Tracing Distributed Data Stream Processing Systems

Zoltán Zvara*†, Péter G.N. Szabó*‡, Gábor Hermann*§, András Benczúr*¶
* Hungarian Academy of Sciences
Institute for Computer Science and Control (MTA SZTAKI)
13-17 Kende u., H-1111 Budapest, Hungary
† Email: zoltan.zvara@sztaki.mta.hu
‡ Email: peter.szabo@sztaki.mta.hu
§ Email: gabor.hermann.dms@sztaki.mta.hu
¶ Email: andras.benczur@sztaki.mta.hu

***Abstract*—Interconnected stream processing components of distributed compute topologies suffer from a variety of problems on shared-nothing clusters. A wide array of issues remain hard to identify on the underlying data processing systems due to the irregular characteristics of incoming data. Moreover, bottlenecks and data anomalies propagate unexpectedly through system boundaries. Developers and system administrators spend countless hours on identifying these problems and debugging streaming applications. By the tracing of individual input records, issues caused by outliers become tractable even in complex topologies. Existing tracing solutions are only suitable for single-system batch workloads, and solely provide debugging capabilities in most cases. We present a distributed, platform-wide tracing design and framework for production streaming applications that helps to solve a variety of optimization problems in real-time. We provide a prototype implementation for an open-source data processing engine, Apache Spark, and we give problems that we have solved in this setting as illustration. Our implementation consists of wrappers suitable in general for JVM environments.**

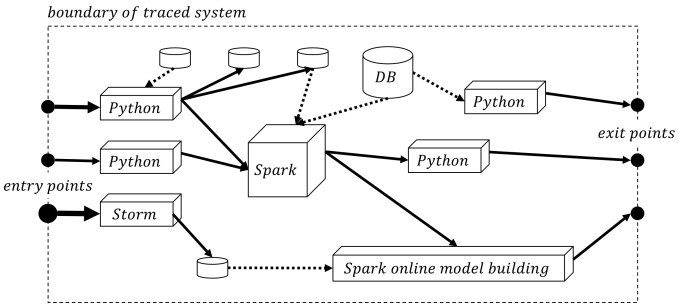


Fig. 1. A distributed compute topology that consists of different processing elements, implemented on top of different runtimes. In this setting, batch and streaming DDPS (square boxes), as well as databases (DB, round boxes) are combined to implement a production application. Solid lines represent continuous streams of data, and dotted lines denote batch access to DBs. Entry and exit points represent boundaries in which stable performance and operation must be maintained.

I. Introduction

Monitoring cloud computing platforms is a complex task of high practical relevance [1]. These platforms host distributed data processing systems (**DDPS**), implemented as standalone streaming components, usually written in different languages and maintained independently. Processing elements are combined into a distributed compute topology in order to implement a user-facing application. Due to the complexity, scale and heterogeneity of these systems, it can be extremely challenging to understand, troubleshoot operation and to detect the cause of performance degradation. Figure 1 shows an example compute topology under discussion.

Lineage, another name of data provenance [2], describes the origins and the processing history of an output record. Lineage can be reconstructed from *trace*s, per-record information collected at runtime, that capture causality relations between past, present and future records at specific points of the topology. Traces are usually collected and analyzed by an external service, that is separate from the tracing framework that records data-mutation. Collecting lineages of individual records during data processing helps to identify many application or platform-wide problems: experience has shown that identifying load imbalances, sub-optimal co-locations between services and tracking outliers are important in order to attain ideal, platform-wide operation of user-facing applications. Data processing systems are connected for a longer period of time or temporarily, on-demand. Bottlenecks propagate from one system to another, resulting in system operators and developers spending countless hours detecting and resolving issues with misconfiguration and application code. In addition to bottlenecks inherited from incorrect configuration or faulty code, complex streaming applications tend to suffer from temporal changes in data (concept drifts) and from outliers. These anomalies related to data characteristics introduce hot spots in the compute topology, causing nodes to fall behind or even to crash. Without low level data tracing in such systems, a straggler node could be misidentified as a problematic hardware in heterogeneous cluster environments.

Tracing is especially difficult for a streaming DDPS. Stream processing systems preserve no data and temporary steps during execution unlike batch computing, where stages of workloads are replayable [3]. Recording causality at the record-level might add an undesirable overhead to the compute topology. Moreover, it is important to maintain uninterrupted operation and stable tail-latency. One of our main goals is to trace individual records in both batch and streaming data processing systems.

