PerfDebug: Performance Debugging of Computation Skew in Dataflow Systems

Jason Teoh
jteoh@cs.ucla.edu
University of California, Los Angeles

Muhammad Ali Gulzar
gulzar@cs.ucla.edu
University of California, Los Angeles

Guoqing Harry Xu
harryxu@cs.ucla.edu
University of California, Los Angeles

Miryung Kim
miryung@cs.ucla.edu
University of California, Los Angeles

ABSTRACT

Performance is a key factor for big data applications, and much research has been devoted to optimizing these applications. While prior work can diagnose and correct *data skew*, the problem of *computation skew*—abnormally high computation costs for a small subset of input data—has been largely overlooked. Computation skew commonly occurs in real-world applications and yet no tool is available for developers to pinpoint underlying causes.

To enable a user to debug applications that exhibit computation skew, we develop a post-mortem performance debugging tool. PerfDebug automatically finds input records responsible for such abnormalities in a big data application by reasoning about deviations in performance metrics such as job execution time, garbage collection time, and serialization time. The key to PerfDebug's success is a data provenance-based technique that computes and propagates *record-level computation latency* to keep track of *abnormally expensive records* throughout the pipeline. Finally, the input records that have the largest latency contributions are presented to the user for bug fixing. We evaluate PerfDebug via in-depth case studies and observe that remediation such as removing the single most expensive record or simple code rewrite can achieve up to 16X performance improvement.

CCS CONCEPTS

• **Information systems** → **MapReduce-based systems**; • **Theory of computation** → **Data provenance**; • **Software and its engineering** → **Software testing and debugging**; *Software performance*; • **General and reference** → *Performance.*

KEYWORDS

Performance debugging, big data systems, data intensive scalable computing, data provenance, fault localization

SoCC '19, November 20–23, 2019, Santa Cruz, CA, USA

ACM ISBN 978-1-4503-6973-2/19/11…$15.00
https://doi.org/10.1145/3357223.3362727

ACM Reference Format:
Jason Teoh, Muhammad Ali Gulzar, Guoqing Harry Xu, and Miryung Kim. 2019. PerfDebug: Performance Debugging of Computation Skew in Dataflow Systems. In *ACM Symposium on Cloud Computing (SoCC '19), November 20–23, 2019, Santa Cruz, CA, USA.* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/3357223.3362727

1 INTRODUCTION

Dataflow systems are prevalent in today's big data ecosystems and continue to grow in popularity. Since these systems ingest terabytes of data as input, they inherently suffer from long execution times and it is important to optimize their performance. Consequently, studying and improving the performance of dataflow systems such as Apache Hadoop [2] and Apache Spark [4] has been a major research area. For example, Ousterhout et al. [23] study performance across system resources such as network, disk, and CPU and conclude that CPU is the primary source of performance bottlenecks. Several prior works aim to optimize system configurations to achieve better resource utilization [7, 10, 26, 27], while others improve parallelization of workloads [17]. To find the source of performance bottlenecks, PerfXplain [15] performs a differential analysis across several Hadoop workloads.

Computation Skew. When an application shows signs of poor performance through an increase in general CPU time, garbage collection (GC) time, or serialization time, the first question a user may ask is *"what caused my program to slow down?"* While stragglers—slow executors in a cluster—and hardware failures can often be automatically identified by existing dataflow system monitors, many real-world performance issues are *not* system problems; instead, they stem from a *combination of certain data records from the input and specific logic of the application code* that incurs much longer latency when processing these records—a phenomenon referred to as *computation skew*. Computation skew commonly occurs in real-world applications [16, 18]. For example, in a StackOverflow post [1] a Stanford Lemmatizer pre-processes customer reviews. The task fails to process a relatively small dataset because certain sentences trigger excessive memory consumption and garbage collection, leading to large memory usage and execution time. This example is described in detail in Section 2.1. Although there is a body of work [6, 16, 17] that attempts to mitigate data skew, computation skew has been largely overlooked and tools that can identify and diagnose computation skew, unfortunately, do not exist.

PerfDebug. The pervasive existence of computation skew in real-world applications as well as the lack of effective tooling strongly calls for development of new debugging techniques that can help developers quickly identify skew-inducing records. In response to this call, we developed PerfDebug, a novel runtime technique that aims to pinpoint expensive records ("needles") from potentially billions ("haystack") of records in the input.

In this paper, we focus on Apache Spark as it is the de-facto data-parallel framework deployed widely in industry. Spark hosts a range of applications in machine learning, graph analytics, stream processing, etc., making it worthwhile to build a specialized performance debugging tool which can provide immediate benefit to all applications running on it. Although PerfDebug was built for Spark, our idea is applicable to other dataflow systems as well. PerfDebug provides fully automated support for postmortem debugging of computation skew. At its heart are a novel notion of *record-level latency* and a data-provenance-based technique that computes and propagates record-level latency along a dataflow pipeline.

A typical usage scenario of PerfDebug consists of the following three steps. First, PerfDebug monitors coarse-grained performance metrics (*e.g.,* CPU, GC, or serialization time) and uses task-level performance anomalies (*e.g.,* certain tasks have much lower throughput than other tasks) as a signal for computation skew. Second, upon identification of an abnormal task, PerfDebug re-executes the application in the debugging mode to collect data lineage as well as record-level latency measurements. Finally, using both lineage and latency measurements, PerfDebug computes the cumulative latency for each output record and isolates the input records contributing most to these cumulative latencies.

We have performed a set of case studies to evaluate PerfDebug's effectiveness. These case studies were conducted on three Spark applications with inputs that come from industry-standard benchmarks [5], public institution datasets [22], and prior debugging work [8]. For each application, we demonstrate how PerfDebug can identify the source of computation skew within 86% of the original job time on average. Applying appropriate remediations such as record removal or application rewriting leads to 1.5X to 16X performance improvement across three applications. On a locally simulated setting, PerfDebug identifies delay-inducing records with 100% accuracy while achieving 10^2 to 10^6 orders of magnitude precision improvement compared to an existing solution [12], at the cost of 30% instrumentation overhead. To the best of our knowledge, PerfDebug is the first debugging technique to diagnose and reason about computation skew in dataflow applications.

Large performance gains can be obtained by appropriately remediating expensive records (*e.g.,* breaking a long sentence into multiple short ones or even deleting them, if appropriate). PerfDebug delegates repair efforts to the user. In many cases, simple modifications of expensive data records do not have much impact on the correctness of program results for two major reasons: (1) many big data workloads use sampled data as input and hence their results are *approximate* anyway; and (2) the number of such expensive records is often small and hence the delta in the final result that comes from altering these records is marginal.

The rest of the paper is organized as follows: Section 2 provides necessary background. Section 3 motivates the problem and Section 4 describes the implementation of PerfDebug. Section 5 presents experimental details and results. We conclude the paper with related works and a conclusion in Sections 6 and 7 respectively.

```
1  val data = "hdfs://nn1:9000/movieratings/*"
2  val lines = sc.textFile(data)
3  val ratings = lines.flatMap(s => {
4      val reviews_str = s.split(":")(1)
5      val reviews = reviews_str.split(",")
6      val counts = Map().withDefaultValue(0)
7      reviews.map(x => x.split("_")(1))
8             .foreach(r => counts(r) += 1)
9      return counts.toIterable
10 })
11 ratings.reduceByKey(_+_).collect()
```

Figure 1: Alice's program for computing the distribution of movie ratings.

2 BACKGROUND

In this section, we explain the difference between computation and data skew along with a brief overview of the internals of Apache Spark and Titian.

