Supporting Data Provenance in Data-Intensive Scalable Computing Systems

Matteo Interlandi, Tyson Condie
Microsoft

Abstract

Debugging data processing logic in Data-Intensive Scalable Computing (DISC) systems is a difficult and time consuming effort. Data provenance support is a key building block in libraries that aim to provide debugging support for data processing pipelines. In this paper we report our experience in building Titian: a data provenance system targeting the Apache Spark framework. Our focus here is to analyze the design choices and trade offs that we and others made. Ultimately, we believe there is still more work to do before reaching a widespread adoption of data provenance outside the research community.

1 Introduction

Data-Intensive Scalable Computing (DISC) systems, like Apache Hadoop [1] and Apache Spark [2], 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., 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 operators on each partition.

Given its distributed and large-scale nature, debugging data processing logic in DISC environments can be daunting. DISC systems expose a batch model of execution: applications are run in the cloud, and the results, including notification of runtime failures, are sent back to users upon completion. Therefore, debugging is mostly done *post-mortem* and the primary source of debugging information is an execution log. Another common debugging pattern is *trial-and-error* iterations, where developers *selectively replay* a portion of their data processing logic on input samples or subsets of intermediate data leading to erroneous results. Trial-and-error debugging is often a slow and error prone process inasmuch as each iteration is executed afresh, and users have to be manually filter records after each iteration. Only recently, a set of tools and libraries [4–6, 8] have started to arise for helping users in interactively identifying the subset of data leading to failures, or to optimize trial-and-error runs in a principled and automatic way. All these tools can be unlocked by a DISC system equipped with the following capabilities: (1) *scalable fine-grained data provenance* (also referred to data lineage) *capturing* introducing low overhead on the runtime; (2) *interactive provenance query capabilities* enabled within the same host framework; and (3) *a flexible and simple to use API* allowing to seamlessly move between provenance and data records without triggering any re-computation.

Bulletin of the IEEE Computer Society Technical Committee on Data Engineering

Newt [10] and RAMP [7] are two frameworks supporting data provenance in DISC systems (Hadoop in this specific case). Unfortunately, none of them satisfy all three requirements: Newt is an external library whereby the dataflow operators of the target DISC system are wrapped with fine-grained provenance capturing logic downpouring provenance information to a storage layer (MySQL). While this design allows some level of portability, we found that interactiveness is somehow restricted because a different system has to be used at provenance query time. Conversely, RAMP tags data records with provenance information which are propagated downstream and collected into HDFS. This design requires modification to DISC internals while only a limited number of tracing queries are supported efficiently. Furthermore, scalability is suboptimal in both system as a consequence of the overheads due to the transfer of provenance information between different sub-systems (Newt) and downstream as tags (RAMP).

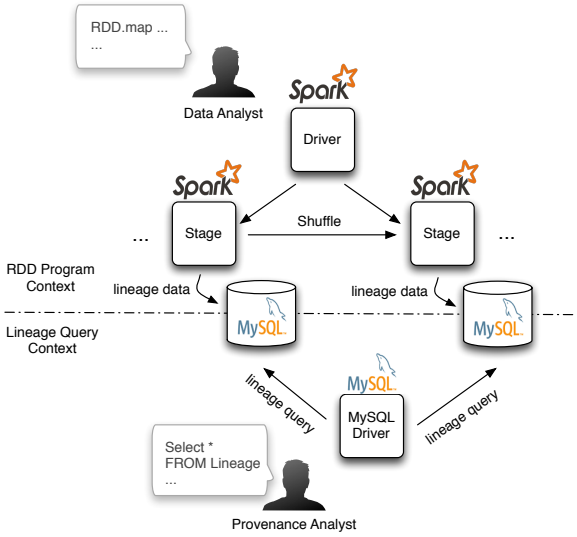
The main focus of this paper is to summarize our experience in building *Titian* [9]: a fine-grained provenance framework specifically designed for the popular Apache Spark system, and satisfying the above three requirements. Differently than RAMP and Newt, Titian was implemented with usability and scalability in mind: provenance capturing and querying is performed on the same host system whereby users can employ the same interface (i.e., RDD transformations [11]) to interactively author DISC programs and query provenance information. By tightly integrating Titian within Spark, both provenance information can be captured with low overhead, and data records can be accessed without any re-computation.

While these design choices made possible the development of several debugging functionalities and tools on top of Titian [4–6, 8], we believe several limitations and open questions remain. Specifically, we found that targeting Titian to Spark's low-level (RDD) API makes it difficult to port to newer versions of Spark. Moreover, mapping data provenance to high-level operations (e.g., SQL, Machine Learning) is non-trivial, and would require a complete overhaul of the framework. A more detailed discussion on these and other issues can be found in Section 5.