Previous works solely focus on single batch DDPS and provide debugging capabilities offline. Although several solutions

have been proposed on top of existing frameworks, low-level metrics of User Defined Functions (**UDF**s) and low overhead can not be provided without intrusionistic modification of data processing engines [3]. Moreover, efficient design to achieve platform-wide tracing in multi-system scenarios has not been studied previously.

We consider holistic tracing of record lineages [4]. Our goal is to detect inefficiencies to increase performance of the compute topology, reduce tail-latency and better utilize the underlying platform. In this paper we focus on the tracing design and practical problems that can be solved using our framework. Due to space limitations, we chose not to discuss our external system, which provides trace analytics. We consider the following sample use cases that can be simultaneously identified by visualizing the traces of the streaming system:

- **Data skew:** most DDPS redistribute data between stages of computation using key-grouping, well-known from map-reduce. We may identify "elephant keys" or keys with increased computational cost, that overload certain nodes in the topology (for example in a skewed join). With a tracing framework, key distributions at all intermediate steps can be approximated, that allows online and balanced partitioning of data.
- **Operator fission:** in a DDPS, processing is split into stages of computation logically, then deployed as parallel operator instances on different nodes. The DDPS' own execution optimizer may pipeline many UDFs into a single operator instance, which may then overuse memory or other resources. Using distributed tracing, processing times of sample records can be provided for each operator (and for all of their pipelined UDFs), so that the logical execution plan can be optimized.
- **Records with latent properties:** input records with certain keys might not be frequent, but may grow very large by aggregations and external joins, causing problems at later stages of computation. These outliers can be traced back, filtered out or partitioned efficiently in advance.
- **Records that lead to temporal deadlocks or exceptions:** typical scenarios include high-effort parsing of certain records or inconsistent record state at later stages of the computation, which leads to exceptions in user code. Tracing can identify and filter out such records as they appear in the traced compute topology for the first time.
- **Sub-optimal co-location:** interconnected DDPS and other systems (for example distributed database systems) require their corresponding data partitions to be co-located onto the same machine in order to reduce unnecessary communication over the network. Using distributed tracing, inefficient communication patterns can be recognized across the platform, and an optimal placement can be provided.

Our contribution is the following:

1) We present a generic tracing framework design for batch and streaming DDPS on holistic compute platforms.
2) We provide a prototype implementation on Apache Spark [5], that yields low-level UDF metrics and detailed representation of causality of individual records.
3) Using the tracing framework, we show that common complex, inter- or intra-system data pipelines can be optimized by identifying issues which are hard to detect otherwise.

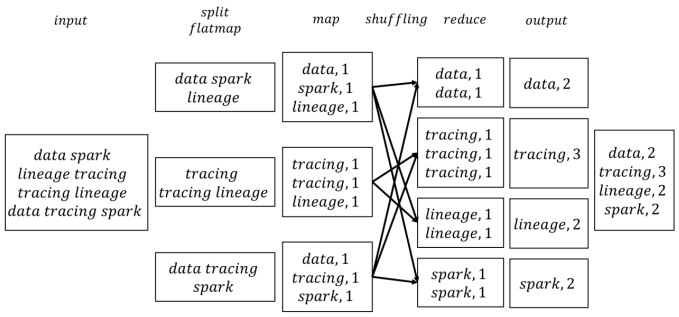


Fig. 2. Schematic illustration of the parallel word count MapReduce execution.

The rest of the paper is organized as follows. First, we outline the general DDPS architecture (Section II). Then, we describe our tracing mechanism and functional API in Section III, while Section IV contains the details of the implementation in Apache Spark. We evaluate the tracing framework in Section V and conclude the paper with an overview of the related works in Section VI.

II. DDPS ARCHITECTURE

In a DDPS, user programs are executed in numerous parallel tasks that process certain partition of the data. The user program consists of UDFs, which are first order functions plugged into second order functions such as *map* and *reduce*. As an illustration of the principle of parallel second order functions, the WordCount Spark code snippet below counts the number of times a word appears in a very large text document.

```
textFile
    .flatMap(line => line.split(" "))
    .map(word => (word, 1))
    .reduceByKey(_ + _)
```

As illustrated in Fig. 2, the text is first split and partitioned to **operator instances**. The first two parallel operations are *flatmap* and *map*, where text is **shuffled** between operator instances, split into words, and the words again shuffled. Shuffling mechanisms can, for example, be round robin or random hashing. *Map* and *flatmap* are the simplest second order functions, which execute an UDF independently on all records of a partition.

