The VLDB Journal (2018) 27:595–615
https://doi.org/10.1007/s00778-017-0474-5

SPECIAL ISSUE PAPER

Adding data provenance support to Apache Spark

Matteo Interlandi[1] · Ari Ekmekji[3] · Kshitij Shah[2] · Muhammad Ali Gulzar[2] · Sai Deep Tetali[2] · Miryung Kim[2] · Todd Millstein[2] · Tyson Condie[2]

Received: 15 January 2017 / Revised: 13 May 2017 / Accepted: 24 July 2017 / Published online: 7 August 2017

Abstract Debugging data processing logic in data-intensive scalable computing (DISC) systems is a difficult and time-consuming effort. Today's DISC systems offer very little tooling for debugging programs, and as a result, programmers spend countless hours collecting evidence (e.g., from log files) and performing trial-and-error debugging. To aid this effort, we built *Titian*, a library that enables *data provenance*—tracking data through transformations—in Apache Spark. Data scientists using the Titian Spark extension will be able to quickly identify the input data at the root cause of a potential bug or outlier result. Titian is built directly into the Spark platform and offers data provenance support at interactive speeds—orders of magnitude faster than alternative solutions—while minimally impacting Spark job performance; observed overheads for capturing data lineage rarely exceed 30% above the baseline job execution time.

Keywords Data provenance · Spark · Debugging

1 Introduction

Data-intensive scalable computing (DISC) systems, like Apache Hadoop [23] and Apache Spark [39], are being used to analyze massive quantities of data. These DISC systems expose a programming model for authoring data processing logic, which is compiled into a directed acyclic graph (DAG) of data-parallel operators. The root DAG operators consume data from an input source (e.g., Amazon S3 or HDFS), while downstream operators consume the intermediate outputs from DAG predecessors. Scaling to large datasets is handled by partitioning the data and assigning tasks that execute the "transformation" operator logic on each partition.

Debugging data processing logic in DISC environments can be daunting. A recurring debugging pattern is to identify the subset of data leading to outlier results, crashes, or exceptions. Another common pattern is trial-and-error debugging, where developers *selectively replay* a portion of their data processing steps on a subset of intermediate data leading to outlier or erroneous results. These features motivate the need for capturing *data provenance* (also referred to as *data lineage*) and supporting appropriate provenance query capabilities in DISC systems. Such support would enable the identification of the input data leading to a failure, crash, exception, or outlier results. Our goal is to provide interactive data provenance support that integrates with the DISC programming model and enables the above debugging scenarios.

Current approaches supporting data lineage in DISC systems (specifically RAMP [25] and Newt [31]) do not meet our goals due to the following limitations: (1) they use external storage such as a shared DBMS or distributed file systems (e.g., HDFS) to retain lineage information; (2) data provenance queries are supported in a separate programming interface; and (3) they provide very little support for viewing intermediate data or replaying (possibly alternative) data processing steps on intermediate data. These limitations prevent support for interactive debugging sessions. Moreover, we show that these approaches do not operate well at scale, because they store the data lineage externally.

In this paper, we introduce *Titian*, a library that enables interactive data provenance in Apache Spark. Titian inte-

Matteo Interlandi was formally affiliated to University of California, Los Angeles, Los Angeles, CA, USA when the work was done.

✉ Matteo Interlandi
mainterl@microsoft.com

1 Microsoft, Redmond, WA, USA

2 University of California, Los Angeles, Los Angeles, CA, USA

3 Stanford University, Stanford, CA, USA

grates with the Spark programming interface, which is based on a resilient distributed dataset (RDD) [44] abstraction defining a set of transformations and actions that process datasets. Additionally, data from a sequence of transformations, leading to an RDD, can be cached in memory. Spark maintains program transformation lineage so that it can reconstruct lost RDD partitions in the case of a failure.

Titian enhances the RDD abstraction with fine-grained data provenance capabilities. From any given RDD, a Spark programmer can obtain a LineageRDD reference, which enables *data tracing functionality*, i.e., the ability to transition backward (or forward) in the Spark program dataflow. From a given LineageRDD reference, corresponding to a position in the program's execution, any native RDD transformation can be called, returning a new RDD that will execute the transformation on the subset of data referenced by the LineageRDD. As we will show, this facilitates the ability to trace backward (or forward) in the dataflow, and from there execute a new series of native RDD transformations on the reference data. The tracing support provided by LineageRDD integrates with Spark's internal batch operators and fault tolerance mechanisms. As a result, Titian can be used in a Spark terminal session, providing interactive data provenance support along with native Spark ad hoc queries.

Debugger applications Titian functionalities can be used stand-alone or in conjunction with other high-level tools. For example, we have recently leveraged Titian for supporting debugging features in BIGDEBUG [21,22]: an Apache Spark debugging toolkit. Below, we highlight two features that build on Titian:

– *Crash culprit determination* In Spark, crashes caused by code errors result in program termination, even when the problem remedy is trivial. BIGDEBUG leverages Titian to identify the input record data that causes a crash. The culprit records are packed together with other run-time information and sent to the driver so that the programmer can examine them and possibly take remediation action, such as revise the code, or ignore the crash and resume the job.
– *Automated fault localization* Errors in code or data are hard to diagnose in platforms like Apache Spark. We explore the feasibility of standard debugging techniques such as *delta debugging* [45] for automated fault localization in Spark programs. The delta debugging technique takes the original failure-inducing record set and divides it into several record subsets using a binary search-like strategy. The program is re-executed on the faulty record subsets, which may again induce a failure. The procedure is recursively applied until a minimum failure-inducing subset is identified. Applying the delta debugging technique to Spark workloads is extremely expensive. However, preliminary results show that Titian is able to improve the performance of program re-execution by narrowing the search strategy using lineage [19,20].

Contributions To summarize, Titian offers the following contributions:

– A data lineage capture and query support system in Apache Spark.
– A lineage capturing design that minimizes the overhead on the target Spark program— most experiments exhibit an overhead of less than 30%.
– We show that our approach scales to large datasets with less overhead compared to prior work [25,31].
– Interactive data provenance query support that extends the familiar Spark RDD programming model.
– An evaluation of Titian that includes a variety of design alternatives for capturing and tracing data lineage.
– A set of optimizations that reduce lineage query time by more than an order of magnitude.

In this paper, we present an improved version of the original Titian system [28] that can reduce the lineage query time by an order of magnitude through two optimization techniques: (1) a custom Spark scheduler that can prune tasks that do not contribute to trace query results (Sect. 6.2) and (2) an index file for storing, and efficiently retrieving, data lineage in the Spark BlockManager (Sect. 6.3). Finally, we have included additional experiments showcasing the performance of Titian in forward tracing scenarios (Sect. 6.4.2).

Outline The remainder of the paper is organized as follows. Section 2 contains a brief overview of Spark and discusses our experience with using alternative data provenance libraries with Spark. Section 3 defines the Titian programming interface. Section 4 describes Titian provenance capturing model and its implementation. The experimental evaluation of Titian is presented in Sect. 5. Optimizations and related performance improvements are introduced in Sect. 6. Related work is covered in Sect. 7. Section 8 concludes with future directions in the DISC debugging space.

2 Background

This section provides a brief background on Apache Spark, which we have instrumented with data provenance capabilities (Sect. 3). We also review RAMP [25] and Newt [31], which are toolkits for capturing data lineage and supporting off-line data provenance queries for Hadoop and AsterixDB [8] workloads. Our initial work in this area leveraged these two toolkits for data provenance support in Spark. During this exercise, we encountered a number of issues, including scalability (the sheer amount of lineage data that could be captured and used for tracing), job overhead (the per-job slowdown incurred from data lineage capture), and usability (both provide limited support for data provenance queries). RAMP and Newt operate externally to the target DISC system, making them more general than Titian. However, this prevents a unified programming environment, in which both data analysis and provenance queries can be co-developed, optimized, and executed in the same (Spark) run-time. Moreover, Spark programmers are accustomed to an interactive development environment (e.g., notebook or spark shell), which we want to support.

2.1 Apache Spark

Spark is a DISC system that exposes a programming model based on resilient distributed datasets (RDDs) [44]. The RDD abstraction provides *transformations* (e.g., map, reduce, filter, group-by, join) and *actions* (e.g., count, collect) that operate on datasets partitioned over a cluster of nodes. 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 Spark "driver" program, which could then trigger another series of RDD transformations. The RDD programming interface can support these data analysis transformations and actions through an *interactive* terminal, which comes packaged with Spark.

Spark *driver programs* run at a central location and operate on RDDs through references. A driver program could be a user operating through the Spark terminal, or it could be a stand-alone Scala program. In either case, RDD references lazily evaluate transformations by returning a new RDD reference that is specific to the transformation operation on the target input RDD(s). Actions trigger the evaluation of an RDD reference, and all RDD transformations leading up to it. Internally, Spark translates a series of RDD transformations into a DAG of *stages*, where each stage contains some sub-series of transformations until a *shuffle step* is required (i.e., data must be re-partitioned). The Spark scheduler is responsible for executing each stage in topological order according to data dependencies. Stage execution is carried out by *tasks* that perform the work (i.e., transformations) of a stage on each input partition. Each stage is fully executed before downstream-dependent stages are scheduled, i.e., Spark batch executes the stage DAG. The final output stage evaluates the action that triggered the execution. The action result values are collected from each task and returned to the driver program, which can initiate another series of transformations ending with an action. Next, we illustrate the Spark programming model with a running example used throughout the paper.

Running example Assume we have a large log file stored in a distributed file system such as HDFS. The Spark program in Fig. 1 selects all lines containing errors, counts the number

```
1  lines = sc.textFile("hdfs://...")
2  errors = lines.filter(_.startsWith("ERROR"))
3  codes = errors.map(_.split("\t")(1))
4  pairs = codes.map(word => (word, 1))
5  counts = pairs.reduceByKey(_ + _)
6  reports = counts.map(kv => (dscr(kv._1),
       kv._2))
7  reports.collect.foreach(println)
```

Fig. 1 Running example: log analysis ("val" declarations omitted for brevity)

of error occurrences grouped by the error code, and returns a report containing the description of each error, together with its count.

The first line loads the content of the log file from HDFS and assigns the result RDD to the lines reference. It then applies a `filter` transformation on lines and assigns the result RDD to the errors reference, which retains lines with errors.[1] The transformations in lines 3 and 4 are used to (1) extract an error code, and (2) pair each error code with the value one, i.e., the initial error code count. The `reduceByKey` transformation sums up the counts for each error code, which is then mapped into a textual error description referenced by reports in line 6.[2] The `collect` action triggers the evaluation of the reports RDD reference, and all transformations leading up to it. The `collect` action result is returned to the driver program, which prints each result record.

Figure 2 schematically represents a toy Spark cluster executing this example on a log file stored in three HDFS partitions. The top of the figure illustrates the stage DAG internally constructed by Spark. The first stage contains the lines, errors, codes, and pairs reference transformations. The second stage contains (1) the counts reference produced by the `reduceByKey` transformation, which groups the records by error code in a shuffle step, and (2) the final `map` transformation that generates the reports reference.

The `collect` action triggers the execution of these two stages in the Spark driver, which is responsible for instantiating the tasks that execute the stage DAG. In the specific case of Fig. 2, three tasks are used to execute stage 1 on the input HDFS partitions. The output of stage 1 is shuffled into two partitions of records grouped by error code, which is also naturally the partitioning key. Spark then schedules a task on each of the two shuffled partitions to execute the `reduceByKey` transformation (i.e., sum) in stage 2, ending with the final `map` transformation, followed by the `collect` action result, which is sent back to the driver.

[1] Note that the underscore (_) character is used to indicate a closure argument in Scala, which in this case is the individual lines of the log file.

[2] `dscr` is a hash table mapping error codes to the related textual description.