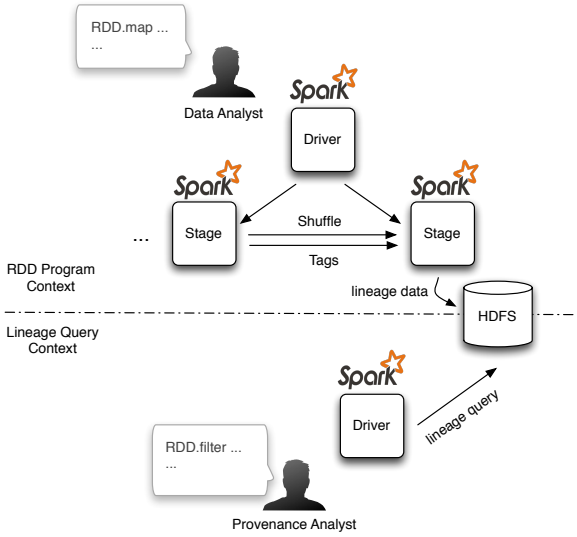
In the remainder of the paper we will first discuss our porting of Newt and RAMP over Apache Spark, and related usability and scalability difficulties (Section 3); then we will introduce Titian design and detailed implementation (Section 4). The paper will end with a discussion on the lesson learned and future directions. To properly put everything into context, we next briefly introduce Apache Spark.

2 Background: Apache Spark

Spark is a JVM-based DISC system exposing a programming model based on Resilient Distributed Datasets (RDDs) [11]. The RDD abstraction provides *transformations* (e.g., map, reduce, filter, group-by, join, etc.) 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) to the Spark *driver program*. A driver program could be a user operating through the Spark terminal, or it could be a standalone Scala program. In either case, RDDs lazily evaluate transformations by returning a new RDD object that is specific to the transformation operation on the target input RDD(s). Actions trigger the evaluation of an RDD, 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). `reduceByKey`, `groupBy` and `join` are common *stage-breaking* RDD transformations requiring data re-partitioning. 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., sequence of transformations) of a stage on each input partition. Records composing each data partition are presented to a task in the form of an iterator i.e., Spark follows the record-at-a-time dataflow model of databases. 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 eventually initiate another series of transformations ending with an action. For fault-tolerance, within stages data is materialized on



(a) With Newt, a MySQL cluster is co-located with the DISC (i.e., Spark) one. The data analyst authors and submits her / his program through the Spark Driver. Spark with Newt instrumentation generates data provenance data and stores it into local MySQL instances. Once the job is completed, a provenance analyst queries the generated provenance graph using the MySQL Driver interface.



(b) With the RAMP instrumentation, provenance records are propagated downstream and saved into HDF. Data and provenance analysts use the same RDD interface. However interactiveness is limited since two different contexts are required.

Figure 1: The Newt and RAMP system approach for DISC provenance.

persistent memory and re-partitioned. Spark additionally allows programmer to *cache* RDDs in memory. When a cached RDD is scheduled for execution, Spark's *BlockManager* eagerly retrieves the saved RDD data instead of triggering the evaluation of preceding RDDs. This mechanism is particularly useful for programs containing iterations.

3 Data Provenance in DISC: RAMP and Newt

Our initial work in adding fine-grained provenance support to Spark leveraged RAMP and Newt designs. During this exercise, we encountered a number of issues, including scalability (the sheer amount of fine-grained provenance data that could be captured and used for tracing), job overhead (the per-job slowdown incurred from data provenance capture), and usability (both provide limited support for provenance queries). Newt operates externally to the target DISC system, making it more general than Titian. However, the Newt design prevents a unified programming environment, in which both the data and its provenance can be queried in the same runtime. RAMP is more tightly integrated into the target DISC system (e.g., Hadoop MapReduce), providing better scalability, but, like Newt, the RAMP design lacks a unified solution for data and provenance analysis.

In the following we present a qualitative analysis of our experience in porting Newt and RAMP over Spark. For a more quantitative evaluation, we refer interested readers to [9]. Figure 1 pictorially summarizes the Newt and RAMP instrumentation of Spark.