2.1 Computation Skew

Computation skew stems from a *combination of certain data records from the input and specific logic of the application code* that incurs much longer latency when processing these records. This definition of computation skew includes some but not all kinds of data skew. Similarly, data skew includes some but not all kinds of computation skew. Data skew is concerned primarily with *data distribution—e.g.,* whether the distribution has a long (negative or positive) tail—and has consequences in a variety of performance aspects including computation, network communication, I/O, scheduling, *etc.* In contrast, computation skew focuses on record-level anomalies—a small number of data records for which the application (*e.g.,* UDFs) runs much slower, as compared to the processing time of other records.

As previously described, a StackOverflow question [1] employs the Stanford Lemmatizer (i.e., part of a natural language processor) to preprocess customer reviews before calculating the lemmas' statistics. The task fails to process a relatively small dataset because of the lemmatizer's exceedingly large memory usage and long execution time when dealing with certain sentences: due to the temporary data structures used for dynamic programming, for each sentence processed, the amount of memory needed by the lemmatizer is three orders of magnitude larger than the sentence itself. As a result, when a task processes sentences whose length exceeds some threshold, its memory consumption quickly grows to be close to the capacity of the main memory, making the system suffer from extensive garbage collection and eventually crash. This problem is clearly an example of computation skew, but *not* data skew. The number of long sentences is small in a customer review and different data partitions contain roughly the same number of long sentences. However, the processing of each such long sentence has a much higher resource requirement due to the combinatorial effect of the length of the sentence and the exponential nature of the lemmatization algorithm used in the application.

As another example of pure computation skew, consider a program that takes a set of (key, value) pairs as input. Suppose that the length of each record is identical, the same key is never repeated, and the program contains a UDF with a loop where the iteration count depends on $f(value)$, where f is an arbitrary, non-monotonic function. There is no data skew, since all keys are unique. A user cannot simply find a large value v, since latency depends on $f(v)$ rather than v and f is non-monotonic. However, computation skew could exist because $f(v)$ could be very large for some value v.

As an opposite example of data skew without computation skew, a key-value system may encounter skewed partitioning and eventually suffer from significant tail latency if the input key-value pairs exhibit a power-law distribution. This is an example of pure data skew, because the latency comes from uneven data partitioning rather than anomalies in record-level processing time.

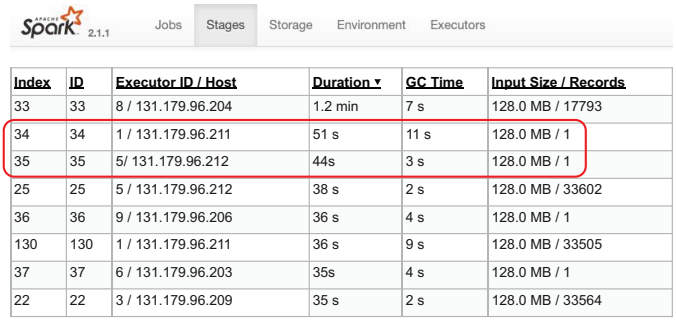
Computation skew and data skew do overlap. In the above review-processing example, if most long sentences appear in one single customer review, the execution would exhibit both data skew (due to the tail in the sentence distribution) and computation skew (since processing these long sentences would ultimately need much more resources than processing short sentences).

2.2 Apache Spark and Titian

Apache Spark [4] is a dataflow system that provides a programming model using Resilient Distributed Datasets (RDDs) which distributes the computations on a cluster of multiple worker nodes. Spark internally transforms a sequence of transformations (logical plan) into a directed acyclic graph (DAG) (physical plan). The physical plan consists of a sequence of *stages*, each of which is made up of pipelined transformations and ends at a shuffle. Using the DAG, Spark's scheduler executes each stage by running, on different nodes, parallel *tasks* each taking a *partition* of the stage's input data.

Titian [12] extends Spark to provide support for *data provenance*—the historical record of data movement through transformations. It accomplishes this by inserting *tracing agents* at the start and end of each stage. Each tracing agent assigns a unique identifier to each record consumed or produced by the stage. These identifiers are collected into agent tables that store the mappings between input and output records. In order to minimize the runtime tracing overhead, Titian asynchronously stores agent tables in Spark's *BlockManager* storage system using threads separated from those executing the application. Titian enables developers to *trace* the movement of individual data records forward or backward along the pipeline by joining these agent tables according to their input and output mappings.

However, Titian has limited usefulness in debugging computation skew. First, it cannot reason about computation latency for any individual record. In the event that a user is able to isolate a delayed output, Titian can leverage data lineage to identify the input records that contribute to the production of this output. However, it falls short of singling out input records that have the largest impact on application performance. Due to the lack of a fine-grained computation latency model (*e.g.,* record-level latency used in PerfDebug), Titian would potentially find a much greater number of input records that are correlated to the given delayed output, as measured in Section 5.5, while only a small fraction of them may actually contribute to the observed performance problem.



Index	ID	Executor ID / Host	Duration ▾	GC Time	Input Size / Records
33	33	8 / 131.179.96.204	1.2 min	7 s	128.0 MB / 17793
34	34	1 / 131.179.96.211	51 s	11 s	128.0 MB / 1
35	35	5/ 131.179.96.212	44s	3 s	128.0 MB / 1
25	25	5 / 131.179.96.212	38 s	2 s	128.0 MB / 33602
36	36	9 / 131.179.96.206	36 s	4 s	128.0 MB / 1
130	130	1 / 131.179.96.211	36 s	9 s	128.0 MB / 33505
37	37	6 / 131.179.96.203	35s	4 s	128.0 MB / 1
22	22	3 / 131.179.96.209	35 s	2 s	128.0 MB / 33564

Figure 2: An example screenshot of Spark's Web UI where each row represents task-level performance metrics. From left to right, the columns represent task identifier, the address of the worker hosting that task, running time of the task, garbage collection time, and the size (space and quantity) of input ingested by the task, respectively.

3 MOTIVATING SCENARIO

Suppose Alice acquires a 21GB dataset of movies and their user ratings. The dataset follows a strict format where each row consists of a movie ID prefix followed by comma-separated pairs of a user ID and a numerical rating (1 to 5). A small snippet of this dataset is as follows:

```
127142:2628763_4,2206105_4,802003_3,...
127143:1027819_3,872323_3,1323848_4,...
127144:1789551_3,1764022_5,1215225_5,...
```

Alice wishes to calculate the frequency of each rating in the dataset. To do so, she writes the two-stage Spark program shown in Figure 1. In this program, line 2 loads the dataset and lines 3-10 extract the substring containing ratings from each row and finds the distribution of ratings only for that row. Line 11 aggregates rating frequencies from each row to compute the distribution of ratings across the entire dataset. Alice runs her program using Apache Spark on a 10-node cluster with the given dataset and produces the final output in 1.2 minutes:

Rating	Count
1	99487661
2	217437722
3	663482151
4	771122507
5	524004701

At first glance, the execution may seem reasonably fast. However, Alice knows from past experience that a 20GB job such as this should typically complete in about 30 seconds. She looks at the Spark Web UI and finds that the first stage of her job amounts for over 98% of the total job time. Upon further investigation into Spark performance metrics as seen in Figure 2, Alice discovers that task 33 of this stage runs for 1.2 minutes while the rest of the tasks finish much early. The median task time is 11 seconds, but task 33 takes over 50% longer than the next slowest task (51 seconds) despite processing the same amount of input (128MB). She also notices that other tasks on the same machine perform normally, which

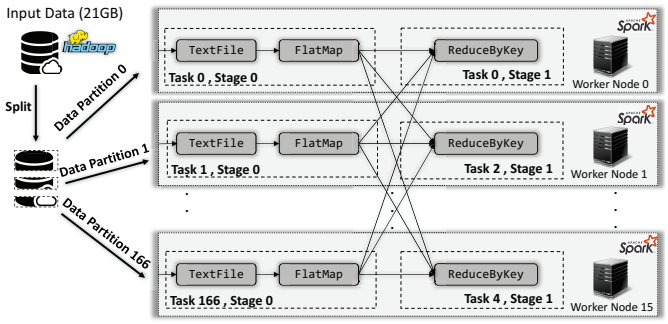


Figure 3: The physical execution of the motivating example by Apache Spark.

eliminates existence of a straggler due to hardware failures. This is a clear symptom of *computation skew* where the processing times for individual records differs significantly due to the interaction between record contents and the code processing these records.