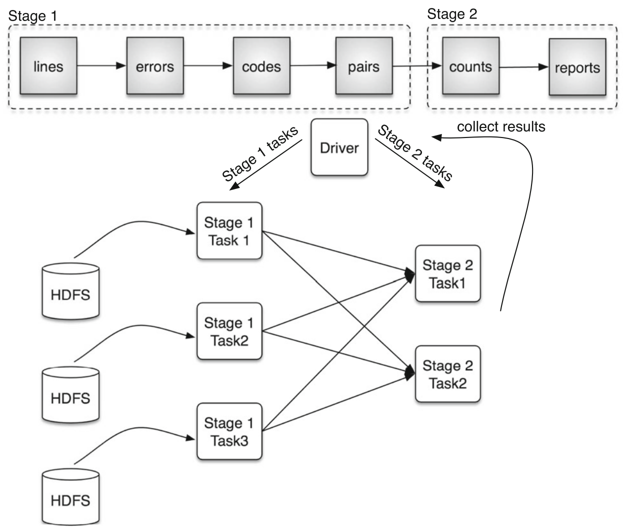
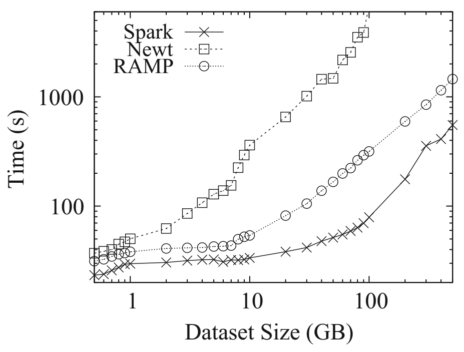


Fig. 2 Example Spark cluster running a job instance executing tasks that run the stage logic on input partitions loaded from HDFS. Data are shuffled between stages 1 and 2. Final results are collected into the driver



dataset size	RAMP	Newt
1GB	1.28×	1.69×
10GB	1.6×	10.75×
100GB	4×	86×
500GB	2.6×	inf

Fig. 3 Run-time of Newt and RAMP data lineage capture in a Spark word-count job. The table summarizes the plot results at four dataset sizes and indicates the run-time as a multiplier of the native Spark job execution time

2.2 Data provenance in DISC

RAMP [25] and Newt [31] address the problem of supporting data provenance in DISC systems through an external, generic library. For example, RAMP instruments Hadoop with "agents" that wrap the user provided `map` and `reduce` functions with lineage capture capabilities. RAMP agents store data lineage in HDFS, where toolkits like Hive [40] and Pig [36] can be leveraged for data provenance queries. Newt is a system for capturing data lineage specifically designed to "discover and resolve computational errors." Like RAMP, Newt also injects agents into the dataflow to capture and store data lineage; however, instead of HDFS, Newt stores the captured data lineage in a cluster of MySQL instances running alongside the target system. Data provenance queries are supported in Newt by directly querying (via SQL) the data lineage stored in the MySQL cluster.

Tracing through the evolution of record data in a DISC dataflow is a common data provenance query. Newt also references support for *replaying* the program execution on a subset of input data that generated a given result, e.g., an outlier or erroneous value. Naturally, this first requires the ability to *trace* to the input data leading to a final result. We use the term trace to refer to the process of identifying the input data that produced a given output data record set. In the context of Spark, this means associating each output record of a transformation (or stage, in Titian's case) with the corresponding input record. Tracing can then be supported by recursing through these input to output record associations to a desired point in the dataflow.

2.3 Newt and RAMP instrumentation

Our first attempt at supporting data provenance in Spark leveraged Newt to capture data lineage at *stage boundaries*. However, we ran into some issues. To leverage Newt, we first had to establish and manage a MySQL cluster alongside our Spark cluster. Second, we ran into scalability issues: as the size of data lineage (record associations) grew large, the Spark job response time increased by orders of magnitude. Third, tracing support was limited: for instance, performing a trace in Newt required us to submit SQL queries in an iterative loop. This was done outside of Spark's interactive terminal session in a Python script. Lastly, both Newt and RAMP do not store the referenced raw data,[3] preventing us from viewing any intermediate data leading to a particular output result. It was also unclear to us, based on this issue, how "replay" on intermediate data was supported in Newt.

Based on the reference Newt documentation, our instrumentation wraps Spark stages with Newt agents that capture data lineage. Newt agents create *unique identifiers* for individual data records and maintain *references* that associate output record identifiers with the relevant input record identifiers. The identifiers and associations form the data lineage, which Newt agents store in MySQL tables.

Figure 3 gives a quantitative assessment of the additional time needed to execute a word-count job when capturing lineage with Newt. The results also include a version of the RAMP design that we built in the Titian framework. For this experiment, only RAMP is able to complete the workload in all cases, incurring a fairly reasonable amount of overhead,

[3] Because doing so would be prohibitively expensive.

```
abstract class LineageRDD[T] extends RDD[T] {
  // Full trace backward
  def goBackAll(): LineageRDD
  // Full trace forward
  def goNextAll: LineageRDD
  // One step backward
  def goBack(): LineageRDD
  // One step forward
  def goNext(): LineageRDD

  @Override
  /* Introspects Spark dataflow
   * for lineage capture */
  def compute(split: Partition,
          context: TaskContext): Iterator[T]
}
```

Fig. 4 LineageRDD methods for traversing through the data lineage in both backward and forward directions. The native Spark `compute` method is used to plug a LineageRDD instance into the Spark dataflow (described in Sect. 4)

i.e., RAMP takes on average 2.3× longer than the Spark baseline execution time. However, the overhead observed in Newt is considerably worse (up to 86× the Spark baseline), preventing the ability to operate on 500 GB. Simply put, MySQL could not sustain the data lineage throughput observed in this job. Section 5 offers further analysis.

3 Data provenance in Spark

Titian is a library that supports data provenance in Spark through a simple LineageRDD application programming interface, which extends the familiar RDD abstraction with tracing capabilities. This section describes the extensions provided by LineageRDD along with some example provenance queries that use those extensions in concert with native RDD transformations (e.g., `filter`). Since our design integrates with the Spark programming model and run-time, Titian extensions can be used naturally in interactive Spark sessions for exploratory data analysis.

Titian extends the native Spark RDD interface with a method (`getLineage`) that returns a LineageRDD, representing the starting point of a trace; the data referenced by the native Spark RDD represent the trace origin. From there, LineageRDD supports methods that transition through the transformation dataflow at stage boundaries.

Figure 4 lists the transformations that LineageRDD supports. The `goBackAll` and `goNextAll` methods can be used to compute the full trace backward and forward, respectively. That is, given some result record(s), `goBackAll` returns all initial input records that contributed to the result record(s); conversely, `goNextAll` returns all the final result records that the starting input record(s) contributed to. A single step backward or forward—at the stage boundary level—is supported by the `goBack` and `goNext`, respectively.

```
frequentPair = reports.sortBy(_._2, false).take(1)
frequent = reports.filter(_ == frequentPair)
lineage = frequent.getLineage()
input = lineage.goBackAll()
input.collect().foreach(println)
```

Fig. 5 Input lines with the most frequent error

These tracing methods behave similarly to a native RDD transformation, in that they return a new LineageRDD corresponding to the trace point, without actually evaluating the trace. The actual tracing occurs when a native Spark RDD action, such as `count` or `collect`, is called, following the lazy evaluation semantics of Spark. For instance, if a user wants to trace back to an intermediate point and view the data, then she could execute a series of `goBack` transformations followed by a native Spark `collect` action. The `compute` method, defined in the native RDD class, is used to introspect the stage dataflow for data lineage capture.

Since LineageRDD extends the native Spark RDD interface, it also includes all native transformations. Calling a native transformation on a LineageRDD returns a new native RDD that references the (base) data at the given trace point and the desired native transformation that will process it. This mode of operation forms the basis of our replay support. Users can trace back from a given RDD to an intermediate point and then leverage native RDD transformations to reprocess the referenced data. We highlight these Titian extensions in the following example.

Example 1 Backward tracing Titian is enabled by wrapping the native `SparkContext` (`sc` in line 1 of Fig. 1) with a `LineageContext`. Figure 5 shows a code fragment that takes the result of our running example in Fig. 1 and selects the most frequent error (via native Spark `sortBy` and `take` operations), and then traces back to the corresponding input lines and prints them.

Next, we describe an example of forward tracing from input records to records in the final result that the input records influenced.

Example 2 Forward tracing Here, we are interested in the error codes generated from the network sub-system, indicated in the log by a “`NETWORK`” tag.
Again, assume the program in Fig. 1 has finished executing. Figure 6 selects log entries related to the network layer (line 1) and then performs a `goNextAll` (line 3) on the corresponding LineageRDD reference (obtained in line 2). Finally, the relevant output containing the error descriptions and counts is printed (line 4).

```
network = lines.filter(_.contains("NETWORK"))
lineage = network.getLineage()
output = lineage.goNextAll()
output.collect().foreach(println)
```

Fig. 6 Network-related error codes

```
lineage = reports.getLineage()
inputLines = lineage.goBackAll()
noGuest = inputLines.filter(!_.contains("Guest"))
newCodes = noGuest.map(_.split("\t")(1))
newPairs = codes.map(word => (word, 1))
newCounts = pairs.reduceByKey(_ + _)
newRep = newCounts.map(kv => (dscr(kv._1), kv._2))
newRep.collect().foreach(println)
```

Fig. 7 Error codes without "Guest"

Provenance queries and actual computation can be interleaved in a natural way, extending the possibilities to interactively explore and debug Spark programs and the data that they operate on.

Example 3 Selective replay Assume that after computing the error counts, we traced backward and notice that many errors were generated by the "Guest" user. We are then interested in seeing the errors distribution without the ones caused by "Guest." This can be specified by tracing back to the input, filtering out "Guest," and then re-executing the computation, as shown in Fig. 7. This is made possible because native RDD transformations can be called from a LineageRDD. Supporting this requires Titian to automatically retrieve the raw data referenced by the lineage and apply the native transformations to it. In additional follow-on work [27], we explore ways of reusing the computation performed during prior executions to speed-up subsequent executions like the one shown in Fig. 7.

4 Titian internal library

Titian is our extension to Spark that enables interactive data provenance on RDD transformations. This section describes the agents used to introspect Spark RDD transformations to capture and store data lineage. We also discuss how we leverage native RDD transformations to support traces through the data provenance via the LineageRDD interface.

4.1 Overview

Similar to other approaches [25,31,37], Titian uses agents to introspect the Spark stage DAG to capture data lineage. The primary responsibilities of these agents are to (1) generate unique identifiers for each new record, and (2) associate output records of a given operation (i.e., stage, shuffle step) with relevant input records.

From a logical perspective, Titian generates new data lineage records in three places:

1. *Input* Data imported from some external source, e.g., HDFS, Java collection.
2. *Stage* The output of a stage executed by a task.
3. *Aggregate* In an aggregation operation, i.e., combiner, group-by, reduce, distinct, and join.

Recall that each stage executes a series of RDD transformations until a shuffle step is required. Stage input records could come from an external data source (e.g., HDFS) or from the result of a shuffle step. Input agents generate and attach a unique identifier to each input record. Aggregate agents generate unique identifiers for each output record and relate an output record to all input records in the aggregation operation, i.e., combiner, reduce, group-by, and join. A Spark stage processes a single input record at a time and produces zero or more output records. Stage agents attach a unique identifier to each output record of a stage and associates it with the relevant input record identifier.

Associations are stored in a table in the local Spark storage layer (i.e., BlockManager). The schema of the table defines two columns containing the (1) input record identifiers and (2) output record identifiers. Tracing occurs by recursively joining these tables.

Remark Titian captures data lineage at the stage boundaries, but this does not prevent tracing to an RDD transformation within a stage. Such a feature could be supported by tracing back to the stage input and re-running the stage transformations, on the referenced intermediate data, up to the RDD transformation of interest. Alternatively, we could surface an API that would allow the user to mark RDD transformation as a desirable trace point. Stage agents could then be injected at these markers.

4.2 Capturing agents

Titian instruments a Spark dataflow with agents on the driver, i.e., where the main program executes. Recall that when the main program encounters an action, Spark translates the series of transformations, leading to the action, into a DAG of stages. The `LineageContext` hijacks this step and supplements the stage DAG with capture agents, before it is submitted to the task scheduler for execution.

Table 1 lists the agents that Titian uses at each capture point, i.e., input, stage, and aggregate. We have defined two input agents. Both assign identifiers to records emitted from a data source. The identifier should be meaningful to the given data source. For instance, the HadoopLineageRDD assigns an identifier that indicates the HDFS partition and

Table 1 Lineage capturing points and agents

CapturePoint	LineageRDDAgent
Input	HadoopLineageRDD
	ParallelLineageRDD
Stage	StageLineageRDD
Aggregate	ReducerLineageRDD
	JoinLineageRDD
	CombinerLineageRDD

record position (e.g., line offset) within the partition. The ParallelLineageRDD assigns identifiers to records based on their location (index) in a Java Collection Object, e.g., java.util.ArrayList.

