T' such that $T \rightarrow U$ is equivalent to $U \rightarrow T'$. For example, if T is a transform that filters only keys (first row), and U is a map only on values (fourth column), then T can be safely commuted with U and no rewrite is necessary. The table uses several notational conventions: $T.f$ refers to the filter function if T is a filter transform; $T.m$ refers to the map function if T is a map transform; $T.r$ refers to the reduce function if T is a reduce transform; $T.col_1$ refers to the first non-key column if T is a join transform; shuf refers to a shuffle operation; m^{-1} refers to the inverse of m; and $\circ$ denotes function composition, i.e., $(f \circ g)(x) = f(g(x))$.

We briefly describe the table entries. A filter on keys (first row) commutes with other filters as well as transforms that do not modify keys, which includes shuffle, reduce, and join. The more interesting case occurs when a filter on keys T is followed by a map on keys U. In that case, pushing T past U requires in general that U *be inverted* before applying the filter function $T.f$. Therefore the new filter function is $T.f \circ U.m^{-1}$. A filter on values (second row) is handled analogously with respect to later filters and maps. However, a filter on values cannot move past a reduce operation, which in general provides no way to recover the original values. Finally, moving a filter on values past a join requires the filter function to first select the column containing the original values; $U.col_1$ is used for that purpose.

A map on keys (third row) is pushed past filter and map transforms using the same techniques as described above. If the map function is invertible, then it is one-to-one and hence preserves the grouping of keys done by a shuffle transform. Such maps can be safely commuted with a shuffle as well as a reduce transform. If the map on keys is not invertible, we require another stage of shuffling after the map. However, this shuffle is generally efficient, as it operates only on the records that are modified by the map. Finally, the only new case for a map on values (fourth row) involves pushing it past a reduce transform. In general, this is not possible, because the original values are not recoverable after the reduce's aggregation. However, in the case where the map function *distributes* over the reduce function, we can safely apply the map after the reduce aggregation is completed. The distributive property is defined as follows: $\forall a,b, U.r(T.m(a), T.m(b)) = T.m(U.r(a,b))$. For example, a map function that doubles each value can be pushed after a reduce function that sums the values (since $2x + 2y = 2(x+y)$).

Algorithm 1 Dataflow Rewriting

```
Input:  An annotated dataflow; An injected transform $\delta T$;
        The index of $\delta T$ in the dataflow.
Output: The rewritten version of $\delta T$;
        The optimal position index for the rewritten $\delta T$.
 1: currentIndex := index +1
 2: $\delta T'$ := $\delta T$
 3: $U$ := dataflow.from(currentIndex).next()
 4: continue := dataflow.from(currentIndex).exists(hasMP())
 5: while continue and $U \neq$ null do
 6:     res := $\delta T'$.commuteWith($U$).
 7:     if res == None then
 8:         break
 9:     $\delta T'$ := res.get()
10:     if $U$.hasMP() then
11:         $\delta T$ := $\delta T'$
12:         index := currentIndex
13:     currentIndex +1
14:     $U$ := dataflow.from(currentIndex).next()
15:     continue := dataflow.from(currentIndex).exists(hasMP())
```

Plan Rewriting. Algorithm 1 describes a dataflow rewriting algorithm, which leverages the commutativity rules described above. The algorithm takes as input the index of the new transform δT in the dataflow, and it iteratively calls the commuteWith function on the immediate downstream transform (line 6) until None is returned (line 7) or no downstream materialization point exists (line 15). A call T.commuteWith(U) decides if transforms T and U, adjacent to each other in the plan, can be commuted using the rules in Table 1. If commuteWith returns None then the transforms do not commute; otherwise a transform T′ (line 9) is returned such that U followed by T′ has the same behavior as T followed by U. The Algorithm returns the optimal position for δT and the final rewritten form of that transform (respectively set in lines 12 and 11). index and δT are updated only when a materialization point is reached (line 10).

3.3 Incremental Execution

Once we have pushed the new transform as far as possible, the dataflow has the form $\delta T_\alpha \rightarrow T_{i+1} \rightarrow \dots T_n$ where δT_α is the newly introduced transformation pushed to some materialization point O_i.[2] VEGA has the ability to execute this dataflow *incrementally* using delta rules [16], or from scratch if it determines that the incremental plan is too costly (discussed further in Section 4.2.2).

δT_α starts the delta computation by producing the pair $(\Delta_{\alpha+}, \Delta_{\alpha-})$ where $\Delta_{\alpha+} = O_\alpha \setminus O_i$ and $\Delta_{\alpha-} = O_i \setminus O_\alpha$. In our running example from Section 2, the delta version of the new filter will become hour => if(hour == 0) −(hour), where each −(hour) result contributes to the $\Delta_{\alpha-}$ multiset (and $\Delta_{\alpha+}$ is empty). VEGA then uses delta rules to replace each T_t in the remainder of the workflow with its incremental version ΔT_t. Each ΔT_t outputs two multisets, Δ_{t+} and Δ_{t-} such that $O'_t = (O_t \cup (\Delta_{t+} \setminus \Delta_{t-})) \setminus (\Delta_{t-} \setminus \Delta_{t+})$.

[2] As discussed above, the new transformation may have been modified by VEGA as it was pushed later in the workflow, but we elide this detail from our notation as it is irrelevant.