3.1 Newt

Newt is a generic framework designed for collecting and querying fine-grained data provenance information from any framework executing data transformations as logical operators. When porting Newt to Spark, we avoided modifications to Spark runtime so that it could be leveraged in different versions of Spark. Newt follows the *agent* model whereby target data transformations are instrument to capture provenance information on operator inputs and operator outputs. Newt provides a simple `addInput` and `addOutput` API accepting data records as input and generating timestamped unique provenance IDs (using hashing). IDs and timestamps are streamed to a *Newt client* and saved into a local log file. Once program execution is complete, Newt uses the timestamp values to infer the temporal order of outputs IDs relative to inputs provenance IDs to reconstruct the original input-output relationships between records. All reconstructed input-output relationship pairs are then loaded into an indexed *association table* stored in a MySQL distributed cluster co-located with the Spark cluster.

Capturing. Within Spark stages, the instrumentation for collecting data provenance using the Newt API is fairly straightforward: we created two special `map` RDD transformations that generate input and output provenance associations for each Spark stage using the `addInput` and `addOutput` API provided by Newt. These provenance associations are then pushed by the Newt client to MySQL once the input partition has been processed to completion. However, transformations such as `reduceByKey` and `join` require a shuffle step, during which all records are materialized. The simple timestsamp-based API of Newt is not effective in generating minimal provenance data because all input records end up being associated with each output record. Newt does provide the ability to add an optional *tag* to each record (for instance to tag records belonging to the same group-by key) so that only records with same tag are linked in the input-output association table. However tags need to be propagated through the shuffle step, which could be supported by either (1) modifying the target program so that shuffle operations (i.e., `reduceByKey`) accept as input pairs in the form $(key, (value, tag))$; or (2) modifying the Spark internals in order to make propagations of tags transparent to users. The latter is the approach that RAMP (and Titian) took, and will be explained in Section 3.2.

Querying. Provenance queries are executed in Newt as a series of SQL joins executed over the association tables stored in the MySQL cluster. Queries are issued from a *MySQL Driver* node: a logically separated entity from the Spark Driver. We found this approach suboptimal from a usability perspective because: (1) users have to setup a MySQL cluster together with the DISC system cluster; (2) analysts needs to be proficient on two systems in order to debug their programs; (3) for the system to be able to properly compose tracing query plans, stored association tables need to be explicitly linked to form a dependency graph, mirroring the position they occupy in the original input program; and (4) populating indexed MySQL tables starting from log files is time consuming (on the order of minutes to hours depending on the data volume). The latter two points make the Newt design difficult to use in interactive sessions.

Additionally, the raw intermediate data is not available in the Newt design, making it only relevant for tracing back to the input datasets; further limiting its use in a (stepwise) debugger like setting. We did not explore checkpointing (saving) the intermediate data in our Newt to Spark port since we had already hit a scalability bottleneck when capturing the provenance alone.

Lessons Learned. Summarizing, from our initial experience of adding data provenance to Spark through Newt we concluded the following:

- To minimize the overhead of tapping record pipelines, it is better by default to *capture provenance information at stage boundary only*, and eventually give to users the ability to manually inject additional capturing points if necessary.
- Without any modification to Spark internals or user code, Newt timestamp-based approach is not able to produce minimal input-output relationship tables for transformations requiring shuffling. As a consequence, capturing and querying have higher overheads wrt more optimal solutions where only minimal

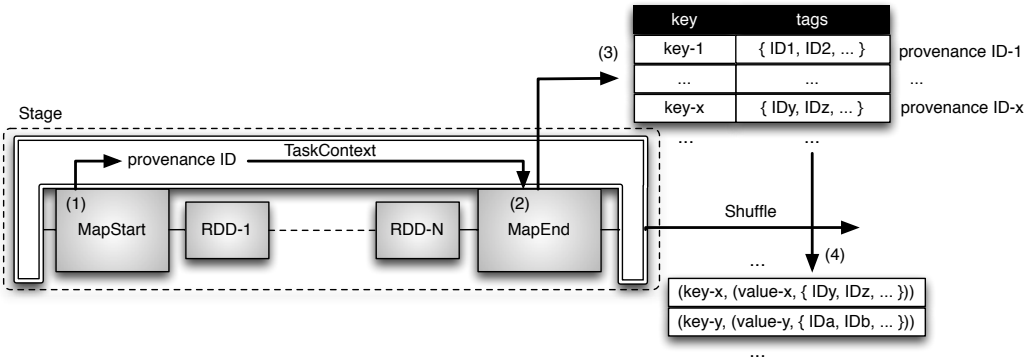


Figure 2: In RAMP Provenance IDs are propagated downstream as record TAGS. IDs are first added to the TaskContext (1). `MapEnd` pulls IDs from the context (2) and populate a local (to each task) hash table (3). Finally, collected provenance identifiers are attached to the proper key-value pair (4).

Figure 3: RAMP approach of propagating provenance through the TaskContext and the shuffle step.

provenance information are saved.