A Spark stage consists of a series of transformations that process a single record at a time and emit zero or more records. At the stage output, Titian will inject a CombinerLineageRDD when a combiner operation is present, or a StageLineageRDD when a combiner is not present. Both agents are responsible for relating an output record to the producer input record(s). In the case of a StageLineageRDD, for each output record produced, it generates an identifier and associates that identifier with the (single) input record identifier. A combiner pre-aggregates one or more input records and generates a new combined output record. For this case, Titian injects a CombinerLineageRDD, which is responsible for generating an identifier for the combined output record, and associating that identifier with the identifiers of all related inputs.

The other two aggregate capture agents introspect the Spark dataflow in the shuffle step used to execute reduce, group-by, and join transformations. Similar to the combiner, these transformations take one or more input records and produce a new output record; unlike the combiner, join operations could produce more than one output record. The reduce and group-by transformations operate on a single dataset (i.e., RDD), while join operates on multiple datasets. The ReducerLineageRDD handles the reduce and group-by aggregates, while JoinLineageRDD handles the join operation. The ReducerLineageRDD associates an output record identifier with all input record identifiers that form the aggregate group. JoinLineageRDD associates an output (join) record identifier with the record identifiers on each side of the join inputs.

Remark Joins in Spark behave similar to those in Pig [36] and Hive [40]: records that satisfy the join are grouped and co-located together based on the "join key." For this reason, we have categorized JoinLineageRDD as an aggregate, even though this is not the case logically.

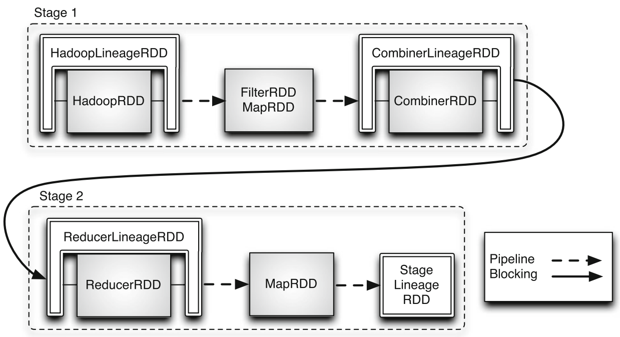


Fig. 8 Job workflow of the running example after adding the lineage capture points

Example 4 Dataflow instrumentation Returning to our running example, Fig. 8 shows the workflow after the instrumentation of capture agents. The Spark stage DAG consists of two stages separated by the `reduceByKey` transformation. The arrows indicate how records are passed through the dataflow. A dashed arrow means that records are pipelined (i.e., processed one at a time) by RDD transformations, e.g., FilterRDD, and MapRDD in stage 1. A solid arrow indicates a blocking boundary, where input records are first materialized (i.e., drained) before the given operation begins its processing step, e.g., Spark's combiner materializes all input records into an internal hash table, which is then used to combine records in the same hash bucket.

The transformed stage DAG includes (1) a HadoopLineageRDD agent that introspects Spark's native HadoopRDD, assigns an identifier to each record, and associates the record identifier to the record position in the HDFS partition; (2) a CombinerLineageRDD that assigns an identifier to each record emitted from the combiner (pre-aggregation) operation and associates it with the (combiner input) record identifiers assigned by HadoopLineageRDD; (3) a ReducerLineageRDD that assigns an identifier to each `reduceByKey` output record and associates it with each record identifier in the input group aggregate; and finally (4) a StageLineageRDD that assigns an identifier to each stage record output and relates that identifier back to the respective `reduceByKey` output record identifier assigned by ReducerLineageRDD.

4.3 Lineage capturing

Data lineage capture begins in the agent `compute` method implementation, defined in the native Spark RDD class, and overridden in the LineageRDD class. The arguments to this method (Fig. 4) include the input partition—containing the stage input records—and a task context. The return value is an iterator over the resulting output records. Each agent's `compute` method passes the partition and task context arguments to its parent RDD(s) `compute` method, and wraps the returned parent iterator(s) in its own iterator module, which it returns to the caller.

Input to Output Identifier Propagation Parent (upstream) agents propagate record identifiers to a child (downstream) agent. Two configurations are possible: (1) propagation inside a single stage and (2) propagation between consecutive stages. In the former case, Titian exploits the fact that stage records are pipelined one at a time through the stage transformations. Before sending a record to the stage transformations, the agent at the input of a stage stores the single input record identifier in the task context. The agent at the stage transformation output associates output record identifiers with the input record identifier stored in the task context. Between consecutive stages, Titian attaches (i.e., piggybacks) the record identifier on each outgoing (shuffled) record. On the other side of the shuffle step, the piggybacked record identifier is grouped with the other record identifiers containing the same key (from the actual record). Each group is assigned a new identifier, which will be associated with all record identifiers in the group.

Next, we describe the implementation of each agent iterator.

- HadoopLineageRDD's parent RDD is a native HadoopRDD. In its iterator, HadoopLineageRDD assigns an identifier to each record returned by the HadoopRDD iterator and associates the identifier with the record position in the (HDFS) partition. Before returning the raw record to the caller, it stores the record identifier in the task context, which a downstream agent uses to associate with its record outputs.
- StageLineageRDD introspects the Spark dataflow on the output of a stage that does not include a combiner. In each call to its iterator, it (1) calls the parent iterator, (2) assigns an identifier to the returned record, (3) associates that identifier with the input record identifier stored in the task context, e.g., by HadoopLineageRDD, and (4) returns the (raw) record to the caller.
- CombinerLineageRDD and ReducerLineageRDD introspect the combiner and reducer (also group-by) transformations, respectively. These transformations are blocking whereby the input iterator is drained into a hash table, using the record key to assign a hash bucket, which form the output group records. The agents introspect this materialization to build a separate internal hash table that associates the record key with the record identifier, before passing the raw record to the native materialization. After draining the input iterator, the native iterator operation begins returning result records, i.e., post-combiner, reducer, or group-by operations. For each result record, the agents assign it an identifier and look up the result record key in the (previously created) internal hash table, which returns the list of input record identifiers that formed the group. The agents can then associate the result record identifier with the list of input record identifiers.
- JoinLineageRDD behaves similarly, except that it operates on the two input iterators that join along a key. Each input iterator returns a record that contains the join key, which the agent uses to build an internal hash table that maps the key to the input record identifier contained in the shuffled record. After draining the input iterator, the native JoinRDD begins to return join results. For each result record, the agent assigns an identifier to it and associates that identifier with the input record ids stored in its internal hash table by the join key.

Discussion The above agents are able to capture lineage for most Spark operations, including `filter`, `map`, `flatMap`, `join`, `reduceByKey`, `groupByKey`, `union`, `distinct`, and `zipPartitions` transformations. Additionally, we are working on efficient support for iterative programs, which can generate large amount of lineage; one could skip capturing lineage for some iterations in order to reduce the total amount of lineage, as done in [10], but the current version of Titian does not support this optimization strategy.

4.4 Lineage storage

Titian stores all data lineage in the BlockManager, which is Spark's internal storage layer for intermediate data. As discussed, agents are responsible for associating the output records of an operation (i.e., stage, combiner, join) with the corresponding inputs. These associations are stored in a BlockManager table, local to the Spark node running the agent. Titian agents batch associations in a local buffer that is flushed to the BlockManager at the end of the operation, i.e., when all input records have been fully processed. We compact the lineage information (i.e., identifiers and associations) into nested format exploiting optimized data structures (such as RoaringBitmaps [9]) when possible. In Sect. 5, we show that Titian maintains a reasonable memory footprint with interactive query performance. If the size of the data lineage grows too large to fit in memory, then Titian asynchronously materializes it to disk using native Spark BlockManager support.

Finally, although Titian is specifically optimized for online (interactive) provenance queries, we also allow users to dump the data lineage information into an external store (e.g., HDFS) for postmortem analysis.

4.5 Querying the lineage data

The lineage captured by the agents is used to trace through the data provenance at stage boundaries. From a logical perspective, tracing is implemented by recursively joining

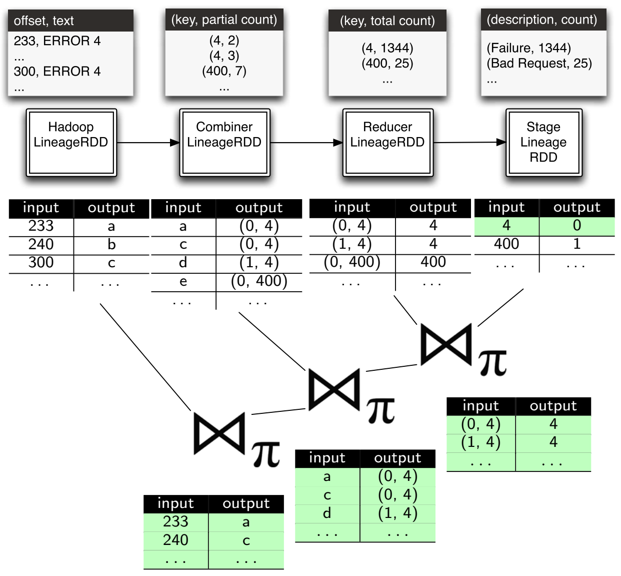


Fig. 9 A logical trace plan that recursively joins data lineage tables, starting from the result with a "Failure" code, back to the input log records containing the error

```
1  failure = reports.filter(_._1 == "Failure")
2  lineage = failure.getLineage()
3  input = lineage.goBackAll()
4  input.collect().foreach(println)
```

Fig. 10 Tracing backwards the "Failure" errors

lineage association tables stored in the BlockManager. A LineageRDD corresponds to a particular position in the trace, referencing some subset of records at that position. Trace positions occur at agent capture points, i.e., stage boundaries. Although our discussion here only shows positions at stage boundaries, we are able to support tracing at the level of individual transformations by simply injecting a StageLineageRDD agent at the output of the target transformation, or by executing the transformations leading up to the target, starting from the relevant stage input records. Next, we describe the logical trace plan for our running example.

Example 5 Logical trace plan Figure 9 is a logical view of the data lineage that each agent captures in our running example from Fig. 2. At the top of the figure, we show some example data corresponding to a log stored in HDFS, along with intermediate data and final results. Recall, in the original running example, we were counting the number of occurrences for each error code. Here, we would like to trace back and see the actual log entries that correspond to a "Failure" (*code* = 4), as shown by the Spark program in Fig. 10.

The output is referenced by the reports RDD reference, which we use to select all "Failure" record outputs and then trace back to the input HDFS log entries. Returning to Fig. 9, the `goBackAll` transformation (Fig. 10 line 3) is implemented by recursively joining the tables that associate the output record identifiers to input record identifiers, until we reach the data lineage at the HadoopLineageRDD agent. Notice that the inter-stage record identifiers (i.e., between CombinerLineageRDD and ReducerLineageRDD) include a partition identifier, which we use in Fig. 11 to optimize the distributed join that occurs during the trace (described below). Figure 9 includes three joins—along with the intermediate join results—used to (recursively) trace back to the HDFS input.

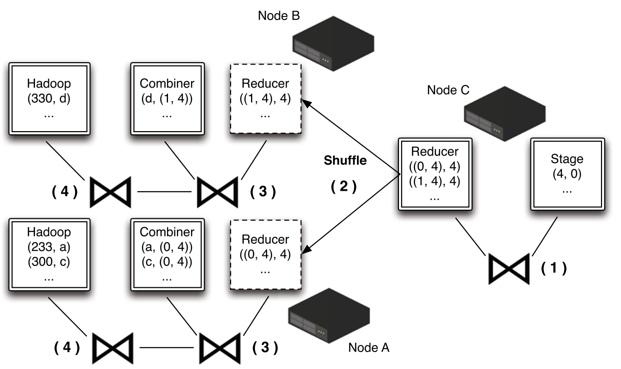


Fig. 11 Distributed tracing plan that recursively joins data lineage tables distributed over three nodes. Each operation is labeled with a number

The logical plan generated by recursively joining lineage points is automatically scheduled in parallel by Spark. Figure 11 shows the distributed version of the previous example. Each agent data lineage is stored across three nodes. The trace begins at the Stage agent on node C, referencing result records with error *code* = 4. Tracing back to the stage input involves a local join (operation 1 in Fig. 11). That join result will contain record identifiers that include a partition identifier, e.g., identifier (0, 4) indicates that an error *code* = 4 occurs in partition 0, which we know to be on node A. We use this information to optimize the shuffle join between the Combiner and Reducer agents. Specifically, the partition identifier is used to direct the shuffling (operation 2) of the lineage records at the Reducer agent to the node that stores the given data partition, e.g., partition $0 \rightarrow A$ and partition $1 \rightarrow B$. The tracing continues on nodes A and B with local joins (operations 3 and 4) that lead to the HDFS partition lines for error *code* = 4.