To investigate which characteristics of the dataset caused her program to show disproportionate delays, Alice requests to see a subset of original input records accountable for the slow task. Since she has identified the slow task already, she may choose to inspect the data partition associated with that task manually. Figure 3 illustrates how this job is physically executed on the cluster. For example, Alice identifies task 1 of stage 0 as the slowest corresponding partition (*i.e.,* `Data Partition 1`). Since it contains 128MB of raw data and comprises millions of records, this manual inspection is infeasible.

As Alice has already identified the presence of computation skew, she enables PerfDebug's debugging mode. PerfDebug re-executes the applications and collects lineage as well as record-level latency information. After collecting this information, PerfDebug reports each output record's computation time (latency) and its corresponding slowest input:

Rating	Count	Latency (ms)	Slowest Input
1	99487661	28906	*"129707:..."*
2	217437722	28891	*"129707:..."*
3	663482151	28920	*"129707:..."*
4	771122507	28919	*"129707:..."*
5	524004701	28842	*"129707:..."*

Alice notices that the reported latencies are fairly similar for all output records. Furthermore, all five records report the same slowest delay-inducing input record with movie id `129707`. She inspects this specific input record and finds that it has far more ratings (91 million) than any other movie. Because Alice's code iterates through each rating to compute a per-movie rating count (lines 6-9 of Figure 1), this particular movie significantly slows down the task in which it appears. Alice suspects this unusually high rating count to be a data quality issue of some sort. As a result, she chooses to handle movie `129707` by removing it from the input dataset. In doing so, she finds that the removal of just one record decreases her program's execution time from 1.2 minutes to 31 seconds, which is much closer to her initial expectations.

Note that Alice's decision to remove movie `129707` is only one example of how she may choose to address this computation skew. PerfDebug is designed to detect and investigate computation skew, but appropriate remediations will vary depending on use cases and must be determined by the user.

4 APPROACH

When a sign of poor performance is seen, PerfDebug performs post-mortem debugging by taking in a Spark application and a dataset as inputs, and pinpoints the precise input record with the most impact on the execution time. Once PerfDebug is enabled, it is *fully automatic* and *does not require any human judgment*. Its approach is broken down into three steps. First, PerfDebug monitors coarse-grained performance metrics as a signal for computation skew. Second, PerfDebug re-executes the application on the entire input to collect lineage information and latency measurements. Finally, the lineage and latency information is combined to compute the time cost of producing individual output records. During this process, PerfDebug also assesses the impact of individual input records on the overall performance and keeps track of those with the highest impact on each output.

Sections 4.2 and 4.3 describe how to accumulate and attribute latencies to individual records throughout the multi-stage pipeline. This record level latency attribution differentiates PerfDebug from merely identifying the top-N expensive records within each stage because the mappings between input records and intermediate output records are not 1:1 in modern big data analytics. Operators such as `join`, `reduce`, and `groupByKey` generate n:1 mappings, while `flatmap` creates 1:n mappings. Thus, finding the top-N slow records from each stage may work on a single stage program but does not work for multi-stage programs with aggregation and data-split operators.

4.1 Performance Problem Identification

When PerfDebug is enabled on a Spark application, it identifies irregular performance by monitoring built-in performance metrics reported by Apache Spark. In addition to the running time of individual tasks, we utilize other constituent performance metrics, such as GC and serialization time, to identify irregular performance behavior. Several prior works, such as Yak [21], have highlighted the significant impact of GC on Big Data application performance. They also report that GC can even account for up to 50% of the total running time of such applications.

A high GC time can be observed due to two reasons: (1) millions of objects are being created within a task's runtime and (2) by the sheer size of individual objects created by UDFs while processing the input data. Similarly, a high serialization/deserialization time is usually induced for the same reasons. In both cases, high GC or serialization times are usually triggered by a specific characteristic of the input dataset. Referring back to our motivating scenario, a single row in the input dataset may comprise a large amount of information and lead to the creation of many objects. As a dataflow framework handles many such objects within a given task, both GC and serialization for that particular task soar. Since stage boundaries represent blocking operations (meaning that each task has to complete before moving to the next stage), the high volume of objects holds back the whole stage and leads to slower application performance. This effect can be propagated over multiple

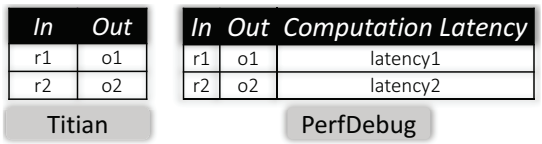


Figure 4: During program execution, PerfDebug also stores latency information in lineage tables comprising of an additional column of *ComputationLatency*.

stages as objects are passed around and repeatedly serialized and deserialized.

PerfDebug applies lightweight instrumentation to the Spark application by attaching a custom listener that observes performance metrics reported by Spark such as (1) task time, (2) GC time, and (3) serialization time. Note that PerfDebug is not limited to only these metrics and can be extended to support other performance measurements. For example, we can implement a custom listener to measure additional statistics described in [20] such as shuffle object serialization and deserialization times. This lightweight monitoring enables PerfDebug to avoid unnecessary instrumentation overheads for applications that do not exhibit computation skew. When an abnormality is identified, PerfDebug starts post-mortem debugging to enable deeper instrumentation at the record level and to find the root cause of performance delays. Alternatively, a user may manually identify performance issues and explicitly invoke PerfDebug's debugging mode.

4.2 Capturing Data Lineage and Latency

As the first step in post-mortem debugging, PerfDebug re-executes the application to collect latency (computation time of applying a UDF) of each record per stage in addition to data lineage information. For this purpose, PerfDebug extends Titian [12] and stores the per-record latency alongside record identifiers.

4.2.1 *Extending Data Provenance.*

PerfDebug adopts Titian [12] to capture record level input-output mapping. However, using off-the-shelf Titian is insufficient as it does not profile the compute time of each intermediate record which is crucial for locating the expensive input records. To enable performance profiling in addition to data provenance, PerfDebug extends Titian by measuring the time taken to compute each intermediate record and storing these latencies alongside the data provenance information. Titian captures data lineages by generating lineage tables that map the output record at one stage to the input of the next stage. Later, it constructs a complete lineage graph by joining the lineage tables, one at a time, across multiple stages. While Titian generates lineage tables, PerfDebug measures the computational latency of executing a chain of UDFs in a given stage on each record and appends it to the lineage tables in an additional column as seen in Figure 4. This extension produces a data provenance graph that exposes individual record computation times, which is used in Section 4.3 to precisely identify expensive input records.

Titian stores each lineage table in Spark's internal memory layer (abstracted as a file system through BlockManager) to lower runtime overhead of accessing memory. However, this approach is not feasible for post-mortem performance debugging as it hogs the memory available for the application and restricts the lifespan of lineage tables to the liveliness of a Spark session. PerfDebug supports post-mortem debugging in which a user can interactively debug anytime without compromising other applications by holding too many resources. To realize this, PerfDebug stores lineage tables externally using Apache Ignite [3] in an asynchronous fashion. As a persistent in-memory storage, Ignite decouples PerfDebug from the session associated to a Spark application and enables PerfDebug to support post-mortem debugging anytime in the future. We choose Ignite for its compatibility with Spark RDDs and efficient data access time, but PerfDebug can also be generalized to other storage systems.