- Data provenance capturing and querying have to be executed on the same (DISC) system if we want to allow interactivity both to end users and to higher-level debugging toolkits.
- Intermediate data records (i.e., records stored into shuffle files) linked to provenance IDs have to be made accessible by users in order to provide better insight into intermediate RDD transformations.

3.2 RAMP

The initial RAMP implementation was specifically tailored for MapReduce workflows, but its design can easily be ported over Spark. Unlike Newt, RAMP integrates with the target DISC system (e.g., Spark) to efficiently tag records with chains of (nested) provenance IDs that are propagated with the data records through the dataflow to the final record outputs. These provenance chains can be seen as materializing backward tracing joins between (virtual) association tables at the final program output. Indeed, with such design, backward tracing provenance queries can be answered very efficiently. Conversely, forward queries are difficult and inefficient to support.

Capturing. When a Spark program is submitted for execution (i.e., when an action is called on an RDD), the RDD dependencies are analyzed and two new `map` RDDs (`rampMapStart` and `rampMapEnd`) are injected into the program before and after (respectively) any RDDs that translate into a single stage. For input datasets that reside in Hadoop HDFS, Spark uses a `HadoopRDD` that emits the data record along with an *offset* of the record in the input file. We inject a `rampMapStart` RDD that consumes these records and uses the partition identifier and offset as provenance ID. Since RAMP does not timestamp records, input provenance IDs need to be propagated downstream to the stage output, where associations are made with the relevant output provenance IDs. We implement provenance propagation by adding the provenance ID of the current input record to the Spark *TaskContext*, as shown in Figure 3 (1). Since records are evaluated one-at-a-time, each provenance ID is relevant to all output records sent to `rampMapEnd` at the stage output[1] (2). `rampMapEnd` is responsible for associating each output record key with all relevant input provenance IDs (3). All the provenance IDs accumulated into local hash tables have to be propagated as well, but this time through the shuffle step. To achieve this the *ExternalSorter* component in Spark had to be modified so that records can be written in the form $(key, (value,$

[1]Note that Newt does not make this assumption, and instead relies on timestamps to infer (indeed overestimate) input-to-output associations.

provenance ID) in the shuffle file, where provenance ID contains all the IDs accumulated into the has table under key (4), i.e., previously stored provenance IDs are nested and used as a new provenance ID. Note that only nested IDs need to be added to each record, since the hash of the key can be recomputed on the fly.

In the following stage, the *ShuffleReader* and *Aggregator* components had to be changed to make them aware of the new record format. Apache Spark computes aggregates (a similar argument holds for joins) by iterating over all records while updating the aggregate value linked to each key. The final aggregate values are then surfaced to the successive RDD transformations as iterator. Hence, after the shuffling step provenance information are generated in two phases. In the first phase, while the aggregator iterates through records, IDs are removed so that aggregation can be computed as in regular Spark. The provenance IDs (each composed by the sequence of IDs coming from the previous stage `rampMapStart`) are then used together with the key to populate a new local hash table, as in `rampMapEnd`. Once the aggregation phase is completed, we store the generated hash table containing all provenance IDs into a buffer in the TaskContext. In the second phase, when the aggregated records are emitted as iterator, a `rampReduceStart` RDD previously injected in the workflow pulls the current provenance ID from the buffer, appends a new provenance ID to it, and stores the updated provenance chain into the TaskContext so that the `rampMapEnd` transformation at the end of the stage can use it. The process repeat if other stages follow. The `rampMapEnd` in the final stage materialize all nested provenance IDs in HDFS.

Querying. Provenance queries are implemented in RAMP by traversing and unnesting HDFS residing provenance IDs. For this task, external scripts can be used, but interestingly also the DISC system itself. This later approach is closer to our target of providing provenance querying capabilities withing the DISC framework. Note however that in both cases, only backward tracing queries are supported efficiently, while forward tracing requires to scan and unroll all provenance IDs. Additionally, RAMP model allows to access the (map) input and (reduce) output data records connected with provenance IDs but no intermediate records.

Lessons Learned. The RAMP porting provided us the baseline for the Titian implementation.

- The offloading of provenance information to HDFS-stored files is simple and effective, and allows to use the same DISC system for provenance querying.
- The approach of propagating nested provenance IDs introduces not negligible overheads into the runtime, moreover redundant information are added to shuffle files.
- Both backward and forward tracing queries should be efficiently supported. Additionally, provenance queries should be expressible in a declarative language leaving the system to decide how to properly execute (and optimize) them.

4 Titian

Titian integrates with the Spark runtime to provide efficient data provenance support. Submitted programs are rewritten to include lineage capturing (map) transformations at stage boundaries that generate provenance association tables, which are stored directly into Spark's in-memory store (BlockManager). Additionally, partition identifiers are propagated with data records through shuffle steps in order to optimize tracing queries.