The *directed shuffle join* described above required modifications to the Spark join implementation. Without this change, the Spark native join would shuffle the data lineage at both the Combiner and Reducer agents (in operation 2), since it has no knowledge of partitioning information. This would further impact the subsequent join with the Hadoop data lineage, which would also be shuffled. In Sect. 5, we show that our directed shuffle join improves the tracing per-

```
failure = reports.filter(_._1 == "Failure")
input = failure.getLineage().goBackAll()
input.filter(_.contains("Zookeeper")
    .collect().foreach(println)
```

Fig. 12 Native transformations on LineageRDDs

formance by an order of magnitude relative to a native join strategy.

4.6 Working with the raw data

LineageRDD references a subset of records produced by a native RDD transformation. When a native RDD transformation is called from a LineageRDD reference, Titian returns a native RDD that will apply the given transformation to the raw data referenced by the LineageRDD.

Figure 12 illustrates this feature by tracing back to log files that contain a "Failure" and then selecting those that are specific to "Zookeeper" before printing.
As before, we start the trace at reports and select the result containing the "Failure" error code. We then trace back to the HDFS log file input, which is referenced by the input LineageRDD. Calling `filter` on the input reference returns a native Spark FilterRDD that executes over the HDFS log file records that contain "Failure" codes,[4] and from that selects the ones containing "Zookeeper" failures. These are then returned to the driver by the `collect` action and printed.

The ability to move seamlessly between lineage data and raw data, in the same Spark session, enables a better interactive user experience. We envision this new Spark capability will open the door to some interesting use cases, e.g., data cleaning and debugging program logic. It also provides elegant support for transformation replay from an intermediate starting point, on an alternative collection of records.

4.7 Discussion

Fault tolerance Our system is completely transparent to the Spark scheduler and in fact does not break the fault tolerance model of Spark. During the capturing phase, data lineage is materialized only when a task has completed its execution. In case of failure, the captured data lineage is not retained, and will be regenerated by the subsequently scheduled task. During the tracing phase, LineageRDDs behave as a normal RDD and follow the native Spark fault tolerance polices.
Alternative designs Debugging systems such as IG [36] and Arthur [12] tag data records with a unique transformation id and piggyback each tag downstream with its related record. This strategy can be easily implemented in Titian: each capturing agent generates the lineage data without storing it; lineage references are instead appended to a list and propagated downstream. The final stage capture point will then store the complete lineage data. In Sect. 5, we will compare this strategy (labeled as Titian-P, for propagation) against the original (distributed) Titian (Titian-D) and a naïve strategy that saves all the lineage into a unique centralized server (Titian-C). In Titian-C, agents write data lineage over an asynchronous channel to a centralized *lineage master*, which stores the data lineage in its local file system. The lineage master executes tracing queries, using a centralized version of LineageRDD, on the local data lineage files. Both Titian-C and Titian-P trade-off space overheads, by aggregating lineage data into a more centralized storage, for a faster tracing time. An interesting area of future work would be an optimizer that is able to estimate the lineage size and select a version of Titian to use (centralized, decentralized, or propagated) based on a size estimate of the data lineage.

Beyond the above alternative designs, we considered configurations in which coarse-grained lineage (e.g., partition-level lineage) is registered, instead of fine-grained (record-level) lineage. This design leads to an interesting trade-off between job performance and provenance query time and is pointed out by an astute reviewer. We initially opted for fine-grained lineage for two reasons. First, we wanted to identify the point in the design space that would lead to an overall minimum provenance query time. Second, coarse-grained lineage would require significant changes to Spark, since record-level lineage would need to be lazily generated during provenance query time. Nevertheless, an exploration of these design alternatives would be an interesting line of future work, which could involve sophisticated scheduling algorithms that mask the effects of lazily generating the relevant fine-grained lineage while executing provenance queries.

5 Experimental evaluation

Our experimental evaluation measures the added overhead in a Spark job caused by data lineage capture and the response time of a trace query. We compare Titian to Newt [31] and RAMP [25]. This required us to first integrate Newt with Spark. We also developed a version of RAMP in Spark using part of the Titian infrastructure. As in the original version, our version of RAMP writes all data lineage to HDFS, where it can be processed off-line. Moreover, in order to show the raw data in a trace, RAMP must also write intermediate data to HDFS. We report the overhead of RAMP when writing only the data lineage. Saving only the data lineage might be

[4] We optimize this HDFS partition scan with a special HadoopRDD that reads records at offsets provided by the data lineage.

relevant when a user simply wants to trace back to the job input, e.g., HDFS input.

5.1 General settings

Datasets and queries We used a mixed set of workloads as suggested by the latest big data benchmarks [41]. All datasets used were generated to a specific target size. Following [25], we generated datasets of sizes ranging from 500 MB to 500 GB seeded by a vocabulary of 8000 terms that were selected from a Zipf distribution. We used these datasets to run two simple Spark jobs: *grep* for a given term and *word-count*. Additionally, we ported eight PigMix "latency queries" (labeled "L#") to Spark for evaluating more complex jobs. These queries are categorized into three groups: aggregate queries (L1, L6, L11), join queries (L2, L3, L5), and nested plans queries (L4, L7). We only report on a representative query from each class based on the worst-case performance. The input data for these queries were created using the PigMix dataset generator, configured to produce dataset sizes ranging from 1 GB to 1 TB.
Hardware and software configurations The experiments were carried out on a cluster containing 16 $i7-4770$ machines, each running at 3.40GHz and equipped with 4 cores (2 hyper-threads per core), 32 GB of RAM and 1 TB of disk capacity. The operating system is a 64bit Ubuntu 12.04. The datasets were all stored in HDFS version 1.0.4 with a replication factor of 3. Titian is built on Spark version 1.2.1, which is the baseline version we used to compare against. Newt is configured to use MySQL version 5.5.41. Jobs were configured to run two tasks per core (1 task per hyper-thread), for a potential total of 120 tasks running concurrently.

We report on experiments using three versions of Titian:

1. *Titian-D* stores data lineage *distributed* in the BlockManager local to the capture agent.
2. In *Titian-P* all agents, other than the last, propagate data lineage downstream. The final agent stores on the local Spark BlockManager the complete lineage of each individual data record as a *nested list*.
3. *Titian-C* agents write the lineage to a centralized server.

We expect Titian-P and Titian-C to be less efficient than Titian-D during the capturing phase, but more effective in the tracing when the lineage data remain relatively small. Titian-D is the best strategy for capture and outperforms Titian-P when tracing very large datasets, while Titian-C hits scalability bottlenecks early for both capture and trace workloads. In both Newt and RAMP, data lineage capture and tracing is distributed. Therefore, in our evaluation, we compare Titian-D to Newt and RAMP and separately compare Titian-D to Titian-P and Titian-C.

5.2 Data lineage capture overheads

Our first set of experiments evaluate the overhead of capturing data lineage in a Spark program. We report the execution time of the different lineage capturing strategies in relation with the native Spark run-time (as a baseline). We executed each experiment ten times, and among the ten runs, we computed the trimmed mean by removing the top two and bottom two results and averaging the remaining six. In Titian(D and P), we store the data lineage using the Spark BlockManager with setting MEMORY_AND_DISK, which spills the data lineage to disk when memory is full.
Grep and word-count Figure 13a reports the time taken to run the grep job on varying dataset sizes. Both axes are in logarithmic scale. Under this workload, Titian and RAMP incur similar overheads, exhibiting a run-time of no more than a 1.35× the baseline Spark. However, Newt incurs a substantial overhead: up to 15× Spark. Further investigation led to the discovery of MySQL being a bottleneck when writing significant amounts of data lineage. Figure 13b compares the three versions of Titian executing the same job. On dataset sizes below 5 GB, the three versions compare similarly. Beyond that, the numbers diverge considerably, with Titian-C not able to finish beyond 20 GB.

Figure 13c reports the execution time for the word-count job. For this workload, Titian-D offers the least amount of overhead w.r.t. normal Spark. More precisely, Titian-D is never more than 1.3× Spark for datasets smaller than 100 GB, and never more than 1.67× at larger dataset sizes. The run-time of RAMP is consistently above 1.3× Spark execution time, and it is 2–4× slower than normal Spark for dataset sizes above 10 GB. Newt is not able to complete the job above 80 GB and is considerably slower than the other systems. Moving to Fig. 13d, Titian-P performance is similar to RAMP, while Titian-C is not able to handle dataset sizes beyond 2 GB.

Table 2 summarizes the time overheads in the grep and word-count jobs for Titian-D, RAMP, and Newt. Interestingly, for the Word-count workload, the run-time of RAMP is 3.2× Spark at 50 GB and decreases to 2.6× at 500 GB. This is due to the overheads associated with capturing lineage being further amortized over the baseline Spark job. Furthermore, we found that job time increases considerably for datasets greater than 100 GB due to (task) scheduling overheads, which also occur in baseline Spark.
Space overhead In general, the size of the lineage increases proportionally with the size of the dataset and is strictly related to the type of program under evaluation. More specifically, we found that the lineage size is usually within 30% of the size of the input dataset (for non-iterative programs), with the exception of word-count on datasets bigger than 90 GB, where the lineage size is on average 50% of the size of the initial dataset. In computing the space overhead, we

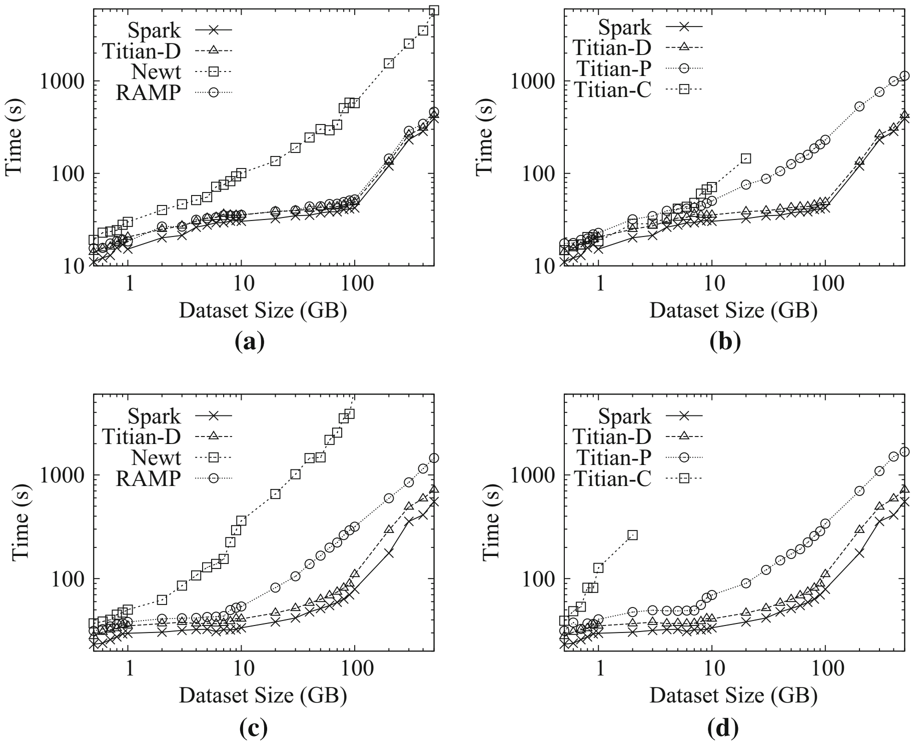


Fig. 13 Lineage capturing performance for grep and Word-count. **a** Grep distributed. **b** Grep Titian versions, **c** Word-count distributed and **d** Word-count Titian versions

Table 2 Run-time of Titian-D, RAMP, and Newt for grep and word-count (wc) jobs as a multiplier of Spark execution time

Dataset	Titian-D		RAMP		Newt	
	Grep	wc	Grep	wc	Grep	wc
500 MB	1.27×	1.18×	1.40×	1.34×	1.58×	1.72×
5 GB	1.18×	1.14×	1.18×	1.32×	1.99×	4×
50 GB	1.14×	1.22×	1.18×	3.2×	8×	29×
500 GB	1.1×	1.29×	1.18×	2.6×	15×	inf

took into account both the size of the actual lineage, and the overhead introduced in the shuffle. For very large datasets (e.g., word-count), the lineage data do not completely fit into memory, causing it to be spilled to disk, which happens asynchronously.