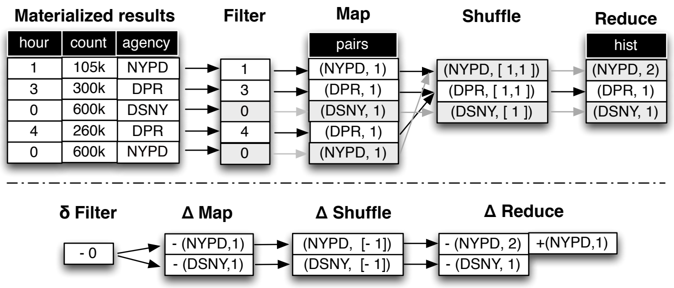


Figure 7. Incremental computation of the physical plan from Figure 6. The top of the figure highlights the data modifications w.r.t. the initial run. The bottom of the figure shows how Δs are computed and propagated downstream.

Example: Incremental Re-execution – Continuing from the running example and the physical plan of Figure 6, VEGA applies the filter over the output of the join; any input that does not satisfy the filter will be added to a Δ_- result, which is then propagated downstream. The REDUCE operator uses the aggregated Δ records to revise its result state. In this particular case that means removing all midnight calls from every agency. Figure 7 illustrates the execution of the above incremental plan on a small sample of data. The top portion of the figure illustrates the execution of the original program, together with the modifications required by the injection of the filter (highlighted in gray). The incremental plan is described in the bottom part of Figure 7. We can notice that the delta evaluation of the REDUCE outputs two Δ records for the NYPD agency: $-(\mathsf{NYPD}, 2)$, and $+(\mathsf{NYPD}, 1)$. This is because the REDUCE incremental operator has to update the count for NYPD from 2 to 1, and eventual downstream operators must be notified of the change. Conversely, $(\mathsf{DSNY}, 1)$ is only removed. Hence only the delta record $-(\mathsf{DSNY}, 1)$ is issued.

Discussion. Although we have only discussed how we manage the addition of transforms, our approach also includes some support for *deleting* and *modifying* existing transforms. Δ-based incremental computation handles both kinds of changes naturally. It simply requires that a “diff” be taken of the output of the transform before and after the modification/deletion, in order to produce the initial delta multisets; the downstream process is unchanged. Query plan rewriting handles both deletion and modification on maps, as long as the map function is invertible: deletion has the same effect as adding the inverse, while updating a map is simulated by adding the inverse of the original map followed by the revised map. The removal or modification of a filter cannot be handled with our rewriting technique.

Summary. Using query plan rewriting we are able to push modifications to later materialization points, saving the upstream transformation work. Incremental evaluation allows us to efficiently re-execute portions of queries where rewrites do not apply. Section 5 shows that these two techniques can provide significantly better performance compared to Spark. Next, we describe the implementations of VEGA for Spark.

4. Spark VEGA Library

The VEGA library implements the query rewriting and incremental processing techniques described in the previous section. The library consists of two modules: VEGA SQL (Section 4.1) and VEGA RDD (Section 4.2). Briefly, VEGA SQL implements the query rewrite technique for Spark SQL queries, while VEGA RDD implements the query rewrite and incremental processing techniques at the lower-level Spark RDD API. Due to the high-level Spark SQL semantics, query rewriting in VEGA SQL is completely transparent to the user, i.e., no additional information is required from the programmer. VEGA RDD instead trades transparency for a larger space for optimizations: VEGA RDD comes with a specifically tailored API allowing a larger class of rewrites (e.g., across map operations) and delta evaluation.

4.1 VEGA SQL

VEGA SQL supports rewrites that push filters past downstream materialized results from previous executions, as shown by the example in Section 2.2. There are no explicit maps at the SQL level; however, the lower-level VEGA RDD framework (described next) supports rewriting both filters and maps. VEGA SQL makes the following modifications to the Spark SQL compiler (i.e., Catalyst):

1. The query plan rewriting logic of Algorithm 1 is added as a new logical optimization rule;
2. The existing filter push down rule is disabled for the filter operators already optimized by Algorithm 1;
3. We force Catalyst to cache the output of exchange operators (i.e., shuffle output) as materialization points.

In total, the added rule logic to Catalyst amounts to less than 100 lines of code. We are actively working on optimizing the mechanisms associated with caching Spark SQL exchange operators triggered by rule 3. When Spark SQL caches such intermediate data, it converts that data to an in-memory columnar format, which can cause significant compute overheads. Presently, this caching happens synchronously with the execution of the Spark SQL operators, thereby slowing down the query progress; we report on this slowdown in Section 5. Since our goal is to potentially use this cached result in a subsequent job, after a user has revised the query, it suffices for our purposes to perform the caching asynchronously; we are working on this change, which will minimize the impact on the active running query. This formatting of cached data is specific to Spark SQL and does not occur in programs written in the lower-level Spark RDD abstraction. Therefore VEGA RDD (described next) does not incur this extra overhead when it retains materialized data produced by stages.

4.2 VEGA RDD

While Spark users often employ the high-level SQL API, it is also common for programmers to directly create Spark programs as a dataflow of RDDs. VEGA RDD extends the Spark RDD abstraction with mechanisms that enable transformation rewrites to take advantage of later materialization points, and operator implementations that incrementally evaluate transformations.