Figure 5 demonstrates the lineage information collected by PerfDebug, shown as *In* and *Out*. Using this information, PerfDebug can execute backward tracing to identify the input records for a given output. For example, the output record `o3` under the *Out* column of ❸ `post-shuffle` can be traced backwards to `[i3,i8]` (*In* column of ❸) through the *Out* column of ❷ `pre-shuffle`. We further trace those intermediate records from *In* column of `pre-shuffle` back to the program inputs `[h1, h2, h3, h4, h5]` in the *Out* column of ❶ `HDFS`.

4.2.2 *Latency Measurement.*

Data provenance alone is insufficient for calculating the impact of individual records on overall application performance. As performance issues can be found both within stages (e.g., an expensive `filter`) and between stages (e.g., due to data skew in shuffling), PerfDebug tracks two types of latency. *Computation Latency* is measured from a chain of UDFs in dataflow operators such as `map` and `filter`, while *Shuffle Latency* is measured by timing shuffle-based operations such as `reduce` and distributing this measurement based on input-output ratios.

For a given record r, the total time to execute all UDFs of a specific stage, $StageLatency(r)$ is computed as:

$$\texttt{StageLatency(r)} = \texttt{ComputationLatency(r)} + \texttt{ShuffleLatency(r)}$$

Computation Latency. As described in Section 2, a stage consists of multiple pipelined transformations that are applied to input records to produce the stage output. Each transformation is in turn defined by an operator that takes in a UDF. To measure computation latency, PerfDebug wraps every non-shuffle UDF in a timing function that measures the time span of that UDF invocation for each record. We define non-shuffle UDFs as those passed as inputs to operators that do not trigger a shuffle such as *flatmap*. Since the pipelined transformations in a stage are applied sequentially on each record, PerfDebug calculates the computation latency $ComputationLatency(r)$ of record r by adding the execution times of each UDF applied to r within the current stage:

$$\texttt{ComputationLatency(r)} = \sum_{\texttt{f} \in \texttt{UDF}} \texttt{Time(f,r)}$$

For example, consider the following program:

```
1 val f1 = (x: Int) => List(x, x*2) // 50ms
2 val f2 = (x: Int) => x < 100 // 10ms, 20ms
3 integerRdd.flatMap(f1).filter(f2).collect()
```

When executing this program for a single input *42*, we obtain outputs of *42* and *84*. Suppose PerfDebug observes that *f1(42)* takes 50 milliseconds, while *f2(42)* and *f2(84)* take 10 and 20 milliseconds

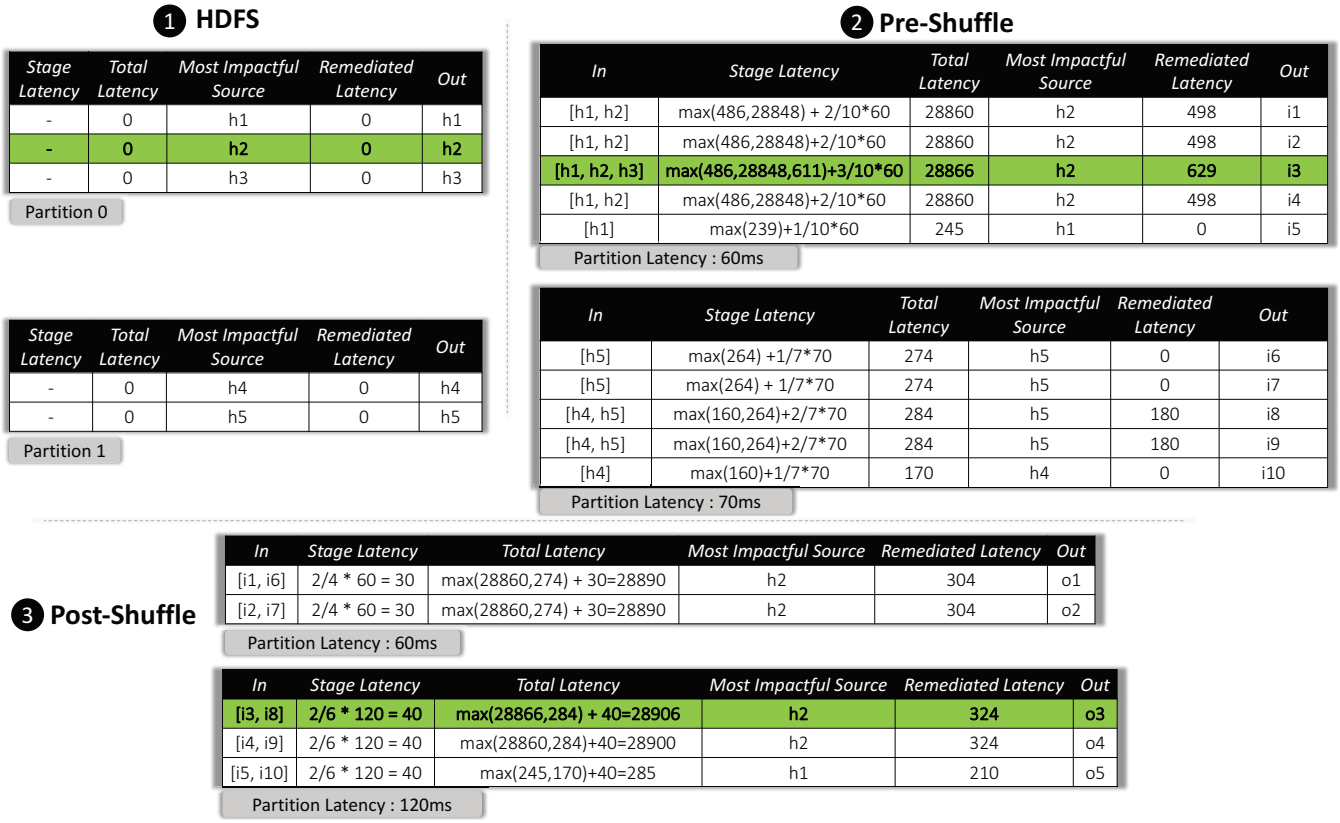


Figure 5: The snapshots of lineage tables collected by PerfDebug. ❶, ❷, and ❸ illustrate the physical operations and their corresponding lineage tables in sequence for the given application. In the first step, PerfDebug captures the *Out*, *In*, and *Stage Latency* columns, which represent the input-output mappings as well as the stage-level latencies per record. During output latency computation, PerfDebug calculates three additional columns (*Total Latency*, *Most Impactful Source*, and *Remediated Latency*) to keep track of cumulative latency, the ID of the original input with the largest impact on *Total Latency*, and the estimated latency if the most impactful record did not impact application performance.

respectively. PerfDebug computes the computation latency for the first output, *42*, as 50 + 10 = 60 milliseconds. Similarly, the second output, *84*, has a computation latency of 50 + 20 = 70 milliseconds.

In stages preceding a shuffle, multiple input records may be pre-aggregated to produce a single output record. In the ❷-`Pre-Shuffle` lineage table shown in Figure 5, the *In* column and the left term in the *StageLatency* column reflect these multiple input identifiers and computation latencies. As the Spark application's execution proceeds through each stage, PerfDebug captures *StageLatency* for each output record per stage and includes it into the lineage tables under the *Stage Latency* column as seen in Figure 5. These lineage tables are stored in PerfDebug's Ignite storage where each table encodes the computation latency of each record and the relationship of that record to the output records of the previous stage.

Shuffle Latency. In Spark, a shuffle at a stage boundary comprises of two steps: a pre-shuffle step and a post-shuffle step. In the pre-shuffle step, each task's output data is sorted or aggregated and then stored in the local memory of the current node. We measure the time it takes to perform the pre-shuffle step on the whole partition as *pre-shuffle latency*. In the post-shuffle step, a node in the next stage fetches this remotely stored data from individual nodes and sorts (or aggregates) it again. Because of this distinction, PerfDebug's shuffle latency is categorized into pre-shuffle and post-shuffle estimations.