Provenance information is exposed as RDDs, on which all Spark native transformations, including some additional tracing capabilities, can be applied. This approach makes program execution and debugging a continuous process carried out interactively using the same Spark context. Figure 5 describes the Titian design.

4.1 Capturing

The entry point of Titian is the `LineageContext` that wraps the original `SparkContext` to enable data provenance capabilities. When a program is submitted for execution, Titian rewrites the program by injecting

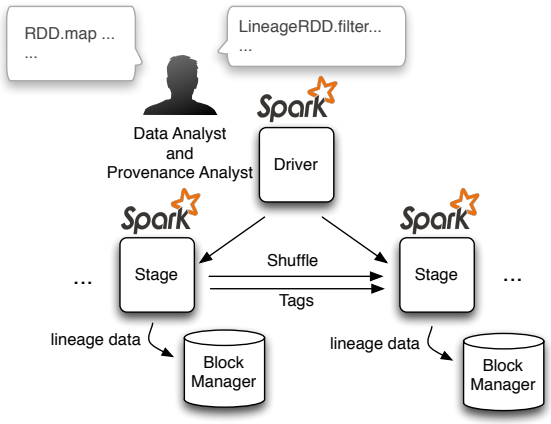


Figure 4: With Titian, users can author DISC programs and analyze their provenance trace interactively within the same (Spark) context.

Figure 5: Titian design for DISC provenance.

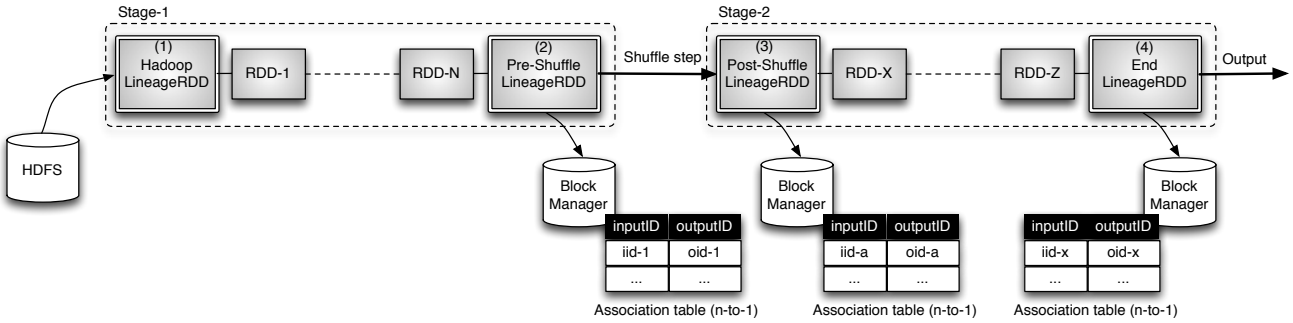


Figure 6: A two stage program instrumented with Titian.

LineageRDD transformations at stage boundaries, as shown in Figure 6. LineageRDDs can be classified into four types based on where they are positioned in the program.

(1) Program Input. Titian adds a HadoopLineageRDD or a ParallelizeLineageRDD, based on whether input records are fetched from HDFS or directly from the driver. The implementation of these LineageRDDs is similar to the `rampMapStart` transformation previously described in Section 3.2. For each input data record, an unique INPUTID (such as the filename, partition id and offset in an HDFS file) is propagated to the stage output using the TaskContext.[2]

A stage ends when one of the following operations occur: *reduction* (i.e., `reduceByKey`), *generic aggregation* (e.g., `groupBy`), or *co-grouping* operators (e.g., `join`, `union`, etc.). The differences between the three are both in the implementation and in the output format. Spark injects a combiner step when a reduction operation is requested, while generic aggregation is performed without a combiner. In both cases, the successive stage starts with an RDD iterating over a sequence of pairs where the first element is the grouping key, while the second element is the related aggregate value. Conversely, co-grouping (similarly to generic aggregation) uses no combiner, and outputs a sequences of pairs composed by the grouping key and an iterator over all the values. Join results are produced by flattening the co-group output. Next, we describe how we take these output results, generate identifiers for them, and associate them with the relevant input record identifier.

[2]Input records follow at 1-to-1 mapping, avoiding the need to generate association tables at this point.