4.2.1 API

VEGA RDD provides an API that allows programmers to obtain VEGA's optimizations in an interactive development setting. Figure 8 lists the main VEGA RDD abstraction: the Transform class, which wraps a Spark RDD and exposes a similar API. Like Spark RDD transformations, VEGA RDD transforms are evaluated *lazily* when an action is called.

```
abstract class Transform[A,B](val rdd:RDD) {
  // Public API
  // Basic transforms
  def map(f:(A=>B),finv:(B=>A)):Transform[A,B]
  def flatMap(f:(A=>Iterable[B])):Transform[A,B]
  def filter(f:(A=>Bool)):Transform[A,A]
  // Pairwise transforms
  def mapKey(f:((K,V)=>(K1,V)),
    finv:((K1,V)=>(K,V))):Transform[(K,V),(K1,V)]
  def mapValue(f:((K,V)=>(K,V1)),
    finv:((K,V1)=>(K,V))):Transform[(K,V),(K,V1)]
  def filterKey(f:(K=>Bool)):Transform[(K,V),(K,V)]
  def filterValue(f:(V=>Bool)):Transform[(K,V),(K,V)]
  def join(o: Trasform[(K,V2)]):Trasform[(K,(V1,V2))]
  def reduceByKey(f:((V,V)=>V1),finv:(V=>V),
    fzero:(V=>Bool)):Transform[(K,V),(K,V1)]

  // Workflow operations
  /* Injecting transform 't' in the workflow
  immediately past the target one */
  def inject(t:Transform[A,B]):this.type
  def delete(t:Transform[A,B]):this.type
  /* Modify works only on map transforms. The new
  map function 'f' must have the signature of the
  original map (for instance a map over keys cannot
  become a map over values */
  def modify(f:(A=>B),finv:(B=>A)):this.type
  // Force Vega to materialize the target transform
  def materialize():this.type
  // Run the full workflow and return the result
  def collect():Array[A]

  // Private API
  ...
}
```

Figure 8. VEGA public target API.

There are three main differences between the VEGA RDD API and the original Spark one. First, VEGA RDD introduces variants of map and filter that identify the part (i.e., key or value) of a key-value pair that the transform modifies or reads. For example, mapKey ensures that the transform will only map over keys, leaving the associated values unchanged. Such variants allow VEGA to employ the rules in Table 1 without requiring analysis of the user-defined functions in each transform. For example, a filterKey transform is guaranteed to safely commute with a mapValue transform.

Second, the map transform in VEGA RDD accepts an inverse function (in addition to the ordinary function argument). When the inverse is `null`, the function is assumed to be non-invertible; otherwise, the inverse can be used to enable more rewriting, as shown in Table 1. VEGA RDD includes a suite of several standard functions (e.g., string reverse, pairWithOne) along with their inverses, which can be directly used in map transforms. The reduceByKey transform similarly takes two extra functions allowing a reduce operation to be inverted during incremental evaluation: the first defines how to remove values from the aggregate (e.g., minus for sum); the second helps understand when the "empty" value is reached for the aggregate (e.g., 0 for sum).

Finally, the VEGA RDD API includes explicit operations to insert, delete, or modify transforms in an existing workflow. VEGA RDD includes an operation that allows the programmer to explicitly define new (intra-stage) materialization points, in addition to the default (inter-stage) ones; these additional materialization points incur extra space and time costs w.r.t. native Spark.

4.2.2 Implementation

The VEGA RDD implementation performs rewriting of the physical plan using the approach described in Algorithm 1 in the previous section. The VEGA Execution Planner (EP) is then responsible for translating the rewritten physical plan into one of two possible execution plans: a *standard plan*, or an incremental Δ *plan*. If the plan was previously executed, then the execution plan begins at the latest materialization point that precedes the point where the workflow is modified. All work leading up to that materialization point is avoided.

In the case of a standard plan, VEGA EP simply translates each transform to a regular Spark transformation that executes natively. The resulting execution plan persists outputs at the default and programmer-specified materialization points. In the case of an incremental plan, VEGA EP does the following: (1) all transforms between the input materialization point and the new transform are executed natively via Spark transformations; (2) the δ transform is compiled into a transformation that generates Δ results; (3) transformations that follow the δ transform are incrementally processed according to delta rules [16]. The approach used in step (2) above depends on the kind of transformation that was inserted. For a new filter (and all variants), any record that does not satisfy the filter condition is added to the Δ_- set. For a new map (and all variants), the transform is executed normally and its result is then "diff"ed with the stored intermediate results from the same point in the previous execution, in order to produce the appropriate Δ_- and $\Delta+$ sets.

Dynamic Plan Swapping. By default VEGA EP employs incremental execution. However, this is not always a performance win over regular execution, particularly when the sizes of the Δ sets are large. In the worst case, if an inserted map transform changes the format of all records, then Δ_- will include all of the old materialized records and Δ_+ will include all outputs from the new map, so the total number of records to propagate is twice that if we used regular execution. To overcome this problem, we have instrumented VEGA RDD to collect statistics when Δ plans are submitted to execution. The system is able to switch to a standard plan when it detects that the delta sets are growing too large.

```
1   format = new SimpleDateFormat("M/d/y h:m:s a")
2   input = sc.textFile("hdfs://...")
3   inputTr = new Transform(input)
4   calls = inputTr.map(line=>format.parse(line.split(","))
5     .map(r =>(r(0),r(1).getHours(),...),null)
6   agencyPerHour = calls.map(x=>(x._2,(x._4, 1)),null)
7   pairs = calls.map(x => (x._2, 1),null)
8   count = pairs.reduceByKey(_+_,-1*_, _==0)
9   filter = count.filter(_._2 > 100000)
10  join = filter.join(agencyPerHour)
11  agencies = join.map(_._2._2,null)
12  result = agencies.reduceByKey(_+_,-1*_, _==0)
13  result.collect.foreach(println)
```

Figure 9. Target program using the VEGA API and generated from the Spark SQL program of Figure 2.