As both pre- and post- shuffle operations are atomic and performed in batches over each partition, we estimate the latency of an individual output record in a pre-shuffle step by (1) measuring the proportion of the input records consumed by the output record and then (2) multiplying it with the total shuffle time of that partition.

$$\texttt{ShuffleLatency}(r) = \frac{|\texttt{Inputs}_r|}{|\texttt{Inputs}|} * \texttt{PartitionLatency}(\texttt{stage}_r)$$

$stage_r$ represents the stage of the record r, $|Inputs|$ is the size of a partition, and $|Input_r|$ is the size of input consumed by output r. For example, the top most lineage table under ❷-`pre-shuffle` in Figure 5 has a pre-shuffle latency of 60ms. Because output `i1` is computed from two of the partition's ten inputs, `ShuffleLatency(i1)` is

equal to two tenths of partition latency *i.e.,* $\frac{2}{10} * 60$. Similarly, output `i3` is computed from three inputs so its shuffle latency is $\frac{3}{10} * 60$.

4.3 Expensive Input Isolation

To identify the most expensive input for a given application and dataset, PerfDebug analyzes data provenance and latency information from Section 4.2 and calculates three values for each output record: (1) the total latency of the output record, (2) the input record that contributes most to this latency (*most impactful source*), and (3) the expected output latency if that input record had zero latency or otherwise did not affect application performance (*remediated latency*). Once calculated, PerfDebug groups these values by their most impactful source and compares each input's maximum latency with its maximum remediated latency to identify the input with the most impact on application performance.

Output Latency Calculation. PerfDebug estimates the total latency for each output record as a sum of associated stage latencies established by data provenance based mappings. By leveraging the data lineage and latency tables collected earlier, it computes the latency using two key insights:

- In dataflow systems, records for a given stage are often computed in parallel across several tasks. Assuming all inputs for a given record are computed in this parallel manner, the time required for all the inputs to be made available is at least the time required for the final input to arrive. This corresponds to the *maximum* of the dependent input latencies.
- A record can only be produced when all its inputs are made available. Thus, the total latency of any given record must be at least the sum of its stage-specific individual record latency, described in Section 4.2, and the slowest latency of its inputs, described above.

The process of computing output latencies is inspired by the forward tracing algorithm from Titian, starting from the entire input dataset.[1] PerfDebug recursively joins lineage tables to construct input-output mappings across stages. For each recursive join in the forward trace, PerfDebug computes the accumulated latency $TotalLatency(r)$ of an output r by first finding the latency of the slowest input ($SlowestInputLatency(r)$) among the inputs from the preceding stage on which the output depends upon, and then *adding* the stage-specific latency $StageLatency(r)$ as described in Section 4.2:

$$\texttt{SlowestInputLatency(r)} = \max(\forall \texttt{i} \in \texttt{Inputs}_{\texttt{prev\_stage}} : \texttt{TotalLatency(i)})$$

$$\texttt{TotalLatency(r)} = \texttt{SlowestInputLatency(r)} + \texttt{StageLatency(r)}$$

Once *TotalLatency* is calculated for each record at each step of recursive join, it is added in the corresponding lineage tables under the new column, *Total Latency*. For example, the output record `i1` in ❷-`Pre-Shuffle` lineage table of Figure 5 has two inputs from the previous stage, `h1` and `h2` with their total latencies of 486ms and 28848ms respectively. Therefore, its $SlowestInputLatency(i1)$ is the maximum of 70 and 28848 which is then added to its $ShuffleLatency(i1) = \frac{2}{10} * 60$ms, making the total latency of `i1` 28860ms.

Tracing Input Records. Based on the output latency, a user can select an output and use PerfDebug to perform a backward trace as described in Section 4.2. However, the input isolated through this technique may not be precise as it relies solely on data lineage. For example, Alice uses PerfDebug to compute the latency of individual output records, shown in Figure 5. Next, Alice isolates the slowest output record, `o3`. Finally, she uses PerfDebug to trace backward and identify the inputs for `o3`. Unfortunately, all five inputs contribute to `o3`. Because there is only one significant delay-inducing input record (`h2`) which contributes to `o3`'s latency, the lineage-based backward trace returns a super-set of delay-inducing inputs and achieves a low precision of 20%.

Tracking Most Impactful Input. To improve upon the low precision of lineage-based backward traces, PerfDebug propagates record identifiers during output latency computation and retains the input records with the most impact on an output's latency. We define the *impact* of an input record as the difference between the maximum latency of all associated output records in program executions with and without the given input record. Intuitively, this represents the degree to which a delay-inducing input is a bottleneck for output record computation.

To support this functionality, PerfDebug takes an approach inspired by the Titian-P variant described in [12]. In Titian-P (referred to as Titian Piggy Back), lineage tables are joined together as soon as the lineage table of the next stage is available during a program execution. This obviates the need for a backward trace as each lineage table contains a mapping between the intermediate or final output and the original input, but also requires additional memory to retain a list of input identifiers for each intermediate or final output record. PerfDebug's approach differs in that it retains only a single input identifier for each intermediate or final output record. As such, its additional memory requirements are constant per output record and do not increase with larger input datasets. Using this approach, PerfDebug is able to compute a predefined backward trace with minimal memory overhead while avoiding the expensive computation and data shuffles required for a backward trace.

As described earlier, the latency of a given record is dependent on the maximum latency of its corresponding input records. In addition to this latency, PerfDebug computes two additional fields during its output latency computation algorithm to easily support debugging queries about the impact of a particular input record on the overall performance of an application.

- **Most Impactful Source**: the identifier of the input record deemed to be the top contributor to the latency of an intermediate or final output record. We pre-compute this so that debugging queries do not need a backward trace and can easily identify the single most impactful record for a given output record.
- **Remediated Latency**: the expected latency of an intermediate or final output record if *Most Impactful Source* had zero latency or otherwise did not affect application performance. This is used to quantify the impact of the *Most Impactful Source* on the latency of the output record.

[1] PerfDebug leverages lineage-based backward trace to remove inputs that do not contribute to program outputs while computing output latencies.

As with *TotalLatency*, these fields are inductively updated (as seen in Figure 5) with each recursive join when computing output latency. During recursive joins, *Most Impactful Source* field becomes the *Most Impactful Source* of the input record possessing the highest *TotalLatency*, similar to an `argmax` function. *Remediated Latency* becomes the current record's *StageLatency* plus the maximum latency over all input records except the *Most Impactful Source*. For example, the output `o3` has the highest *TotalLatency* with the *most impactful source* of `h2`. This is reported based on the reasoning that, if we remove `h2`, the latencies of input `i3` and `i8` drop the most compared to removing either `h1` or `h3`.

In addition to identifying the most impactful record for an individual program output, PerfDebug can also use these extended fields to identify input records with the largest impact on overall application performance. This is accomplished by grouping the output latency table by *Most Impactful Source* and finding the group with the largest difference between its maximum *TotalLatency* and maximum *Remediated Latency*. In the case of Figure 5, input record `h2` is chosen because its difference (28906ms - 324ms) is greater than that of `h1` (285ms - 210ms).

5 EXPERIMENTAL EVALUATION

Our applications and datasets are described in Table 1. Our inputs come from industry-standard PUMA benchmarks [5], public institution datasets [22], and prior work on automated debugging of big data analytics [8]. Case studies described in Sections 5.3, 5.2, and 5.4 demonstrate when and how a user may use PerfDebug. PerfDebug provides diagnostic capability by identifying records attributed to significant delays and leaves it to the user to resolve the performance problem, e.g., by re-engineering the analytical program or refactoring UDFs.

5.1 Experimental Setup

All case studies are executed on a cluster consisting of 10 worker nodes and a single master, all running CentOS 7 with a network speed of 1000 Mb/s. The master node has 46GB available RAM, a 4-core 2.40GHz CPU, and 5.5TB available disk space. Each worker node has 125GB available RAM, a 8-core 2.60GHz CPU, and 109GB available disk space.