(2) Pre-Shuffle. Before the shuffle step, Titian injects a proper LineageRDD based on the operation the program is implementing: i.e., PreReduceLineageRDD in case of `reduceByKey`, PreGroupLineageRDD for `groupBy`, and PreCoGroupLineageRDD for a transformation requiring co-grouping. Pairs of ids (INPUTID, OUTPUTID) are generated and buffered in memory by each operator, where INPUTID is the tag of the record propagated from some other LineageRDD upstream, and OUTPUTID is the hash of the key of the record. Additionally, Titian attaches to each shuffled record a partition ID so that post-shuffle records can be efficiently joined with pre-shuffle records i.e., the partition ID indicates which pre-shuffle partitions need to be considered; without this information, we would need to join with all pre-shuffle partitions, which is expense for tracing starting from a small set of output records.

(3) Post-Shuffle. In the successive stage following the shuffle step, Titian rewrites the program workflow by substituting Spark default ShuffleReader with one of the three ReduceShuffleReader, GroupShuffleReader, and CoGroupShuffleReader. Additionally, a PostReduceLineageRDD, PostGroupLineageRDD and PostCoGroup LineageRDD are added to the program based on its semantics. ReduceShuffleReader and GroupShuffleReader follows the same logic of the previously introduced RampReduceStart. Instead, when co-grouping, Spark does not aggregate values by key, but instead it directly returns an iterator over key-value pairs, in which the values are iterators of all the records having the same key. Because of this, each of this records contain the partition ID previously attached during the pre-shuffle phase. PostCoGroupLineageRDD implementation therefore (1) unrolls each record-iterator and save the partition identifier of each record in an in-memory buffer; (2) generates a new unique identifier and stores it into the TaskContext; and (3) emits the new key-value pair without partition identifiers into the successive RDD transformations.

(4) Program Output. At the end of each program Titian injects an EndLineageRDD creating a pair (INPUTID, OUTPUTID) for each data record. Additionally, EndLineageRDD attaches OUTPUTID to the record so that users can, at query time, connect records to provenance IDs. Since the input-output relationship is 1-to-n (e.g., due to a flat map transformation), an association table is explicitly materialized in the BlockManager.

Buffering and Lineage Storage. In order to reduce the overhead of capturing lineage, Titian stores provenance information in-memory into Spark's BlockManager. Additionally, to reduce the number of objects created at runtime, we have extended Spark's worker nodes runtime (Executors) with a pool of *bytebuffers* of different length. When a partition is scheduled for execution, at the first call each LineageRDD initializes a local in-memory buffer by fetching a bytebuffer from one of the pools. Each input-output pair of provenance IDs generated by the LineageRDD is then stored into the local buffer. When a partition is completed, local buffers are asynchronously materialized in the BlockManager after that a LineageRDD-specific compression logic is executed over the provenance records. Specifically, we store pre- and post-shuffle provenance data in a nested format in order to not waste memory space over redundant information. In fact, pre- and post-shuffle LineageRDD operators creates a sequence of (INPUTID, OUTPUTID) pairs in which the same OUTPUTID can appear multiple times based on how many times values with the same key exist. The trade off is that such technique increases the overhead of accessing data at query time since records need to be unnested, but other techniques such as *targeted joins* (described in Section 4.2) can be used to speed-up query processing.

4.2 Querying

Once the DISC program with provenance capturing enabled is completed, Titian surfaces provenance data as LineageDataRDDs: a specific RDD equipped with provenance-specific operations such as `TraceBackward` or `TraceForward` additionally to regular RDD transformations. To enable provenance queries, upon program completion, Titian (1) labels all LineageRDDs injected into the original program as cached; and (2) generates a dependency graph formed by all such LineageRDDs now surfaced as LineageDataRDD references. With (1), Titian is basically instrumenting Spark to fetch the provenance data available in the BlockManager when an