Example: Full VEGA RDD Pipeline – We now describe how VEGA RDD works in practice using our running example. Figure 9 shows how a user can implement the query of Figure 2 directly using the VEGA RDD API. The program is identical to the logical plan discussed previously, except (1) a transform wrapping the input RDD is added, to define the source of the VEGA RDD workflow (line 3); and (2) the reduce transforms include functions that allow them to be inverted (lines 8 and 12).

When the collect method is called, since this is the first time the query is executed, VEGA RDD will directly run the plan without any optimization, and data will be saved at the default materialization points (i.e., before transformations defining Spark stage inputs). Assume now that a user wants to inject the filter removing the midnight occurrences. This is implemented in VEGA RDD with the following line of code.

```
pairs.inject(pairs.filterKey(x => x!=0))
```

When the collect method is called again, VEGA RDD first tries to optimize the program using the query plan rewriting. Algorithm 1 detects that the new filter can be pushed to just before the later join transformation because (1) it commutes with the `reduceByKey` and `filter` operations of lines 8 and 9; (2) it does not commute with the `map` of line 11; and (3) even though it commutes with the join operation in 10, the latest reachable materialization point is on the input of the join. Once the rewritten plan is generated, VEGA RDD compiles it into an incremental execution plan where each RDD transformation takes as input a set of positive and negative Δs. The initial Δ_- is generated with the records not satisfying the filter condition.

5. Evaluation

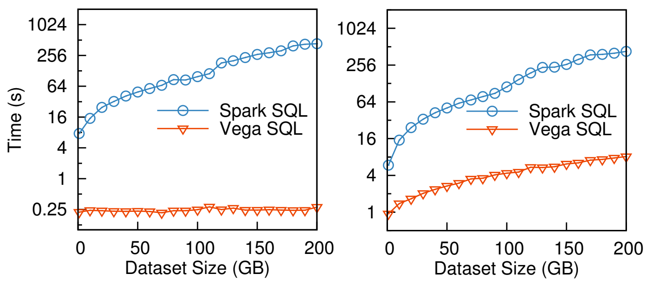
In this section, we evaluate our two key mechanisms—incremental evaluation and query rewrites—for improving the response time of re-executing Spark programs (and Spark SQL queries) after some modification. Our experiments focus on three workloads:

1. *Wiki reverse:* The WikiReverse project [5] aims to understand how people use Wikipedia on the web. It contains statistics such as the number of links incoming to `wikipedia.org` domain, the number of popular websites linking to Wikipedia, and many others. We will use this scenario to showcase the performance improvements offered by VEGA SQL, and the cost of materialization.
2. *PigMix:* A popular benchmark for large-scale data analytics. We use this workload to motivate our dynamic plan selection, by showing cases when re-running from scratch is preferable to incremental evaluation i.e., Δ plans are not silver bullets.
3. *Word count:* A simple workload showing that commutative plans run in time (1) independent of the size of the input dataset, and (2) dependent on the unique number of words existing in the dataset.

Datasets. The word count experiments use two datasets of sizes ranging from 2GB to 200GB. The *Word Bag* dataset (taken from [20]) contains 8000 unique words generated from Zipf distribution. The *Wikipedia* dataset contains words that are generated from randomly sampling Wikipedia, which itself contains approximately 56 million unique words. WikiReverse uses the Common Crawl dataset [2], which comes from the Common Crawl non-profit foundation that collects data from web pages. For this scenario we will use sizes ranging from 1GB to 200GB. Finally, we use the PigMix generator to create datasets of sizes ranging from 50GB to 200GB.

Experiments Configuration. All experiments were carried out on a cluster of 16 machines, each with a 3.40GHz i7 processor (4 cores and 2 hyper-threads per core), 32GB of RAM and 1TB of disk capacity. The operating system is 64-bit Ubuntu 12.04. The datasets were all stored in HDFS version 1.0.4 with a replication factor of two. VEGA uses Spark 1.4.0 as the execution engine for running the workflows. Materialization points are persisted using the MEMORY_AND_DISK_SER level.

For each VEGA experiment we run the initial workflow, make a change (e.g., add a filter), and run the modified workflow. In the PigMix scenario we compared two plans: i.e., Δ (incremental evaluation), and standard (from scratch starting from a materialization point). Materialization points are created according to the default VEGA policy i.e., only at Spark stage boundaries. The VEGA results are compared against native Spark, which always runs the entire workflow from scratch. Each experiment is run seven times: the first two runs are used to warm up OS caches; from the remaining 5 runs we report the trimmed mean computed by removing the top and bottom results and averaging the other three.



(a) Query 1   (b) Query 2

Figure 10. Results for the WikiReverse scenario (log-scale used for the time axis). For this scenario VEGA SQL is over two order-of-magnitude faster than Spark.