PigMix queries The results for Titian-D on the three PigMix queries are consistently below 1.26× the baseline Spark job. Figure 14a–c shows the running times for queries L2, L6, and L7. We summarize the results for each query below.

L2 We observe that Titian-D exhibits the least amount of overhead for all dataset sizes, with Titian-P adding slightly more (up to 40%) overhead. Titian-C is only able to execute dataset size 1 GB at around 2.5× Spark execution time. RAMP is on par with Titian-D for 1 GB and 10 GB datasets. Its running time degrades at 100 GB (around 2.4× the Spark run-time), and it is not able to complete on the 1 TB dataset. Newt incurs significant overhead throughout and is not able to complete on the 100 GB and 1 TB datasets.

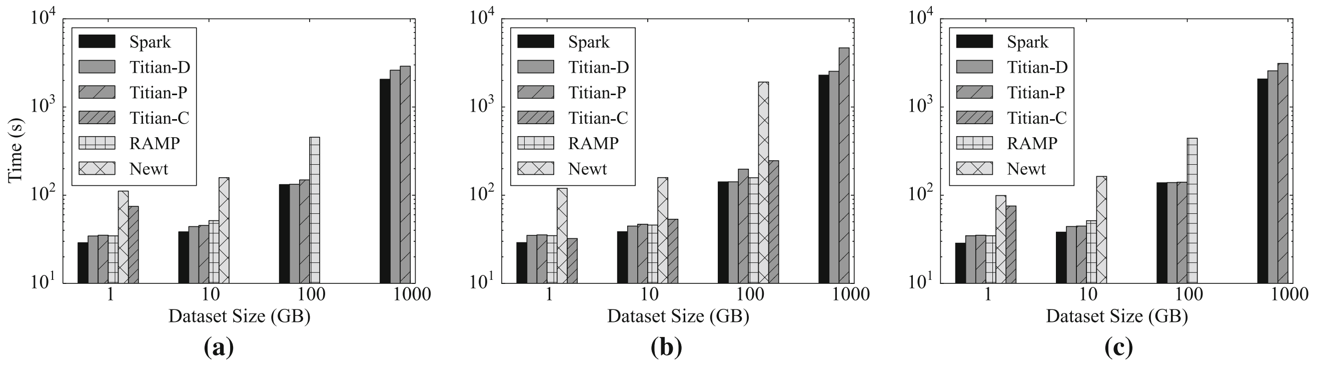


Fig. 14 Lineage capturing performance for Pigmix queries. **a** L2 execution time. **b** L6 execution time. **c** L7 execution time

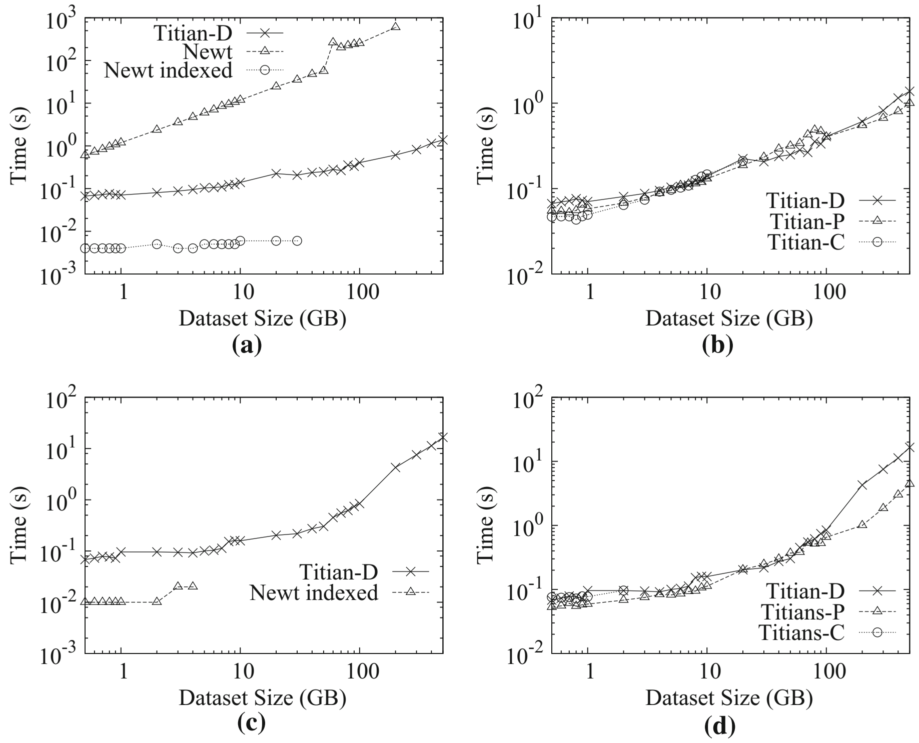


Fig. 15 Tracing time for grep and word-count. **a** Grep distributed, **b** Grep Titian versions. **c** Word-count distributed and **d** Word-count Titian versions

L6 Titian-D and Titian-P are able to complete this query for all dataset sizes. Titian-D is constantly less that 1.2× Spark execution time, while the execution time for Titian-P goes up to 2× in the 1 TB case. Titian-C is able to complete up to the 100 GB dataset with a run-time ranging from 1.1× (at 1 GB) to 1.95× (at 100 GB). RAMP and Newt are able to complete up to the 100 GB case, with an average running time of 1.16X Spark in RAMP, and in Newt we see a 7× slowdown w.r.t. Spark.
L7 The general trend for this query is similar to L2: Titian-D and Titian-P and RAMP have similar performance on the 1 GB and 10 GB datasets. For the 100 GB dataset, RAMP execution time is 2.1× the baseline Spark. Titian-C can only execute the 1 GB dataset with an execution of 2.6× Spark. Newt can only handle the 1 GB and 10 GB datasets at an average time of 3.6× over baseline Spark.

5.3 Tracing

We now turn to the performance of tracing data lineage, starting from a subset of the job result back to the input records. We compare against two configurations in Newt: (1) that indexes the data lineage for optimizing the trace, and (2) no indexes are used. The later configuration is relevant since indexing the data lineage can take a considerable amount of time, e.g., upwards of 10 min to 1 h, depending on the data lineage size.

The experiments described next were conducted as follows. First, we run the capturing job to completion. For Newt, the first step also includes building the index (when applicable). We do not report these times in our plots. Next, we randomly select a subset of result records, and from there trace back to the job input. The trace from result to input is repeated 10 times, on the same selected result, and we report the trimmed mean. We only report on backward traces since the forward direction, from input to output, exhibits similar results (we will, however, report forward tracing performance in Sect. 6 when showing the impact of different optimizations). We do not report the results for tracing queries taking more than 10 min. Also, for the Newt case with indexes, we do not report results when the index building time exceeds 1 h.

Optimal spark join for tracing In Sect. 4.5, we have described our modification to the Spark join operator to leverage the partition identifier information embedded in the record identifier.

This optimization avoided the naïve join strategy, in which the data lineage, stored in the BlockManager, would be fully shuffled in every join. Figure 17 shows the benefits of this optimization when tracing the data lineage from the word-count job. The naïve join is around one order of magnitude slower than the optimal plan up to 80 GB. Beyond that, the naïve performance degrades considerably, with the trace at 200 GB taking approximately 15 min, compared to the optimal 5 s. The naïve join strategy is not able to complete traces above 200 GB.

Trace grep and word-count The time to trace backward one record for grep is depicted in Fig. 15a. In the Newt case,

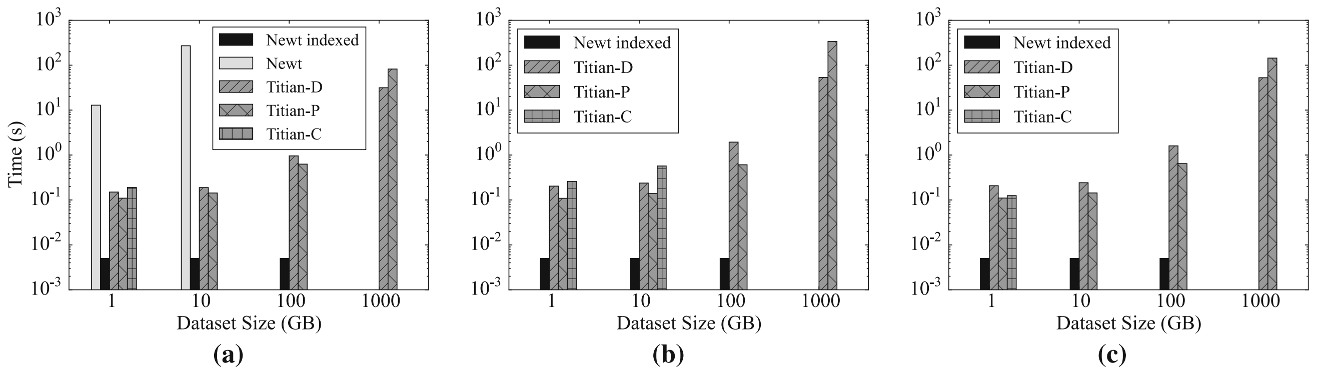


Fig. 16 Lineage tracing performance for PigMix queries. **a** L2 tracing time. **b** L6 tracing time. **c** L7 tracing time

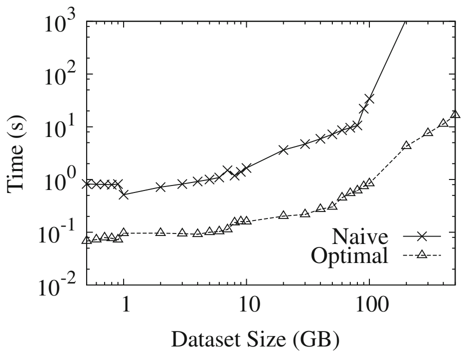


Fig. 17 Naïve versus optimal plan when executing backward tracing for word-count

the query to compute the trace backward is composed of a simple join. Not surprisingly, when relations are indexed the time to compute a full trace is small. When the relations are not indexed, the time to execute the query increases to 10 min. Tracing queries over Titian-D scale linearly from 0.07 s (at 500 MB) to 1.5 s (at 500 GB). Figure 15b compares the three versions of Titian. As expected, Titian-P and Titian-C are slightly faster than Titian-D since the data lineage is more localized and not large. Note that no tracing time is reported for datasets bigger than 20 GB for Titian-C because lineage capturing is not able to complete (Figs. 16, 17).

Figure 15c shows the execution time for retrieving the full lineage of a single record for word-count. Newt without indexes is not able to complete any trace in less than 10 min. When the data lineage is indexed, Newt is able to trace up to the 5 GB dataset, after which the index build time reached 1 h. Titian-D executed the 1 GB dataset trace in 0.08 s, and it took no more than 18 s for larger datasets. In Fig. 15d Titian-D, Titian-P and Titian-C have similar performances for small dataset sizes, while Titian-P outperform Titian-D by a factor of 4 for bigger sizes. As for the grep experiment, Titian-C tracing time is available only up to 2 GB.

PigMix queries Figure 14a, d, e shows the tracing time for PigMix queries L2, L6 and L7, respectively.

L2 We were able to execute the non-indexed Newt trace only for the 1 GB (13 s) and 10 GB datasets (more than 200 s). The indexed version maintains constant performance (5 ms) up to 100 GB, but failed to build the index in less than 1 h for 1 TB dataset. All the Titian versions exhibit similar performance for datasets smaller than 1 TB, executing the trace in a few hundreds of milliseconds (Titian-P has the best results for the 1 GB and 10 GB datasets). For the 1 TB experiment, Titian-D has the best result with 31 s, while Titian-P takes 82 s.