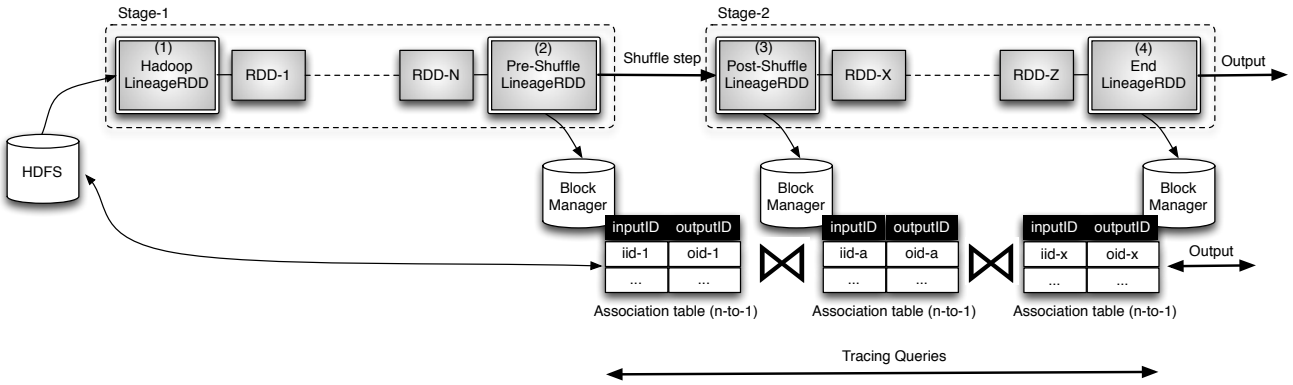


Figure 7: Tracing queries are implemented in Titian as a sequence of (distributed) joins between association tables.

action requires to compute a LineageDataRDD[3]; since provenance data is exposed as RDDs, we simply leverage Spark to query it, without asking users to manually specify the dependencies between associations tables (as for instance Newt and RAMP require).

Starting from a target LineageDataRDD of interest, analyst can submit tracing queries using the `TraceBackward` or `TraceForward` transformations.[4] Furthermore, regular RDD transformations such as `filter` can be used to remove out all provenance records that are not of interest (for instance by specifying the OUTPUTIDs of the records that need to be traced). Taking the backward direction as example, when a user call the method `traceBackard()` over a (filtered) LineageDataRDD reference, Titian's *QueryPlanner* schedules a left semi-join between the current LineageDataRDD and the preceding one in the dependency graph. If users prefer to execute more than one step backward, `traceBackward(numSteps)` can be called where `numSteps` specifies the numbers of joins to be executed. Finally, a full trace backward up to the initial input IDs can be executed by calling `fullTraceBackward()`. Figure 7 is a replica of Figure 6 where we show how joins are executed using the association tables generated from the two-stages program.

Query Planning and Optimizations. When a tracing query is issued for evaluation (e.g., after an action is invoked on a LineageDataRDDs reference returned from a `traceBackword` call), Titian generates a query plan that attempts to minimize data movement, and unrolling nested records only when necessary. Titian's QueryPlanner avoids Spark's shuffle join implementation when tracing within a stage i.e., from the stage output to the stage input (or vice-versa), since the respective associations already join locally. A shuffle join is needed when tracing between shuffle steps. However, instead of using Spark's native shuffle join, we implemented a *direct shuffle join* that uses the partition identifier information to directly traces to only the relevant partitions on the other side of the shuffle step. For example, if we are tracing back from a record with key "foo", then it might be the case that not all pre-shuffle steps generated such a key, in which case the partition identifiers will inform the direct shuffle to avoid joining those partitions with the "foo" records. Additionally, indexes are create at capture time to speed-up the joining process when tracing backward. More details on query planning and optimizations can be found in [9].

Accessing Original Data Records. Titian enables users to inspect the data records that each provenance ID is linked to, including intermediate data, without introducing any overhead in the capturing phase. In fact, while input and output records are directly connected to the related provenance ID by construction, intermediate

[3]Note that in Spark caching is usually request at program-time *before* an RDD is scheduled for execution. Titian instead asks Spark to cache LineageRDD *after* they are computed. Indeed LineageRDDs internally materialize provenance data into the BlockManager, i.e., without using Spark support. Since LineageRDD are manually saved, Spark is unaware of the checkpoints and therefore fault tolerance mechanisms work as usual.

[4]The tracing transformations are simply syntactic sugar over native Spark (filtered) joins.

records (i.e., records produced as output of a intermediate stage and consumed by successively scheduled stages) can be retrieved by directly accessing (saved) shuffle files. Spark, in general, maintains shuffle files in cluster memory for fault tolerance. Such files survive after the execution of the target program and therefore can be read by provenance queries. This is possible exclusively because one unique context is used for both data and provenance analysis.

Accessing the data linked to a set of provenance IDs is possible in Titian by calling the `showData` method over the target LineageDataRDD object. Underneath, Titian's query planner issues a join between the LineageDataRDD and the related object reference pointing to the data. For example, a call to `showData()` over an HadoopLineageDataRDD object will issue a join between the provenance records of the HadoopLineageDataRDD object, and the input dataset stored in HDFS. Similarly, a call to `showData()` over a PreShuffleLineageDataRDD issues a join with the shuffle file. Note that the Titian framework automatically maintains the reference to all (intermediate) files and objects storing data records that can be eventually requested by users during tracing.