5.1 Query Rewrite

This section leverages the Common Crawl dataset to evaluate the benefits (and costs) of our rewrite technique in VEGA SQL. Each page record defines the page URL, from which a domain (e.g., `cnn.com`) can be extracted. Both queries described in this section leverage a DataFrame LINKS(domain, link), which associates the reference page domain with the links contained in that page.

Query 1. Our first query computes how many pages in the Common Crawl dataset point to a Wikipedia page. The Spark SQL query below gives the count of the number of domains for each unique link.

```
SELECT link, count(*)
FROM links
GROUP BY link
```

After running the program, the analyst realizes that the query does not filter LINKS to only include Wikipedia pages. Instead, this query returns how many domains reference each link, regardless of whether that link references a Wikipedia page. The analyst fixes the bug by adding a selection predicate to only include the Wikipedia links.

```
SELECT link, count(*)
FROM links
WHERE link like '%wikipedia.org%'
GROUP BY link
```

VEGA SQL is able to rewrite the query and push the filter to the end, leveraging the materialization point on the output of the previous query. The response time for this query under various input data sizes is presented in Figure 10(a). With commutative rewrites, VEGA SQL is able to outperform Spark by over three orders-of-magnitude.

Query 2. Building off of the previous query, the analyst would now like to measure the number of popular domains referencing Wikipedia pages:

```
SELECT links.link, count(*)
FROM links JOIN popular ON domain
WHERE links.link like '%wikipedia.org%'
GROUP BY links.link
```

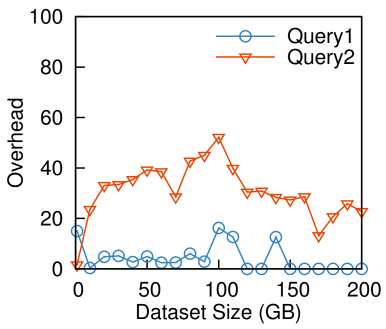


Figure 11. Overhead of materializing intermediate results in VEGA SQL compared against the base case.

The analyst realizes there is a small problem with the above query: `wikipedia.org` itself is among the popular websites and therefore self references (i.e., internal Wikipedia links) are included in the count. The following revised query removes records that refer to Wikipedia:

```
SELECT links.link, count(*)
FROM links JOIN popular ON domain
WHERE links.link like '%wikipedia.org%'
 AND links.domain not LIKE '%wikipedia.org%'
GROUP BY links.link
```

VEGA SQL is able to recognize that the filter commutes with the `join`, however is not able to push it past the successive `group by` operation, which projects out the domain attribute. Therefore, the latest applicable materialization point exists after the shuffle operation preceding the `join`.[3] Consequently, VEGA SQL rewrites the query to perform a map-side join on the materialized (shuffled) partitions of LINKS and POPULAR, followed by the group-by count aggregation. In contrast, Spark SQL will execute this query from scratch, performing shuffle-based hash-join on LINKS (build phase) and POPULAR (stream). The results for re-executing this query revision are plotted in Figure 10(b), which shows that VEGA SQL outperforms Spark SQL by up to two order-of-magnitude.

Materialization Cost. Figure 11 depicts the overhead of saving partial results in VEGA SQL. As we can see, the cost of materialization is minimal for Query 1 (always less than 20%), while is in average around 30% for Query 2, with a peak of 52% for 100GB. [4] In general, we deem this costs as reasonable compared with the two or more order-of-magnitude saving when doing the re-execution. However, as mentioned in Section 4.1, we are actively working on optimizing the materialization mechanisms in VEGA SQL to be asynchronous with the target query execution.

[3] The stage boundary follows the shuffle and proceeds the join.

[4] Note that the irregular overhead is a consequence of the underlying words distribution in the dataset.

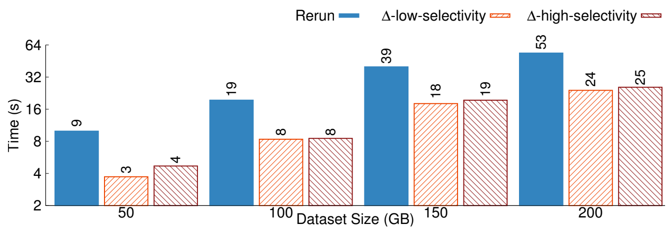


Figure 12. Comparison for query L3 between Δ evaluation and rerun from intermediate results.

5.2 Incremental Evaluation

We now turn our attention to evaluating benefits of incremental evaluation in the context of VEGA RDD. Our evaluation leverages two queries from the PigMix benchmark.

PigMix L3 computes the total estimated revenues for each user visiting a webpage. We added a selection predicate that we use in our experiments. The query involves a join between a page_view table and a users table, followed by a group by operation that sums up the revenues per user. The added selection predicate over the users randomly prunes records based on a selectivity parameter. In our experiments, we will remove this selection predicate and evaluate the performance of an incremental evaluation that adds previously filtered users. A more selective predicate will add back more users, increasing the size of the Δ_+ set. The VEGA RDD plan for query L3 is as follows:

FILE("...\users")
$\rightarrow$ MAP (line $\Rightarrow$ ...)
$\rightarrow$ MAP (name, 1)
$\rightarrow$ FILTERKEY(Rand.nextFloat $\leq$ sel) $\rightarrow$ u

FILE("...\page_views")
$\rightarrow$ MAP (line $\Rightarrow$ ...)
$\rightarrow$ MAP (user, revenues)
$\rightarrow$ SHUFFLE
$\rightarrow$ JOIN (u)
$\rightarrow$ SHUFFLE
$\rightarrow$ REDUCEBYKEY (user, SUM)