L6 For this aggregate query, Newt is not able to complete any tracing query under the threshold of 600 s. The indexed version still maintains constant performance and up to 100 GB. Titian-C is able to complete both the 1 GB and 10 GB workloads with a tracing time, respectively, of 0.25 and 0.57 s. Titian-D and Titian-P have comparable performance up to 10 GB. For 100 GB, Titian-P performs relatively better than Titian-D (0.6 vs 1.9 s), while for the 1 TB, Titian-D is 6 times faster (53 against 337 s).

L7 The trend for this query follows the one of query L6. For instance, Titian-D takes 52 s for tracing one record over the 1 TB dataset, while Titian-P takes 150 s.

Discussion For small datasets, in general Titian-P performs better than Titian-D. However, as expected, for bigger datasets Titian-P performance decrease considerably. Recall, to minimize the memory footprint, Titian-D and Titian-P save lineage in nested format, i.e., a single top-level identifier is used to reference the set of identifier corresponding to the record group. The un-nesting (dereferencing) of the data lineage is an expensive operation for tracing through aggregate and join operators, especially in the Titian-P case, because data are more centralized, resulting in many more dereference operations per-task.

5.4 Show me the data

Measuring the efficiency of replay includes the performance of tracing, retrieving the data records referenced by the lin-

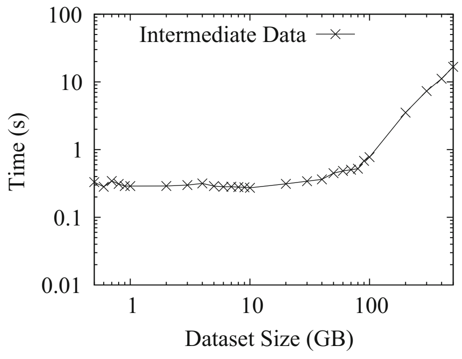


Fig. 18 Data retrieval performance, i.e., the cost of retrieving one intermediate (raw) data record from its lineage identifier

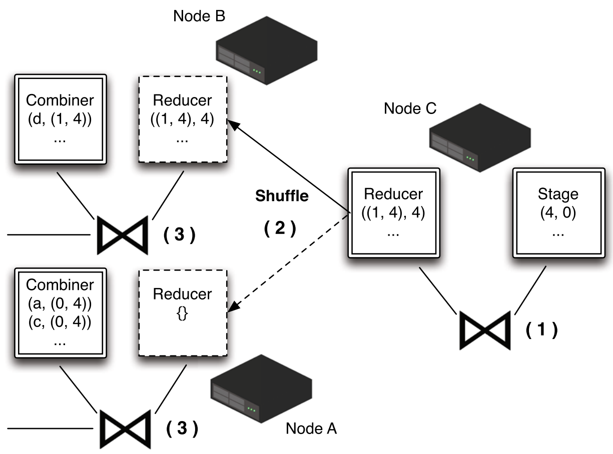


Fig. 19 A modified version of the backward query of Fig. 11. The Spark scheduler is not aware of the records distribution inside partitions; therefore, the join of operation 3 on Node A is scheduled even if does not produce any output

eage, and the cost of re-computation. In this section, we focus on the performance of the second step, i.e., raw data retrieval.

Figure 18 depicts the time of retrieving an intermediate data record from its lineage identifier. This operation involves reading the intermediate data partition, and scanning up to the data record location. As the figure shows, for datasets smaller than 70 GB, this operation takes less than 1 s. The time increases to 27 s for the 500 GB case. This increase in access time is due to the increased size of the intermediate data partitions as we scale up the experiment.

6 Optimizations

This section describes our extensions to the Spark Titian library for improving the performance of lineage queries, i.e., tracing through the data lineage. Over the course of the last year, we observed that programmers are often interested in tracing from a few records that they deem to be of interest, e.g., outliers and crash culprits. As a result, only a small amount of lineage records must be examined at query time. However, the Spark platform only supports a full-scan access method.[5] Therefore, our first optimization extends Spark with an access method for retrieving lineage records more efficiently than native Spark plans.

Additionally, we recognized that for trace queries, the majority of scheduled tasks do not contribute to the result. For instance, consider Fig. 19, which contains a modified version of the backward tracing example of Fig. 11. The query begins at node C where the Stage agent is locally joined with the Reducer agent (operation 1). The output of the join is distributed using the direct shuffle (operation 2), which uses the partition identifier (embedded in the lineage identifier) to direct lineage records to the corresponding Reducer partition, which will be joined with the Combiner partition (operation 3). Note that the output of the join in operation 1 only contains identifiers with partition $id = 1$. As a consequence, no record is shuffled to Node A, yielding an empty output for the join in operation 3 on Node A. Regardless, the Spark native scheduler assigns a task to all partitions, including Node A,[6] because Spark lacks the run-time statistics needed to determine whether a partition join is empty. We found that for large datasets, small tracing queries incur significant overhead from tasks that do not contribute to the trace.

The remainder of this section describes how we addressed the inefficiencies mentioned above. Section 6.1 begins with some background on how native Spark schedules and executes tasks over data partitions. Section 6.2 introduces a new custom Spark scheduler, called *Hyperdrive*, which uses partition statistics to avoid scheduling tasks on partitions that do not contribute to the trace query. In addition, Hyperdrive decouples task execution from partitions and assigns multiple partitions to a single task, yielding similar benefits to reusing threads across multiple HTTP requests in SEDA [42]. Finally, Sect. 6.3 describes an access method that efficiently retrieves the relevant lineage ids in a partition without having to scan all the records.

6.1 Spark internals

A Spark cluster is composed of two node types: a *master* node and a set of *worker* nodes. The master node is responsible for instantiating the driver program and scheduling tasks that

[5] Apache Spark SQL has the ability to direct these full scans to certain data partitions only.

[6] The task assigned to Node A performs a non-trivial amount of work, i.e., it builds a hash table on the Combiner partition that will be probed with the Reducer partition, which, however, will be empty after the directed shuffle.

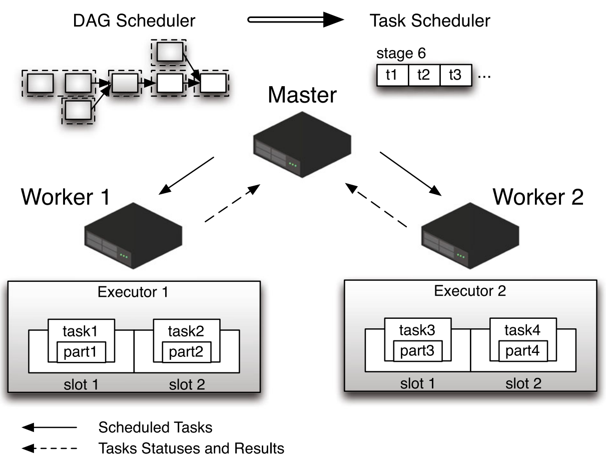


Fig. 20 A representation of the two-level scheduling logic of Spark: the DAGScheduler is taking track of current and already executed stages (i.e., the stages containing gray transformations in the figure); the TaskScheduler launches on worker nodes the set of tasks composing a stage. Worker nodes install tasks on empty slots. Tasks execute a set of transformations on the data partition assigned to them

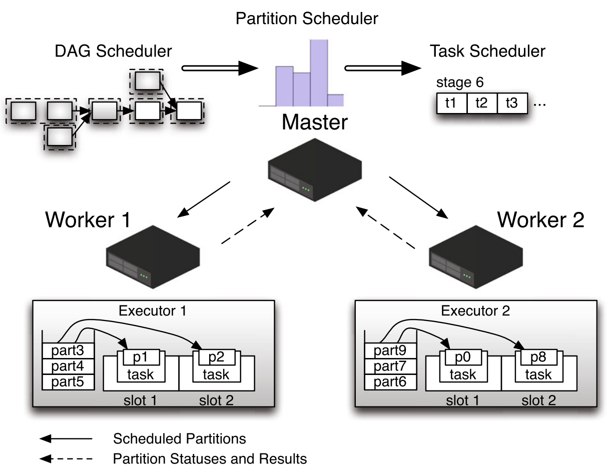


Fig. 21 With Hyperdrive, tasks are now unrelated to a particular partition. Partitions are accumulated into a queue that is concurrently drained by a pool of (persistent) long running tasks. Partition statistics are collected into the master so that better scheduling decisions can be made

execute transformations on data partitions via worker nodes. Each worker node runs an Executor run-time instance that provides the facilities required to spawn and execute tasks on data partitions.

Figure 20 depicts how Spark internally operates on a cluster composed by a master node and two worker nodes. Spark employs a two-level scheduling approach: a top-level DAGScheduler is responsible for (1) translating the RDD transformations defined by the user program into a DAG of stages; (2) batching scheduling stages according to data dependencies; and (3) instantiating tasks to execute each stage in parallel. A task is an execution unit that evaluates the sequence of transformations in a stage on a data partition. A lower-level TaskScheduler assigns tasks to Executors based on the data locality requirements (if available) and Executor resources (i.e., slots).

On worker nodes, an Executor runs Spark tasks on core slots. Each task executes a stage on a provided input data partition. Records composing each data partition are presented to a task in the form of an iterator, i.e., Spark follows the Volcano-style [17] dataflow model. When the partition iterator is drained, the Executor sends the task status, along with any action results, to the master, which forwards the information to the TaskScheduler. When all tasks for a given stage are acknowledged, the DAGScheduler schedules the next stage, until the final program result (i.e., action) is generated.

6.2 Hyperdrive

The Spark scheduler maintains a one-to-one mapping between data partitions and executing tasks. This design simplifies scheduling policies (e.g., for locality and stragglers) and fault handling (e.g., replay the task), but can place significant overheads on the scheduler for stages that contain many lightweight tasks, i.e., tasks that produce little or no output results. This section introduces our *Hyperdrive* scheduler, which uses *persistent tasks*[7] to reduce the scheduling overhead of lightweight tasks by extending the task lifetime over multiple data partitions. Perhaps interestingly, we are able to support persistent tasks without sacrificing fault tolerance or common scheduling policies.

Figure 21 presents the main components of the Hyperdrive scheduler, which includes (1) a *partition queue* on each Executor; (2) support for persistent tasks on Executor slots; and (3) a PartitionScheduler on the master that collects partition statistics and supports partition-based scheduling policies. The partition queue (henceforth, queue) dispatches partitions to persistent tasks (henceforth, task) running on the Executor. When a task finishes processing a given partition, it requests a new partition from the queue. The Executor (worker) coordinates with the PartitionScheduler (master) to fill its queue with stage input partitions. Tasks are decommissioned when the queue is empty, and the PartitionScheduler has no further outstanding stage partitions to assign.

Partition statistics from task outputs are collected by the Executor and sent to the PartitionScheduler during regular heartbeat internals. Presently, we track partition sizes, which we use to detect and avoid the scheduling of empty partitions in tracing queries. The PartitionScheduler also uses the partition size information to make partition assignments to Executor queues. Specifically, (1) we group partitions into batches of similar sizes to improve cluster utilization,

[7] The term persistent is used to indicate that the lifetime of a task spans beyond a single data partition.

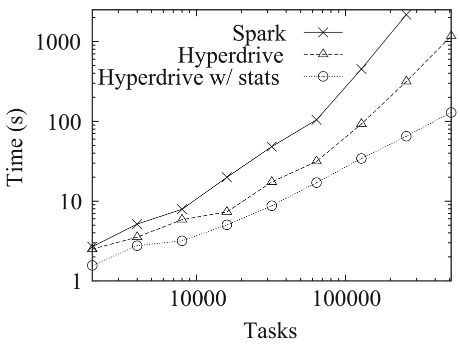


Fig. 22 Comparison between Spark scheduler and Hyperdrive. We report two versions of Hyperdrive: one where empty tasks are scheduled as normal tasks (labeled as "Hyperdrive") and one where empty tasks are identified using partition statistics and not scheduled (labeled as "Hyperdrive w/ stats")

and (2) inside each batch, we order the partitions by size to improve the performance under skew. Empty partitions are pruned from this assignment.