Throughout our experiments, each Spark Executor is allocated 24GB of memory. Apache Hadoop 2.2.0 is used to host all datasets on HDFS (replication factor 2), with the master configured to run only the NameNode. Apache Ignite 2.3.0 servers with 4GB of memory are created on each worker node, for a total of 10 ignite servers. PerfDebug creates additional Ignite client nodes in the process of collecting or querying lineage information, but these do not store data or participate in compute tasks. Before running each application, the Ignite cluster memory is cleared to ensure that previous experiments do not affect measured application times.

5.2 Case Study A: NYC Taxi Trips

Alice has 27GB of data on 173 million taxi trips in New York [22], where she needs to compute the average cost of a taxi ride for each borough. A borough is defined by a set of points representing a polygon. A taxi ride starts in a given borough if its starting coordinate lies within the polygon defined by a set of points, as computed via the ray casting algorithm. This program is written as a two-stage Spark application shown in Figure 6.

```
1  val avgCostPerBorough = lines.map { s =>
2    val arr = s.split(',')
3    val pickup = new Point(arr(11).toDouble,
4                           arr(10).toDouble)
5    val tripTime = arr(8).toInt
6    val tripDistance = arr(9).toDouble
7    val cost = getCost(tripTime, tripDistance)
8    val b = getBorough(pickup)
9    (b, cost)}
10   .aggregateByKey((0d, 0))(
11   {case ((sum, count), next) => (sum + next, count+1)},
12   {case ((sum1, count1), (sum2, count2)) =>
         (sum1+sum2,count1+count2)}
13 ).mapValues({case (sum, count) => sum.toDouble/count}).collect()
```

Figure 6: A Spark application computing the average cost of a taxi ride for each borough.

Alice tests this application on a small subset of data consisting of 800,000 records in a single 128MB partition, and finds that the application finishes within 8 seconds. However, when she runs the same application on the full data set of 27GB, it takes over 7 minutes to compute the following output:

Borough	Trip Cost($)
1	56.875
2	67.345
3	97.400
4	30.245

This delay is higher than her expectation, since this Spark application should perform data-parallel processing and computation for each borough is independent of other boroughs. Thus, Alice turns to the Spark Web UI to investigate this increase in the job execution time. She finds that the first stage accounts for almost all of the job's running time, where the median task takes 14 seconds only, while several tasks take more than one minute. In particular, one task runs for 6.8 minutes. This motivates her to use PerfDebug. She enables a post-mortem debugging mode and resubmits her application to collect lineage and latency information. This collection of lineage and latency information incurs 7% overhead, after which PerfDebug reports the computation latency for each output record as shown below. In this output, the first two columns are the outputs generated by the Spark application and the last column, Latency (ms), is the total latency calculated by PerfDebug for each individual output record.

Borough	Trip Cost($)	Latency (ms)
1	56.875	3252
2	67.345	2481
3	97.400	2285
4	30.245	9448

Alice notices that borough #4 is much slower to compute than other boroughs. She uses PerfDebug to trace lineage for borough #4 and finds that the output for borough #4 comes from 1001 trip records in the input data, which is less than 0.0006% of the entire dataset. To understand the performance impact of input data for borough #4, Alice filters out the 1001 corresponding trips and reruns the application for the remaining 99.9994% of data. She finds that the application finishes in 25 seconds, significantly faster than the original 7 minutes. In other words, PerfDebug helped Alice

#	Subject Programs	Source	Input Size	# of Ops	Program Description	Input Data Description
S1	Movie Ratings	PUMA	21 GB	2	Computes the number of ratings per rating score (1-5), using `flatMap` and `reduceByKey`.	Movies with a list of corresponding rater and rating pairs
S2	Taxi	NYC Taxi and Limousine Commission	27 GB	3	Compute the average cost of taxi trips originating from each borough, using `map` and `aggregateByKey`.	Taxi trips defined by fourteen fields, including pickup coordinates, drop-off coordinates, trip time, and trip distance.
S3	Weather Analysis	Custom	15 GB	3	For each (1) state+month+day and (2) state+year: compute the median snowfall reading, using `flatMap`, `groupByKey`, and `map`.	Daily snowfall measurements per zipcode, in either feet or millimeters.

Table 1: Subject programs with input datasets.

```
val pairs = lines.flatMap { s =>
  val arr = s.split(',')
  val state = zipCodeToState(arr(0))
  val fullDate = arr(1)
  val yearSplit = fullDate.lastIndexOf("/")
  val year = fullDate.substring(yearSplit+1)
  val monthdate =
       fullDate.substring(0, yearSplit)
  val snow = arr(2).toFloat
  Iterator( ((state, monthdate), snow),
            ((state , year) , snow) )}
val medianSnowFall =
   pairs.groupByKey()
        .mapValues(median).collect()
```

Figure 7: A weather data analysis application

discover that removing 0.0006% of the input data can lead to an almost 16X improvement in application performance. Upon further inspection of the delay-inducing input records, Alice notes that while the polygon for most boroughs is defined as an array of 3 to 5 points, the polygon for borough #4 consists of 20004 points in a linked list—i.e., a neighborhood with complex, winding boundaries, thus leading to considerably worse performance in the ray tracing algorithm implementation.

We note that currently there are no easy alternatives for identifying delay-inducing records. Suppose that a developer uses a classical automated debugging method in software engineering such as delta debugging (DD) [28] to identify the subset of delay-inducing records. DD divides the original input into multiple subsets and uses a binary-search like procedure to repetitively rerun the application on different subsets. Identifying 1001 records out of 173 million would require at least 17 iterations of running the application on different subsets. Furthermore, without an intelligent way of dividing the input data into multiple subsets based on the borough ID, it would not generate the same output result.

Furthermore, although the Spark Web UI reports which task has a higher computation time than other tasks, the user may not be able to determine which input records map to the delay-causing partition. Each input partition could map to millions of records, and the 1001 delay-inducing records may be spread over multiple partitions.

5.3 Case Study B: Weather

Alice has a 15GB dataset consisting of 470 million weather data records and she wants to compute the median snowfall reading for each state on any day or any year separately by writing the program in Figure 7.

Alice runs this application on the full dataset, with PerfDebug's performance monitoring enabled. The application takes 9.3 minutes to produce the following output. She notices that there is a straggler task in the second stage that ran for 4.4 minutes, where 2 minutes are attributed to garbage collection time. In contrast, the next slowest task in the same stage ran for only 49 seconds, which is 5 times faster than the straggler task. After identifying this computation skew, PerfDebug re-executes the program in the post-mortem debugging mode and produces the following results along with the computation latency for each output record, shown on the third column:

(State,Date) or (State,Year)	Median Snowfall	Latency (ms)
(28,2005)	3038.3416	1466871
(21,4/30)	2035.3096	89500
(27,9/3)	2033.828	89500
(11,1980)	3031.541	67684
(36,3/18)	3032.2273	67684
...	...	...

Looking at the output from PerfDebug, Alice realizes that producing the output `(28,2005)` is a bottleneck and uses PerfDebug to trace the lineage of this output record. It finds that approximately 45 million input records, in other words almost 10% of the input, map to the key `(28, 2005)`, causing data skew in the intermediate results. PerfDebug reports that the majority of this latency comes from shuffle latency, as opposed to the computation time taken in applying UDFs to the records. Based on this symptom of the performance delays, Alice replaces the `groupByKey` operator with the more efficient `aggregateByKey` operator. She then runs her new program, which now completes in 45 seconds. In other words, PerfDebug aided in the diagnosis of performance issues, which resulted in a simple application logic rewrite with 11.4X performance improvement.