Our experiment first executes query L3 with the FILTERKEY transform, and then re-executes it without the FILTERKEY transform. Commutative rewrites are not applicable here, so we are left with two options: from the closest materialization point, execute incrementally or from scratch. Figure 12 depicts the performance of these two options. In general we can see that Δ evaluation is 2-3 times faster than re-running the query from scratch. Interestingly, the selectivity of the filter has very little effect on the performance of the Δ plan. The reason for this is due to our incremental hash-join implementation, which hashes users and streams the page_view table; Δ records revise the previously hashed users table, after which the page_view table is streamed from the SHUFFLE materialization point. In this particular query, page_view is considerably bigger than users, hence the stream scan dominates the performance.

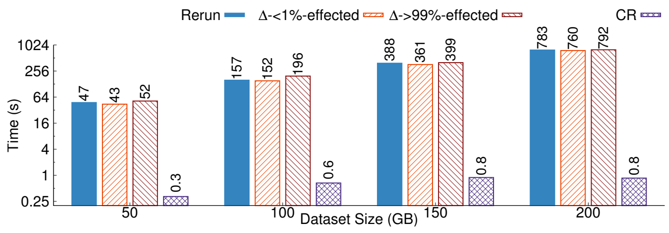


Figure 13. Comparison for query L2. Here Δ evaluation is of no help. Commutative rewrites (CR) are applicable in this case, and outperforms the other techniques.

PigMix L2 returns estimated revenues from page visits coming from "power users". Similar to L3, it joins two tables, page_view and power_users, of different sizes, i.e., page_view is considerably larger than power_users. This query is represented by the following VEGA RDD plan.

FILE("...\power_users")
→ MAP (line ⇒ ...)
→ MAP (name) → p

FILE("'...\page_views")
→ MAP (line ⇒ ...)
→ MAP (users, revenues)
→ SHUFFLE
→ JOIN (p)

For this experiment, we modify the query (shown below) by randomly changing the revenue value for a user to 0 (i.e., no revenue) based on a parameter that allows us to select the number of affected records.

FILE("'...\page_views")
→ MAP (line ⇒ ...)
→ MAPVALUE (Rand.nextFloat≤par?revenue:0)
→ SHUFFLE
→ JOIN (p)

The Δ records in this scenario will include both a removal of the previous version and the addition of the new version. However, this particular modification can be rewritten by moving the MAPVALUE transform to the output of the join. Figure 13 shows the results of executing the revised query from scratch, along with an incremental evaluation with rewriting and without. As shown, the incremental evaluation without rewriting does not outperform the simple rerun. On the contrary, when the percentage of the record affected is high, incremental evaluation performs worse than the rerun. This is because the size of Δ set for this case is twice the number of total records. This case happens when all the previous records are eliminated, while the new version of each record is added. For this query incremental evaluation does not help because the modification is on the bigger table: to create the delta set a full scan of the table must be executed, so even if few records are actually affected by the map, the improvement is limited by this big scan.

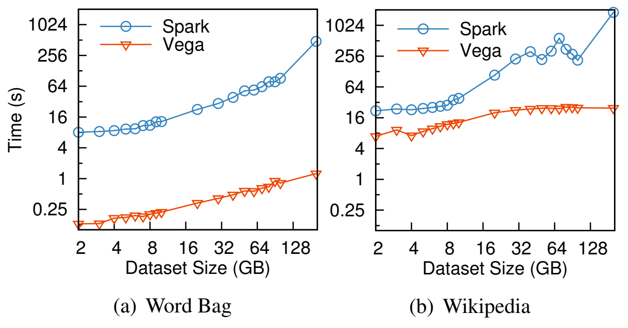


Figure 14. Performance for the modified word count using Word Bag and Wikipedia datasets (log-scale used for both axis). In Wikipedia the output contains up to 54 millions words; the running time of the CR plan therefore increases w.r.t. the Word Bag dataset.

5.3 Word Count

This section measures the performance of our commutative rewrites in VEGA RDD on a simple word-count job.

FILE("...") → FLATMAP (line.split(" "))
→ MAP (word, 1)
→ SHUFFLE
→ REDUCEBYKEY (word, SUM)

The FLATMAP transform splits every line into words, then MAP associates an initial count (i.e., 1) with every word, and finally REDUCEBYKEY aggregates all counts for each word by summing them up.

The following query modification adds a map adjoining a suffix to every word in the dataset. This could be useful to convert the output into CSV format, for example.

FILE("...") → FLATMAP (line.split(" "))
→ MAP (word, 1)
→ MAPKEY (word ⇒ word + suffix)
→ SHUFFLE
→ REDUCEBYKEY (word, SUM)

VEGA RDD pushes the map all the way to the end of the dataflow, i.e., past the final materialization point. Figure 14 compares the performance of running the modifiedquery with Spark wrt using the VEGA RDD plan produced from the commutative-rewrite (CR). Here we used both the Word Bag (Figure 14(a)) and the Wikipedia dataset (Figure 14(b)).