Summary The modifications that Hyperdrive introduces to Spark were motivated by the tracing workloads that we have observed in Titian. Specifically, tracing from a narrow set of results (e.g., crash culprit or outlier) generates many intermediate empty partitions that clearly do not join with corresponding upstream (non-empty) partitions. Regardless, the native Spark scheduler will spawn tasks to execute such joins. Hyperdrive optimizes such workloads through (1) persistent tasks that reduce the overheads associated with lightweight tasks; (2) more aggressive scheduling policies enabled by the decoupling of tasks from partitions. For instance, part of scheduling duties can be offloaded to worker machines by batch-scheduling partitions; and (3) partitions statistics are collected at run-time and used to avoid executing tasks that produce empty results, and for load balancing heavyweight (or skewed) data partitions by scheduling them early.

Figure 22 depicts the performance gains from these two optimizations in a mini-benchmark on a three-stage job, where each stage is executing empty tasks. Notably, Hyperdrive with empty partition scheduling is up to 6× (2170 vs 315 s for 256,000 tasks[8]) faster than the normal Spark scheduler. If we completely avoid the execution of empty partitions by using partition statistics, Hyperdrive becomes 30× faster (2170 vs 65 s, again for 256,000 tasks). Note that native Spark is not able to complete the benchmark for 512,000 tasks.

Further optimizations Hyperdrive offers a few other notable optimizations that we briefly describe.

– When there are many empty partitions, single-threaded components in the native Spark scheduler (i.e., DAGScheduler) can become overwhelmed. In order to alleviate this bottleneck, we made some multi-threaded extensions to these components.
– In an early version of Hyperdrive, we observed starving tasks waiting on partition assignments and data loads. We were able to mitigate this problem through double buffering, often used in external sorting and hashing algorithms.
– We implemented a work stealing mechanism to address situations in which Executor partition queues become unbalanced. However, we found that the required coordination logic introduces a non-negligible overhead, whereby less aggressive batching policies are more effective in avoiding load unbalance. High-speed networks may prove this approach to be more viable.[9]

6.3 Optimizing lineage access

The native Spark record iterator forces each task to perform a linear scan over the full partition. However, we have information (embedded in lineage identifiers) that can be used to directly identify the exact records of interest within a partition. To take advantage of this extra knowledge, we extended Spark with a partition access method that supports filter pushdown and indexing.

Filter pushdown Spark stores cached records in the BlockManager as arrays of record objects. Our lineage identifiers contain the index at which the specific lineage record is stored in this array. To exploit this information, we extended the BlockManager API with a filter pushdown predicate that returns an iterator to the records of interest and avoids the cost of deserializing other records.

Indexing In CombinerLineageRDD agents, lineage records are stored in a nested format because of the grouping operation performed by the combiner. In order to further boost lineage id retrieval in these agents, we have implemented an hash-based index that maps a record key to the BlockManager array indexes corresponding to the lineage records in the group. The filter pushing technique is then used to efficiently retrieve the lineage records in the group.

Discussion The implementation of our filter uses the order in which lineage identifiers are generated during the capture phase. Additionally, record keys are hash-indexed at combiner outputs, i.e., record keys form the index key, which references the corresponding combiner input lineage identifiers. These two design choices make our access method relevant for backward traces only. Supporting forward traces would require additional secondary indexes that would consume a significant amount of space and time overhead to build during lineage capture. Therefore, we abandoned

[8] 256,000 tasks correspond to approximately the scheduling workload generated by a dataset of about 16 TB (using the default Spark settings).

[9] Our experimental cluster was equipped with a 1 Gbps Ethernet.

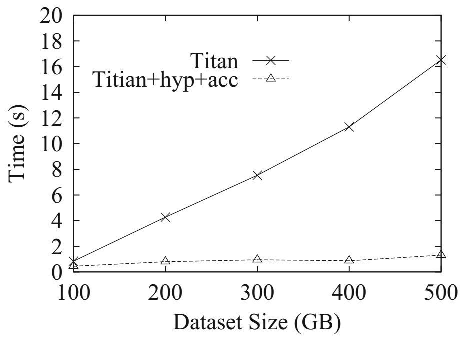


Fig. 23 Backward tracing comparison of Titian with and without optimizations for word-count

this approach since forward traces are less common than backward traces in our applications, e.g., determining failure-inducing inputs during debugging. However, the Hyperdrive scheduler optimization (described in Sect. 6.2) is relevant to both tracing directions, and in fact, in the next section we will see major performance improvements for forward tracing queries as well.

6.4 Evaluation

In this section, we assess the performance of Titian with the optimizations previously described. Our experiments consist of the same queries used in Sect. 5, with the exception of grep which already showed good performance in Sect. 5. We avoid dataset sizes less than 100 GB since tracing time is less than a few seconds. Section 6.4.1 presents experiments for backward tracing where the benefits of both scheduling and access method optimizations can be observed. Section 6.4.2 concludes with forward tracing experiments showing the benefits of Hyperdrive only. The experimental setting is the same as described in Sect. 5: we run the initial query once, followed by a tracing query for one record, executed ten times. We plot the trimmed mean computed over the ten executions.

6.4.1 Backward tracing

Word-count Figure 23 depicts the comparison between Titian with and without optimizations, respectively, labeled as "Titian+hyp+acc" and "Titian." As the figure shows, Titian with optimizations achieves almost a 2× improvement for 100 GB (0.44 vs 0.84 s) and up to a 10× improvement for the 400 GB and 500 GB datasets (0.88 compared to 11.3 s, and 1.31 vs 16.5 s, respectively).

PigMix queries The comparison for queries L2, L6, and L7 is depicted in Fig. 24. For the smaller dataset (100 GB), both approaches perform similarly (only for L6, the optimized version is more than 3× faster, 0.48 compared to 1.93 s). For larger datasets, we observe higher benefits. For L6 and L7, in particular, the optimized Titian is at least 10× faster than the non-optimized version. More specifically, for L7 with 1 TB, optimized Titian takes only 2.9 s to trace one record backward, compared to approximately 52 s for non-optimized Titian. In general, with the new optimizations Titian never takes more than 6 s to trace backward from a single record.

Lineage capture overhead We found that indexing at combiner outputs does not introduce any major overhead during lineage capturing. For example, for the world count scenario, indexing introduces an average overhead of 14% w.r.t. unoptimized (regular) Titian execution. Hyperdrive is enabled only at tracing time; therefore, no additional overhead is introduced at capturing time by scheduler optimizations.

6.4.2 Forward tracing

Word-count In this scenario, the performance of Titian without optimizations are close to the one already discussed in Sect. 5.3. As shown in Fig. 25, Titian with Hyperdriver is uniformly faster, by an average factor of 2×. For instance, at 500 GB Titian takes 15.4 s, compared to Titian with Hyperdrive, which takes 7.5 s. Note that for the same query in the backward case, because of the access method optimizations, tracing a record takes 2.8 s in the worst case, i.e., forward

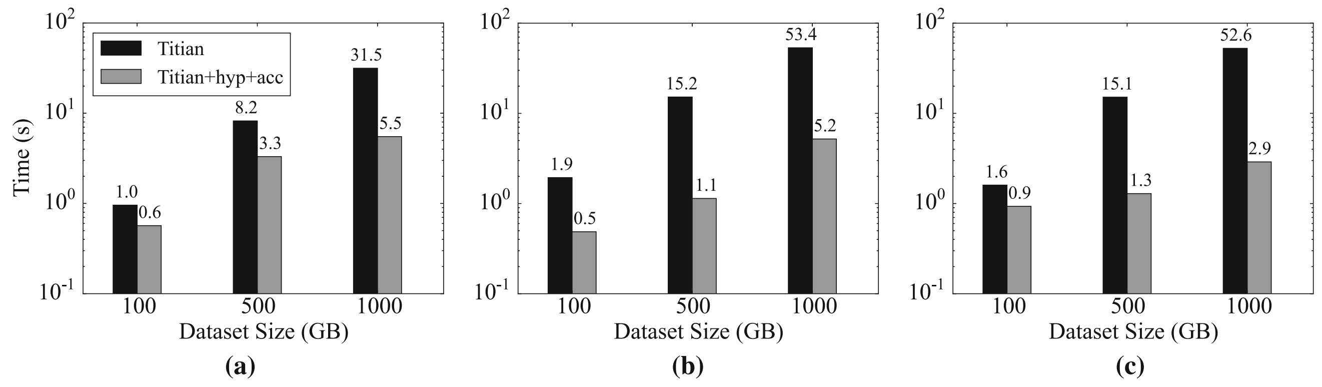


Fig. 24 Backward tracing comparison of Titian with and without optimizations for PigMix queries. **a** L2. **b** L6. **c** L7

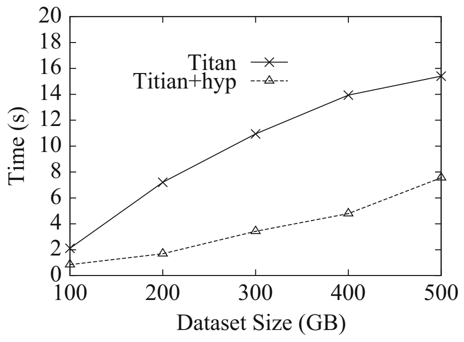


Fig. 25 Forward tracing comparison of Titian with and without Hyperdrive for word-count

tracing is about 2 to 3 times slower than backward tracing for word-count.

PigMix queries Forward tracing for the PigMix queries takes about 2–3 more time than backward tracing for regular Titian. For instance, tracing a record forward for L6 on a 1 TB dataset takes approximately 116 s (Fig. 26b), whereas tracing backward from one record takes 53 s. Similarly, for query L7, tracing forward takes up to 3× more time than tracing backward, as shown in Fig. 26c for a 1 TB dataset: tracing forward from one record takes 119 s, whereas tracing backward from one record takes 52 s. Nevertheless, the Hyperdrive scheduler offers significant improvements to forward traces. For instance, we observed for 1 TB a 15× improvement for L6 (7.2 vs 116 s), and 13× improvement for L7 (9.4 vs 119 s).

7 Related work

There is a large body of work that studies techniques for capturing and querying data lineage in data-oriented workflows [2,3,7,11,24,33]. Data provenance techniques have also been applied in other fields such as fault injection [1], network forensics [46], data integration and exchange [15, 18,29], and distributed network systems analysis [47]. In this paper, we have compared Titian against the approaches relevant to DISC workflows [25,31].

Inspector gadget (IG) [37] defines a general framework for monitoring and debugging Apache Pig programs [36]. IG introspects the Pig workflow with monitoring agents. It does not come with full data lineage support, but rather it provides a framework for tagging records of interest, as they are passed through the workflow. In this setting, programmers can embed code in the monitoring agents that tag records of interest, while IG is responsible for passing those tags through to the final output.

Arthur [12] is a Spark library that can re-execute (part of) the computation to produce the lineage information on demand. Although such an approach introduces zero overhead on the target dataflow (i.e., no data lineage is captured during the initial query execution), it sacrifices the ability to provide interactive tracing queries. Similarly to IG (and to Titian-P), Arthur uses tagging techniques to generate and propagate lineage data when it is requested after the initial execution.

A common pattern for provenance systems is to use different languages for querying the data and querying the lineage [3,30]. In our system, we instead provide a lightweight extension of the transformations already provided by Spark. In this way, we are able to provide users with a uniform language for data and lineage analysis. To our knowledge, only Glavic et al. [14] provided a similar feature, in the context of relational database systems. This capability allowed us to build on top of Titian a set of debugging features [19,21,22] that developers can natively use to analyze their Big Data applications.