5.4 Case Study C: Movie Ratings

The Movie Ratings application is described in Section 3 as a motivating example. The numbers reported in Section 3 are the actual numbers found through our evaluation. To avoid redundancy, this subsection quickly summarizes the evaluation results from the case study of this application. The original job time for 21GB data takes 1.2 minutes, which is much longer than what the user would normally expect. PerfDebug reports task-level performance metrics such as execution time that indicate computation skew in the first stage. Collecting latency information during the job execution incurs 8.3% instrumentation overhead. PerfDebug then analyzes the

collected lineage and latency information and reports the computation latency for producing each output record. Upon recognizing that all output records have the same slowest input, which has an abnormally high number of ratings, Alice decides to remove the single culprit record contributing the most delay. By doing so, the execution time drops from 1.2 minutes to 31 seconds, achieving 1.5X performance gain.

5.5 Accuracy and Instrumentation Overhead

For the three applications described below, we use PerfDebug to measure the accuracy of identifying delay-inducing records, the improvement in precision over a data lineage trace implemented by Titian, and the performance overhead in comparison to Titian. The results for these three applications indicate the following: (1) PerfDebug achieves 100% accuracy in identifying delay-inducing records where delays are injected on purpose for randomly chosen records; (2) PerfDebug achieves 10^2 to 10^6 orders of magnitude improvement in precision when identifying delay-inducing records, compared to Titian; and (3) PerfDebug incurs an average overhead of 30% for capturing and storing latency information at the fine-grained record level, compared to Titian.

The three applications we use for evaluation are *Movie Ratings*, *College Student*, and *Weather Analysis*. *Movie Ratings* is identical to that used in Section 3, but on a 98MB subset of input consisting of 2103 records. *College Student* is a program that computes the average student age by grade level using `map` and `groupByKey` on a 187MB dataset of five million records, where each record contains a student's name, sex, age, grade, and major. Finally, *Weather Analysis* is similar to the earlier case study in Section 5.3 but instead computes the delta between minimum and maximum snowfall readings for each key, and is executed on a 52MB dataset of 2.1 million records. All three applications described in this section are executed on a single MacBook Pro (15-inch, Mid-2014 model) running macOS 10.13.4 with 16GB RAM, a 2.2GHz quad-core Intel Core i7 processor, and 256GB flash storage.

Identification Accuracy. Inspired by automated fault injection in the software engineering research literature, we inject artificial delays for processing a particular subset of intermediate records by modifying application code. Specifically, we randomly select a single input record r and introduce an artificial delay of ten seconds for r using a `Thread.sleep()`. As such, we expect r to be the slowest input record. This approach of inducing faults (or delays) is inspired by *mutation testing* in software engineering, where code is modified to inject known faults and then the fault detection capability of a newly proposed testing or debugging technique is measured by counting the number of detected faults. This method is widely accepted as a reliable evaluation criteria [13, 14].

For each application, we repeat this process of randomly selecting and delaying a particular input record for ten trials and report the average accuracy in Table 2. PerfDebug accurately identifies the slowest input record with 100% accuracy for all three applications.

Precision Improvement. For each trial in the previous section, we also invoke Titian's backward tracing on the output record with the highest computation latency. We measure precision improvement by dividing the number of delay-inducing inputs reported by PerfDebug by the total number of inputs mapping to the output record with the highest latency reported by Titian. We then average these precision measurements across all ten trials, shown in Table 2. PerfDebug isolates the delay-inducing input with 10^2-10^6 order better precision than Titian due to its ability to refine input isolation based on cumulative latency per record. This fine-grained latency profiling enables PerfDebug to slice the contributions of each input record towards the computational latency of a given output record substantially to identify a subset of inputs with the most significant influence on performance delay.

Benchmark	Accuracy	Precision Improvement	Overhead
Movie Ratings	100%	2102X	1.04X
College Student	100%	1250000X	1.39X
Weather Analysis	100%	294X	1.48X
Average	100%	417465X	1.30X

Table 2: Identification Accuracy of PerfDebug and instrumentation overheads compared to Titian, for the subject programs described in Section 5.5.

Instrumentation Overhead. To measure instrumentation overhead, we execute each application ten times for both PerfDebug and Titian without introducing any artificial delay. To avoid unnecessary overheads, the Ignite cluster described earlier is created only when using PerfDebug. The resulting performance multipliers are shown in Table 2. We observe that the performance overhead of PerfDebug compared to Titian ranges from 1.04X to 1.48X. Across all applications, PerfDebug's execution times average 1.30X times as long as Titian's. Titian reports an overhead of about 30% compared to Apache Spark [12]. PerfDebug introduces additional overhead because it instruments every invocation of a UDF to capture and store the record level latency. However, such fine-grained profiling differentiates PerfDebug from Titian in terms of its ability to isolate expensive inputs. PerfDebug's overhead to identify a delay inducing record is small compared to the alternate method of trial and error debugging, which requires multiple execution of the original program.

6 RELATED WORK

Kwon et al. present a survey of various sources of performance skew in [16]. In particular, they identify data-related skews such as *expensive record* skew and *partitioning* skew. Many of the skew sources described in the survey influenced our definition of *computation skew* and motivated potential use cases of PerfDebug.

Ernest [26], ARIA [27], and Jockey [7] model job performance by observing system and job characteristics. These systems as well as Starfish [10] construct performance models and propose system configurations that either meet the budget or deadline requirements. However, these systems focus on performance prediction rather than performance debugging. Furthermore, none of these systems focus on computation skew, nor do they provide visibility into fine-grained computation at the individual record level.

PerfXplain [15] is a debugging tool that allows users to compare two similar jobs under different configurations through a simple

query language. When comparing similar jobs or tasks, PerfXplain automatically generates an explanation using the differences in collected metrics. However, PerfXplain does not take into account the computational latency of individual records and thus does not report how performance delays could be attributed to a subset of input records.

Sarawagi et al. [25] propose a discovery-driven exploration approach that preemptively analyzes data for statistical anomalies and guides user analysis by identifying exceptions at various levels of data cube aggregations. Later work [24] also automatically summarizes these exceptions to highlight increases or drops in aggregate metrics. However, both works focus on OLAP operations such as rollup and drilldown which are insufficient for processing the complex mappings between input and output records in a DISC application. In addition, both of these works do not directly address debugging of performance skews.

SkewTune [17] is an automatic skew mitigation approach which elastically redistributes data based on estimated time to completion for each worker node. It primarily focuses on data skew issues and provides automatic re-balancing of data rather than providing performance debugging assistance. As a result, application developers cannot use SkewTune to answer performance debugging queries about their jobs nor analyze performance or latency at the record level.

Titian implements data provenance within Apache Spark and is used as a foundation for PerfDebug. Other systems such as RAMP [11] and Newt [19] also provide data provenance within dataflow systems. However, none of these systems measure the performance latency of individual data records alongside their data provenance. Extensions of these data provenance tools include use cases such as interactive debugging [9] and automated fault isolation [8], but PerfDebug is unique in that it provides visibility into performance issues at a fine-grained record level and it automates the diagnosis of interaction between individual records and their influence on the overall application performance.

7 CONCLUSION AND FUTURE WORK

PerfDebug is the first automated performance debugging tool to diagnose the root cause of performance delays induced by interaction between data and application code. PerfDebug automatically reports the symptoms of *computation skew*—abnormally high computation costs for a small subset of data records—in terms of garbage collection, serialization, and task execution time. It combines a novel latency estimation technique with an existing data provenance tool. Based on this novel notion of record-level latency, PerfDebug isolates delay-inducing inputs automatically.

On average, we find that PerfDebug can detect injected delays with a high accuracy (100%), improves the precision of isolating delay-inducing records by orders of magnitude (10^2 to 10^6), and incurs a reasonable instrumentation overhead (4% to 48% extra) compared to an existing data provenance technique, Titian. Our case studies show a user may achieve performance improvement of 16X by simply removing the most expensive record from the input data or through a simple code rewrite by investigating delay-inducing input records reported by PerfDebug. In the future, we plan to expand the scope of instrumentation and performance debugging queries to account for performance delays caused by both framework-level configurations and interaction between data and application code.