As can be noticed from the regular plan plot of Figure 14(b), the execution of the word count program might depend not only on the dataset size but also on words distribution. This behavior is not noticeable in Figure 14(a) because the number of unique words in the Word Bag dataset is small. Conversely, using rewrites VEGA RDD computes incremental results one to three orders of magnitude faster than re-running the computation, and is independent on both (1) input dataset size, and (2) words distribution. The results of Figure 15 lead to the following two observations for commutative plans: Firstly, in Word Bag the number of unique words is small. Hence, VEGA RDD performs incremental computations in roughly the same time, even when the input size increases. Secondly, since Word Bag has few unique words, its output is significantly smaller than the Wikipedia dataset, and hence VEGA performs better on the former.

This experiment highlights an important feature of VEGA—it computes incremental results in time proportional to the output of the modified transformation. Many big data workflows start out with a large input dataset but reduce it down to significantly smaller sizes after processing; VEGA performs especially well on such workflows.

6. Related Work

Answering queries using views addresses the problem of rewriting a query in terms of a given set of views, for example to enable data integration or query optimization [18]. Our problem can be seen as an instance of answering queries using views, where the materialized intermediate results from a previous execution of the Spark program constitute the views. However, our setting of interactive program development leads to a very different set of challenges and opportunities. Traditionally, the challenge for answering queries using views is to rewrite an arbitrary query in terms of the view relations, leading to techniques based on logical query containment and equivalence. In our setting the challenge is instead to rewrite the query *in terms of the base relations* in order to push a specific modification as late as possible in a query plan; this reasoning, based on a commutativity analysis, is completely independent of the particular views.

Incremental view maintenance is a well studied problem [10, 17, 27] for efficiently handling changes to base tables used in view definitions. REX [26] and Naiad [25] leverage incremental view maintenance techniques, such as delta rules [16], to speed up iterative computations e.g., recursive queries and graph algorithms. In contrast, VEGA addresses the problem of handling changes to the query itself efficiently. We show how to leverage incremental techniques from this prior work as part of our solution.

Cache-based systems such as [23, 28] try to optimize recurring (fixed) queries over evolving data by materializing partial results and taking advantage of the overlap in results between successive executions. Similarly, systems such as Nectar [15] and Tachyon [24] (and to some degree Spark itself) store (workflow) lineage information to speed-up shared sub-computations and for fault-tolerance. If applied to the iterative program development scenarios that VEGA is targeting, such systems are at best able to resume the computation up to the latest available materialization point before the query modification, and run incrementally from there (e.g., Nectar). VEGA instead is also able to take advantage of later materialized results by using query rewriting. For instance, Figure 13 in Section 5 shows a case in which incremental evaluation is of no help for query re-execution, while VEGA is more than 100X faster.

Database query optimization is traditionally a two-phase process: first, query rewrites transform the logical plan into an equivalent plan based on optimization heuristics e.g., pushdown selection predicates and projections; second, a physical plan is selected based on a cost model and search strategy that enumerates some subset of equivalent plan options. The traditional query planning process does not take into account results of previous executions of a similar query. For example, Catalyst is an extensible optimizer for Spark SQL [7], and—like other traditional optimizers [12, 30]—its search strategy considers pushing filters as close to the input source as possible. Hueske *et al.* [19] show how to leverage an analysis of user-defined functions in the context of a DISC system to enable even more traditional query optimizations. In contrast, VEGA pushes filters and maps toward the end of the workflow so that prior materialized results can be used to avoid redundant work. However, our work shares with the prior work the need to reason about the commutativity of operations. Hueske's analysis of user-defined functions could be adapted in VEGA to enable more rewriting, and our commutativity conditions would likewise be useful in the context of traditional query optimization.

7. Conclusion and Future Work

In this paper we presented VEGA: a library adding explicit support for interactive query development to Apache Spark. VEGA employs a novel rewriting technique to maximize the reuse of previous computations, and it leverages incremental computation to minimize the overhead of re-execution.

Our implementation of VEGA is completely external to Spark. In this way (1) we can easily port VEGA to different platforms (e.g., Hadoop [3]), (2) we avoid having to modify the RDD semantics, and (3) we can provide Spark's fault-tolerance guarantees at no additional cost.

For some of its optimizations, VEGA relies on function invertibility in order to "go back" to an earlier state of the computation. An alternative approach is to leverage available fine grained lineage information [21], which keeps track of the input records of each transform. For example, to invert a map's output, we can simply consult the map's data lineage. Nonetheless, the current requirements on invertibility have not been overly onerous in practice: for example, VEGA has successfully enabled interactive debugging sessions for distributed workflows [13, 14].

Acknowledgements

Vega is supported through NSF grants CNS-1161595, IIS-1302698, CNS-1239498, and CNS-1351047, as well as U54EB020404 awarded by the National Institute of Biomedical Imaging and Bioengineering (NIBIB) through funds provided by the trans-NIH Big Data to Knowledge (BD2K) initiative (`www.bd2k.nih.gov`). We would also like to thank our industry partners at IBM Research Almaden and Intel for their generous gifts in support of this research.

REFERENCES

[1] 311 service requests dataset. https://data.cityofnewyork.us/Social-Services/311-Service-Requests-from-2010-to-Present/erm2-nwe9.

[2] Common crawl dataset. http://commoncrawl.org.

[3] Hadoop. http://hadoop.apache.org.

[4] Spark. http://spark.apache.org.

[5] WikiReverse. https://wikireverse.org.

[6] P. Agrawal, A. Silberstein, B. F. Cooper, U. Srivastava, and R. Ramakrishnan. Asynchronous view maintenance for vlsd databases. In SIGMOD '09, pages 179–192, New York, NY, USA, 2009. ACM.