In the database domain, several systems have recently started to address the interesting problem of explaining anomalous results by devising lineage fragments having some sort of "influence" on the outlier result. Meliou et al. pioneered this research area by introducing the concept of degree of responsibility to identify tuples, seen as potential

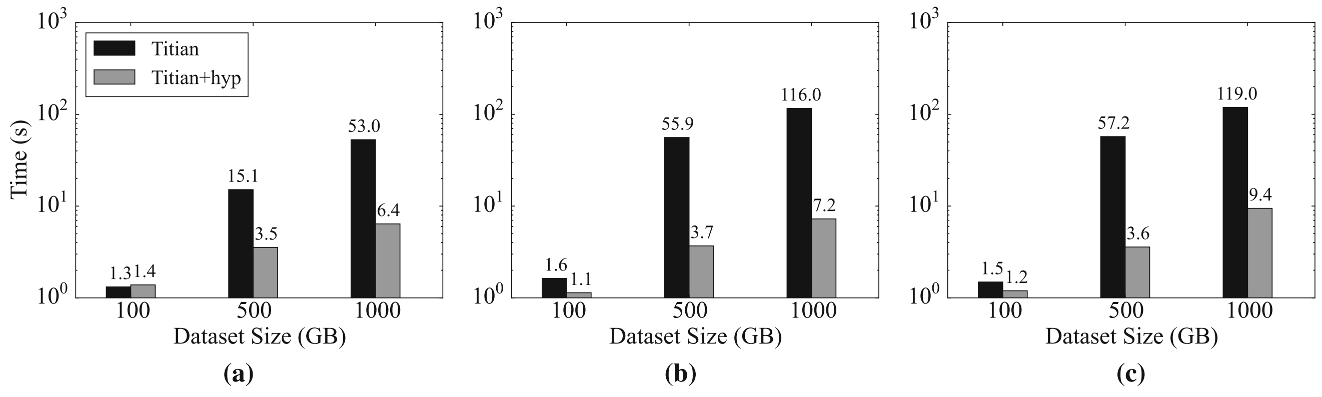


Fig. 26 Forward tracing comparison of Titian with and without Hyperdrive for PigMix queries. Although data access optimizations do not apply for this set of experiments, Hyperdrive is still able to provide up to 10× improvement w.r.t. normal Titian. **a** L2, **b** L6 and **c** L7

causes, that are responsible for answers and non-answers to queries [32]. The Scorpion system finds outliers in the dataset that have the most influence on the final outcome [43]. The Scorpion approach is restricted to simple queries with aggregates and no join operations. Roy et al. overcome the limit of Scorpion in generating explanations over a single table only [38]. From our perspective, we deem these explanation systems as high-level applications of data provenance, and in fact they share many similarities with our automated fault localization service provided in BigDebug [19,20]. However, while they focus on finding tuples that maximize the influence over a set of records of interest, the goal of automated fault localization is to generate the minimal failure-inducing set of records. Moreover, the above systems mainly target specific set of queries and structured data, and therefore, their approach is not applicable to generic programs containing, for example, arbitrary UDFs.

More recently, Chothia et al. [10] implemented data provenance capabilities in a predecessor system to Naiad [35]. Again, different from Titian, their approach is more focused on how to provide semantically correct (and minimal) explanations of outputs through replay, which is much more efficient to support in a differential dataflow system. In contrast, we have shown how data provenance support can be efficiently supported in a (distributed) DISC system like Apache Spark.

8 Conclusion and future work

We began this work by leveraging Newt for data provenance support in Apache Spark. During this exercise, we ran into some usability and scalability issues, mainly due to Newt operating separately from the Spark run-time. This motivated us to build Titian, a data provenance library that integrates directly with the Spark run-time and programming interface.

Titian provides Spark programmers with the ability to trace through the intermediate data of a program execution. Titian's programming interface extends the Spark RDD abstraction, making it familiar to Spark programmers and allowing it to operate seamlessly through the Spark interactive terminal. We introduced several optimizations allowing Titian to execute tracing queries at interactive speeds: our experiments show tracing times constantly below 10 s even for terabyte-sized datasets. We believe the Titian Spark extension will open the door to a number of interesting use cases: we have already discussed here and in other works [19,21,22] how Titian can be leveraged to implement big data programming debugging features; additional use cases are data cleaning [26], large-scale outlier detection, and exploratory data analysis. Titian is publicly available [6] and can be used stand-alone or through the debugging features provided by BigDebug.

In the future, we plan to further integrate Titian both vertically with the many Spark high-level libraries, such as *GraphX* (graph processing) [16], *MLlib* (machine learning) [34] and *Spark SQL* [4]; and horizontally with other DISC systems such as Apache Flink [13] and Apache AsterixDB [5]. We envision that each library and system will motivate certain optimization strategies and data lineage storage requirements, e.g., how lineage can be efficiently generated and stored in a streaming system, or how tracing queries spanning different systems or libraries can be effectively supported.

Another interesting topic we are currently exploring is how to summarize provenance information both from the usability perspective (the output of a tracing query may contain thousands of records, making it impossible to conduct a manual inspection), and space efficiency perspective.

Acknowledgements Titian is supported through Grants NSF IIS-1302698 and CNS-1351047, and 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. Alvaro, P., Rosen, J., Hellerstein, J.M.: Lineage-driven fault injection. In: SIGMOD, pp. 331–346 (2015)
2. Amsterdamer, Y., Davidson, S.B., Deutch, D., Milo, T., Stoyanovich, J., Tannen, V.: Putting lipstick on pig: enabling database-style workflow provenance. VLDB **5**(4), 346–357 (2011)
3. Anand, M.K., Bowers, S., Ludäscher, B.: Techniques for efficiently querying scientific workflow provenance graphs. In: EDBT, pp. 287–298 (2010)
4. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: SIGMOD, pp. 1383–1394 (2015)
5. Asterixdb. https://asterixdb.apache.org/
6. Bigdebug. sites.google.com/site/sparkbigdebug/
7. Biton, O., Cohen-Boulakia, S., Davidson, S.B., Hara, C.S.: Querying and managing provenance through user views in scientific workflows. In: ICDE, pp. 1072–1081 (2008)
8. Borkar, V., Carey, M., Grover, R., Onose, N., Vernica, R.: Hyracks: a flexible and extensible foundation for data-intensive computing. In: ICDE, pp. 1151–1162 (2011)
9. Chambi, S., Lemire, D., Kaser, O., Godin, R.: Better bitmap performance with roaring bitmaps. Softw. Pract. Exp. **46**(5), 709–719 (2016)
10. Chothia, Z., Liagouris, J., McSherry, F., Roscoe, T.: Explaining outputs in modern data analytics. Proc. VLDB Endow. **9**(12), 1137–1148 (2016)
11. Cui, Y., Widom, J.: Lineage tracing for general data warehouse transformations. VLDBJ **12**(1), 41–58 (2003)
12. Dave, A., Zaharia, M., Shenker, S., Stoica, I.: Arthur: Rich post-facto debugging for production analytics applications. Tech. Rep. (2013)
13. Flink. https://flink.apache.org/

14. Glavic, B., Alonso, G.: Perm: Processing provenance and data on the same data model through query rewriting. In: ICDE, pp. 174–185 (2009)
15. Glavic, B., Alonso, G., Miller, R.J., Haas, L.M.: TRAMP: understanding the behavior of schema mappings through provenance. PVLDB **3**(1), 1314–1325 (2010)
16. Gonzalez, J.E., Xin, R.S., Dave, A., Crankshaw, D., Franklin, M.J., Stoica, I.: Graphx: graph processing in a distributed dataflow framework. In: OSDI, pp. 599–613 (2014)
17. Graefe, G., McKenna, W.J.: The volcano optimizer generator: extensibility and efficient search. In: ICDE, pp. 209–218 (1993)
18. Green, T.J., Karvounarakis, G., Ives, Z.G., Tannen, V.: Update exchange with mappings and provenance. In: Proceedings of the 33rd International Conference on Very Large Data Bases, VLDB '07, pp. 675–686. VLDB Endowment (2007)
19. Gulzar, M.A., Han, X., Interlandi, M., Mardani, S., Tetali, S.D., Millstein, T., Kim, M.: Interactive debugging for big data analytics. In: 8th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 16). USENIX Association, Denver, CO (2016)
20. Gulzar, M.A., Han, M.I.X., Li, M., Condie, T., Kim, M.: Automated debugging in data-intensive scalable computing. In: Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '17. ACM, New York (2017)
21. Gulzar, M.A., Interlandi, M., Condie, T., Kim, M.: Bigdebug: interactive debugger for big data analytics in apache spark. In: FSE, pp. 1033–1037 (2016)
22. Gulzar, M.A., Interlandi, M., Yoo, S., Tetali, S.D., Condie, T., Millstein, T., Kim, M.: Bigdebug: debugging primitives for interactive big data processing in spark. In: ICSE, pp. 784–795 (2016)
23. Hadoop. http://hadoop.apache.org
24. Heinis, T., Alonso, G.: Efficient lineage tracking for scientific workflows. In: SIGMOD, pp. 1007–1018 (2008)
25. Ikeda, R., Park, H., Widom, J.: Provenance for generalized map and reduce workflows. In: CIDR, pp. 273–283 (2011)
26. Interlandi, M., Tang, N.: Proof positive and negative in data cleaning. In: ICDE, pp. 18–29 (2015)
27. Interlandi, M., Tetali, S.D., Gulzar, M.A., Noor, J., Condie, T., Kim, M., Millstein, T.: Optimizing interactive development of data-intensive applications. In: Proceedings of the Seventh ACM Symposium on Cloud Computing, SoCC '16, pp. 510–522. ACM, New York, NY, USA (2016)
28. Interlandi, M., Shah, K., Tetali, S.D., Gulzar, M.A., Yoo, S., Kim, M., Millstein, T.D., Condie, T.: Titian: data provenance support in spark. PVLDB **9**(3), 216–227 (2015)
29. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying data provenance. In: Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data, SIGMOD '10, pp. 951–962. ACM, New York, NY, USA (2010)
30. Karvounarakis, G., Ives, Z.G., Tannen, V.: Querying data provenance. In: SIGMOD, pp. 951–962 (2010)
31. Logothetis, D., De, S., Yocum, K.: Scalable lineage capture for debugging disc analytics. In: SOCC, pp. 17:1–17:15 (2013)
32. Meliou, A., Gatterbauer, W., Moore, K.F., Suciu, D.: The complexity of causality and responsibility for query answers and non-answers. PVLDB **4**(1), 34–45 (2010)
33. Missier, P., Belhajjame, K., Zhao, J., Roos, M., Goble, C.A.: Data lineage model for Taverna workflows with lightweight annotation requirements. In: IPAW, pp. 17–30 (2008)
34. Mllib. http://spark.apache.org/mllib
35. Murray, D.G., McSherry, F., Isaacs, R., Isard, M., Barham, P., Abadi, M.: Naiad: a timely dataflow system. In: SOSP. ACM (2013)
36. Olston, C., Reed, B., Srivastava, U., Kumar, R., Tomkins, A.: Pig latin: a not-so-foreign language for data processing. In: SIGMOD, pp. 1099–1110. ACM (2008)
37. Olston, C., Reed, B.: Inspector gadget: a framework for custom monitoring and debugging of distributed dataflows. PVLDB **4**(12), 1237–1248 (2011)
38. Roy, S., Suciu, D.: A formal approach to finding explanations for database queries. In: SIGMOD, pp. 1579–1590 (2014)
39. Spark. http://spark.apache.org
40. Thusoo, A., Sarma, J.S., Jain, N., Shao, Z., Chakka, P., Anthony, S., Liu, H., Wyckoff, P., Murthy, R.: Hive: a warehousing solution over a map-reduce framework. VLDB **2**(2), 1626–1629 (2009)
41. Wang, L., Zhan, J., Luo, C., Zhu, Y., Yang, Q., He, Y., Gao, W., Jia, Z., Shi, Y., Zhang, S., Zheng, C., Lu, G., Zhan, K., Li, X., Qiu, B.: Bigdatabench: a big data benchmark suite from internet services. In HPCA, pp. 488–499 (2014)
42. Welsh, M., Culler, D., Brewer, E.: Seda: an architecture for well-conditioned, scalable internet services. In: SOSP, pp. 230–243 (2001)
43. Wu, E., Madden, S.: Scorpion: explaining away outliers in aggregate queries. Proc. VLDB Endow. **6**(8), 553–564 (2013)
44. Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauley, M., Franklin, M.J., Shenker, S., Stoica, I.: Resilient distributed datasets: a fault-tolerant abstraction for in-memory cluster computing. In: NSDI (2012)
45. Zeller, A., Hildebrandt, R.: Simplifying and isolating failure-inducing input. TSE **28**(2), 183–200 (2002)
46. Zhou, W., Fei, Q., Narayan, A., Haeberlen, A., Loo, B.T., Sherr, M.: Secure network provenance. In: SOSP, pp. 295–310 (2011)
47. Zhou, W., Sherr, M., Tao, T., Li, X., Loo, B.T., Mao, Y.: Efficient querying and maintenance of network provenance at internet-scale. In: SIGMOD, pp. 615–626 (2010)