5 Considerations and Future Directions

Newt, RAMP, and Titian provided different contributions over the state of the art. Newt showed that a *portable*, external library can be developed such that (different) DISC systems can be easily instrumented to capture provenance. Higher-level tooling can then target such library and seamlessly work over different DISC platforms. In our experiments Newt however showed *poor scalability*. This is both the result of its portable design, and the choice of using MySQL as lineage capturing backend. This later choice of separating the query subsystem from the DISC system, lowers its usability and makes features such as visualization of intermediate data difficult to achieve.

RAMP chose a more integrated design with the host DISC system at the cost of less portability. Propagating full provenance information downstream introduces however major overhead on the running program. Lastly, while provenance queries are supported in RAMP using the same DISC language, the original RAMP design did not consider forward tracing queries, but only (already materialized) backward queries.

Titian implementation merges the advantages of both the Newt and RAMP approaches: Titian integrates the provenance capturing infrastructure within the host DISC system, but instead of propagating tags, it materialized input-output association tables in memory. Titian's integration with Spark does not limit Spark's scaling capabilities (we experimented with dataset up to 1TB) with an average overhead of 30%. Titian choice of surfacing provenance data as RDD, greatly improves usability by allowing users to employ the same Spark Context for both program authoring and provenance analysis. Additionally, the optimizations implemented in Titian brings interactive speed evaluation of tracing queries. However, we think that still many open questions remain before truly reaching the goal of having a industry-ready data provenance library for DISC systems. We next sketch few possible directions for future work on the three dimensions of *portability*, *usability* and *scalability*.

Portability. We found Titian tailored integration with Spark low-level (RDD) API both a blessing and a curse: upgrading Titian to newer versions of Spark is in general not easy; similarly enabling fine-grained provenance capturing over Spark's graph, ML, or relational API is not trivial and requires re-implementing major parts of the framework. We are starting to see more applications requiring mixed programming models. While Spark does provide a unifying infrastructure to execute mixed applications, to our knowledge no solution exists which is able to trace data provenance effectively end-to-end through such mixed programs. The problem is even exacerbated when instead of a single system (Spark), multiple specialized frameworks are used to implement a data analytics pipeline. While each framework may have the ability to provide some provenance information, unifying it in a single interactive session and language is an open problem.

Usability. While we started the exploration of high-level tooling exploiting provenance for debugging DISC program [4–6, 8], we think that much work still have to be done in the field. For instance, in [6] we introduced

a system for automatically detecting the minimum set of failures inducing inputs given a user-provided test function. It would be interesting to see the application of similar techniques over different domains beyond software engineer (e.g., outliers detection or data cleaning), and generating proper domain-specific explanations or actions, beyond a minimum set of evidences of a failure.

Scalability. From our experience with Titian, we found that capturing fine-grained provenance data can generate extremely large provenance graphs; in the same order of the original input data. Instead of focusing on integrating the capturing infrastructure as close as possible to the data source, another possible solution to achieve better scalability is to exploit application information to summarize the provenance data [3], or, alternatively, to push provenance queries into the capturing phase, so that only a subset of the provenance is actually captured. These approaches may lower the generality of the type of analysis that can be carried on over the provenance graph, but for certain applications (or scales) could be the right solution.

References

[1] Apache Hadoop. `http://hadoop.apache.org`

[2] Apache Spark. `http://spark.apache.org`

[3] R. Diestelkämper, M. Herschel and P. Jadhav. Provenance in DISC Systems: Reducing Space Overhead at Runtime. *TaPP*, 2017.

[4] M. A. Gulzar, M. Interlandi, S. Yoo, S. D. Tetali, T. Condie, T. Millstein and M. Kim. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. *ICSE*, 2016.

[5] M. A. Gulzar, M. Interlandi, T. Condie, and M. Kim. Debugging Big Data Analytics in Spark with BigDebug. *SIGMOD*, 2017.

[6] M. A. Gulzar, M. Interlandi, X. Han, M. Li, T. Condie, and M. Kim. Automated Debugging in Data-Intensive Scalable Computing. *SoCC*, 2017.

[7] R. Ikeda, H. Park and J. Widom. Provenance for Generalized Map and Reduce Workflows. *CIDR*, 2011.

[8] M. Interlandi, M. A. Gulzar, M. Kim and T Condie. Optimizing Interactive Development of Data-Intensive Applications. *SoCC*, 510-522, 2016.

[9] M. Interlandi, A. Ekmekji, K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. Millstein and T Condie. Adding data provenance support to Apache Spark *VLDBJ*, 2017.

[10] L. Dionysios, D. Soumyarupa and Y. Kenneth. Scalable Lineage Capture for Debugging DISC Analytics *SOCC*, 2013.

[11] M. Zaharia, M .Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker and I. Stoica. Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing. *NSDI*, 2012.