[7] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia. Spark sql: Relational data processing in spark. In SIGMOD '15, pages 1383–1394, New York, NY, USA, 2015. ACM.

[8] P. Bhatotia, A. Wieder, R. Rodrigues, U. A. Acar, and R. Pasquin. Incoop: Mapreduce for incremental computations. In SOCC 2011, pages 7:1–7:14, New York, NY, USA, 2011. ACM.

[9] Y. Bu, B. Howe, M. Balazinska, and M. D. Ernst. Haloop: Efficient iterative data processing on large clusters. *Proc. VLDB Endow.*, 3(1-2):285–296, Sept. 2010.

[10] S. Ceri and J. Widom. Deriving production rules for incremental view maintenance. 1991.

[11] S. Ewen, K. Tzoumas, M. Kaufmann, and V. Markl. Spinning fast iterative data flows. *Proc. VLDB Endow.*, 5(11):1268–1279, July 2012.

[12] G. Graefe and W. J. McKenna. The volcano optimizer generator: Extensibility and efficient search. In ICDE '93, pages 209–218, Washington, DC, USA, 1993. IEEE Computer Society.

[13] M. A. Gulzar, X. Han, M. Interlandi, S. Mardani, S. D. Tetali, T. D. Millstein, and M. Kim. Interactive debugging for big data analytics. In *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016.*, 2016.

[14] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. D. Millstein, and M. Kim. Bigdebug: debugging primitives for interactive big data processing in spark. In ICSE '16, Austin, TX, USA, May 14-22, 2016, pages 784–795, 2016.

[15] P. K. Gunda, L. Ravindranath, C. A. Thekkath, Y. Yu, and L. Zhuang. Nectar: Automatic management of data and computation in datacenters. In OSDI '10, October 4-6, 2010, Vancouver, BC, Canada, Proceedings, pages 75–88, 2010.

[16] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. In SIGMOD '93, pages 157–166, New York, NY, USA, 1993. ACM.

[17] A. Gupta, I. S. Mumick, and V. S. Subrahmanian. Maintaining views incrementally. *ACM SIGMOD Record*, 22(2):157–166, 1993.

[18] A. Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[19] F. Hueske, M. Peters, M. Sax, A. Rheinländer, R. Bergmann, A. Krettek, and K. Tzoumas. Opening the black boxes in data flow optimization. *PVLDB*, 5(11):1256–1267, 2012.

[20] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *Fifth Biennial Conference on Innovative Data Systems Research* CIDR 2011, pages 273–283, Asilomar, CA, USA, January 9-12, 20112011.

[21] M. Interlandi, K. Shah, S. D. Tetali, M. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[22] S. Kandel, A. Paepcke, J. M. Hellerstein, and J. Heer. Enterprise data analysis and visualization: An interview study. *Visualization and Computer Graphics, IEEE Transactions on*, 18(12):2917–2926, 2012.

[23] C. Lei, E. A. Rundensteiner, and M. Y. Eltabakh. Redoop: Supporting recurring queries in hadoop. In EDBT '14, Athens, Greece, March 24-28, 2014, pages 25–36, 2014.

[24] H. Li, A. Ghodsi, M. Zaharia, S. Shenker, and I. Stoica. Tachyon: Reliable, memory speed storage for cluster computing frameworks. In SOCC '14, pages 6:1–6:15, New York, NY, USA, 2014. ACM.

[25] F. McSherry, D. G. Murray, R. Isaacs, and M. Isard. Differential dataflow. In *Conference on Innovative Data Systems Research (CIDR)*, 2013.

[26] S. R. Mihaylov, Z. G. Ives, and S. Guha. REX: recursive, delta-based data-centric computation. *PVLDB*, 5(11):1280–1291, 2012.

[27] V. Nigam, L. Jia, B. T. Loo, and A. Scedrov. Maintaining distributed logic programs incrementally. *Computer Languages, Systems & Structures*, 38(2):158–180, 2012.

[28] C. Olston, G. Chiou, L. Chitnis, F. Liu, Y. Han, M. Larsson, A. Neumann, V. B. Rao, V. Sankarasubramanian, S. Seth, C. Tian, T. ZiCornell, and X. Wang. Nova: Continuous pig/hadoop workflows. In SIGMOD '11, pages 1081–1090, New York, NY, USA, 2011. ACM.

[29] D. Peng and F. Dabek. Large-scale incremental processing using distributed transactions and notifications. In *OSDI*, volume 10, pages 1–15, 2010.

[30] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In SIGMOD '79, pages 23–34, New York, NY, USA, 1979. ACM.

[31] A. Shkapsky, M. Yang, M. Interlandi, H. Chiu, T. Condie, and C. Zaniolo. Big data analytics with datalog queries on spark. In SIGMOD '16, pages 1135–1149, New York, NY, USA, 2016. ACM.

[32] R. S. Xin, J. Rosen, M. Zaharia, M. J. Franklin, S. Shenker, and I. Stoica. Shark: Sql and rich analytics at scale. In SIGMOD 2013, pages 13–24, New York, NY, USA, 2013. ACM.

[33] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauley, M. J. Franklin, S. Shenker, and I. Stoica. Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing. In *Proceedings of the 9th USENIX conference on Networked Systems Design and Implementation*, pages 2–2. USENIX Association, 2012.