Acknowledgments. We thank the anonymous reviewers for their comments and Tim Harris for his guidance as a shepherd. The participants of this research are in part supported by NSF grants CCF-1723773, CCF-1764077, CCF-1527923, CCF-1460325, CNS-1613023, CNS-1703598, CNS-1763172, ONR grants N00014-16-1-2913, N00014-18-1-2037, Intel CAPA grant, Samsung grant, and Google PhD Fellowship.

REFERENCES

[1] 2015. Out of memory error in customer review processing. https://stackoverflow.com/questions/20247185.
[2] 2018. *Apache Hadop.* https://hadoop.apache.org/
[3] 2018. *Apache Ignite.* https://ignite.apache.org/
[4] 2018. *Apache Spark.* https://spark.apache.org/
[5] Faraz Ahmad, Seyong Lee, Mithuna Thottethodi, and TN Vijaykumar. 2012 . TRECE-12-11. *Puma: Purdue mapreduce benchmarks suite.* Technical Report. School of Electrical and Computer Engineering, Purdue University. https://engineering.purdue.edu/~puma/datasets.htm
[6] Lu Fang, Khanh Nguyen, Guoqing Xu, Brian Demsky, and Shan Lu. 2015. Interruptible Tasks: Treating Memory Pressure As Interrupts for Highly Scalable Data-parallel Programs. In *Proceedings of the 25th Symposium on Operating Systems Principles*. 394–409.
[7] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. 2012. Jockey: Guaranteed Job Latency in Data Parallel Clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems (EuroSys '12)*. ACM, New York, NY, USA, 99–112. https://doi.org/10.1145/2168836.2168847
[8] Muhammad Ali Gulzar, Matteo Interlandi, Xueyuan Han, Mingda Li, Tyson Condie, and Miryung Kim. 2017. Automated debugging in data-intensive scalable computing. In *Proceedings of the 2017 Symposium on Cloud Computing*. ACM, 520–534.
[9] Muhammad Ali Gulzar, Matteo Interlandi, Seunghyun Yoo, Sai Deep Tetali, Tyson Condie, Todd D. Millstein, and Miryung Kim. 2016. BigDebug: Debugging Primitives for Interactive Big Data Processing in Spark. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*. 784–795. https://doi.org/10.1145/2884781.2884813
[10] Herodotos Herodotou, Harold Lim, Gang Luo, Nedyalko Borisov, Liang Dong, Fatma Bilgen Cetin, and Shivnath Babu. 2011. Starfish: A Self-tuning System for Big Data Analytics. In *In CIDR*. 261–272.
[11] Robert Ikeda, Hyunjung Park, and Jennifer Widom. 2011. Provenance for generalized map and reduce workflows. In *In CIDR*. 273–283.
[12] Matteo Interlandi, Ari Ekmekji, Kshitij Shah, Muhammad Ali Gulzar, Sai Deep Tetali, Miryung Kim, Todd Millstein, and Tyson Condie. 2018. Adding data provenance support to Apache Spark. *The VLDB Journal* 27, 5 (01 Oct 2018), 595–615. https://doi.org/10.1007/s00778-017-0474-5
[13] Y. Jia and M. Harman. 2011. An Analysis and Survey of the Development of Mutation Testing. *IEEE Transactions on Software Engineering* 37, 5 (Sep. 2011), 649–678. https://doi.org/10.1109/TSE.2010.62
[14] René Just, Darioush Jalali, Laura Inozemtseva, Michael D. Ernst, Reid Holmes, and Gordon Fraser. 2014. Are Mutants a Valid Substitute for Real Faults in Software Testing?. In *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE 2014)*. ACM, New York, NY, USA, 654–665. https://doi.org/10.1145/2635868.2635929
[15] Nodira Khoussainova, Magdalena Balazinska, and Dan Suciu. 2012. PerfXplain: Debugging MapReduce Job Performance. *Proc. VLDB Endow.* 5, 7 (March 2012), 598–609. https://doi.org/10.14778/2180912.2180913
[16] Yongchul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2011. A study of skew in mapreduce applications. In *In the 5th Open Cirrus Summit*. Moskow.
[17] YongChul Kwon, Magdalena Balazinska, Bill Howe, and Jerome Rolia. 2012. SkewTune: Mitigating Skew in Mapreduce Applications. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data (SIGMOD '12)*. ACM, New York, NY, USA, 25–36. https://doi.org/10.1145/2213836.2213840
[18] YongChul Kwon, Kai Ren, Magdalena Balazinska, and Bill Howe. 2013. Managing Skew in Hadoop. *IEEE Data Eng. Bull.* 36 (2013), 24–33.
[19] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable Lineage Capture for Debugging DISC Analytics. In *Proceedings of the 4th Annual Symposium on Cloud Computing (SOCC '13)*. ACM, New York, NY, USA, Article 17, 15 pages. https://doi.org/10.1145/2523616.2523619

[20] Khanh Nguyen, Lu Fang, Christian Navasca, Guoqing Xu, Brian Demsky, and Shan Lu. 2018. Skyway: Connecting Managed Heaps in Distributed Big Data Systems. In *Proceedings of the Twenty-Third International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '18)*. ACM, New York, NY, USA, 56–69. https://doi.org/10.1145/3173162.3173200

[21] Khanh Nguyen, Lu Fang, Guoqing Xu, Brian Demsky, Shan Lu, Sanazsadat Alamian, and Onur Mutlu. 2016. Yak: A High-Performance Big-Data-Friendly Garbage Collector. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*. USENIX Association, Savannah, GA, 349–365. https://www.usenix.org/conference/osdi16/technical-sessions/presentation/nguyen

[22] NYC Taxi and Limousine Commission. [n.d.]. NYC Taxi Trip Data 2013 (FOIA/-FOIL). https://archive.org/details/nycTaxiTripData2013. Accessed: 2019-05-31.

[23] Kay Ousterhout, Ryan Rasti, Sylvia Ratnasamy, Scott Shenker, and Byung-Gon Chun. 2015. Making Sense of Performance in Data Analytics Frameworks. In *12th USENIX Symposium on Networked Systems Design and Implementation (NSDI 15)*. USENIX Association, Oakland, CA, 293–307. https://www.usenix.org/conference/nsdi15/technical-sessions/presentation/ousterhout

[24] Sunita Sarawagi. 1999. Explaining Differences in Multidimensional Aggregates. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 42–53. http://dl.acm.org/citation.cfm?id=645925.671500

[25] Sunita Sarawagi, Rakesh Agrawal, and Nimrod Megiddo. 1998. Discovery-driven Exploration of OLAP Data Cubes. In *In Proc. Int. Conf. of Extending Database Technology (EDBT'98*. Springer-Verlag, 168–182.

[26] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. 2016. Ernest: Efficient Performance Prediction for Large-scale Advanced Analytics. In *Proceedings of the 13th Usenix Conference on Networked Systems Design and Implementation (NSDI'16)*. USENIX Association, Berkeley, CA, USA, 363–378. http://dl.acm.org/citation.cfm?id=2930611.2930635

[27] Abhishek Verma, Ludmila Cherkasova, and Roy H. Campbell. 2011. ARIA: Automatic Resource Inference and Allocation for Mapreduce Environments. In *Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC '11)*. ACM, New York, NY, USA, 235–244. https://doi.org/10.1145/1998582.1998637

[28] Andreas Zeller. 1999. Yesterday, My Program Worked. Today, It Does Not. Why?. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-7)*. Springer-Verlag, Berlin, Heidelberg, 253–267. http://dl.acm.org/citation.cfm?id=318773.318946