In the example of Fig. 2, the next second order parallel function is **reduce**, when the first order UDF produces a single value for each key by observing all corresponding records. Reduce uses a special case of shuffling called **key grouping** when the data is represented by key-value pairs and the records

with the same key are assigned to the same instance. Another example of key grouping includes a map side **combiner**, an optional second order function that pre-aggregates records into key-groups before they are shuffled over the network.

DDPS differ in *when* data is passed between operators. If synchronous redistribution is used, each operator instances in a stage must be completed before tasks of the next stage can be started, whereas with asynchronous redistribution, data is buffered for a short period of time (usually for milliseconds), then sent to the next operator immediately for further processing. For example, Apache Spark[1] is a synchronous and Apache Flink[2] is an asynchronous DDPS. Since in Hadoop MapReduce[3], reducers can be started after a few mappers finish, the framework is a hybrid of both models. In a streaming scenario, data is preserved at each operator instance only for a limited time.

III. The Tracing Framework

In this section, we describe our tracing framework that traces individual records in a DDPS. We build record lineage by a generic wrapper mechanism that encapsulates the record and exposes it to a functional API. We incorporate three key requirements into our design:

1) We model how bottlenecks propagate between interconnected systems, therefore, we let records be traced across several systems, probably implemented in different languages.
2) We ensure that existing user programs can run under the modified DDPS with no user code change required.
3) We trace the execution of real time data streaming frameworks, hence we allow fast and online analytics of lineage graphs to identify bottlenecks in the shortest time possible. To this end, we report and collect traces with low latency unlike in state-of-the-art solutions for batch systems.

It is already known that without certain level of invasive modification in the DDPS, we cannot provide low-level runtime information and maintain low system overhead [6]. Our goal is to minimize the level of invasiveness and in particular require absolutely no modification over the user code. Our system for Apache Spark is implemented in 1,200 lines of patch over the base system so that any Spark user code can be traced under our modified framework.

Some of the pivotal points where records are traced outside user code is shown in Figure 3. The event in which record causality is observed is called a *checkpoint*. Our framework captures record-by-record causality, measures important UDF characteristics (runtime, callsite, etc.) and collects other useful information from the underlying DDPS. In addition to UDF metrics, we collect valuable information from the internals of the DDPS' engine, for example, the time spent in intermediate buffers or the time spent on network by a record or serialization overhead.

[1]https://spark.apache.org/
[2]https://flink.apache.org/
[3]https://hadoop.apache.org/

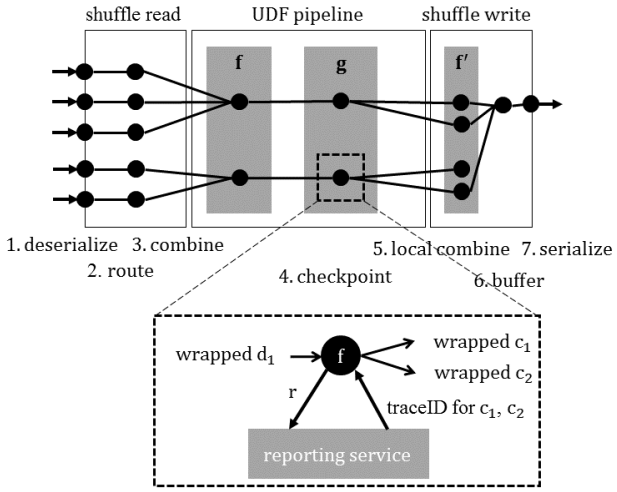


Fig. 3. Records are traced through tasks, each consisting of a shuffle read, a UDF pipeline and a shuffle write phase. Traces are recorded at the pivotal points numbered 1–7. Each dot represents a checkpoint. For one checkpoint at UDF g, we show how the reporting service is invoked over the wrapped record.

When a record d of type T passes through a DDPS with tracing capabilities, it is automatically wrapped into a lightweight **wrapper** $w(d)$ of type $W[T]$ at the entrance. To reduce unnecessary overhead caused by traces, incoming records are **sampled** randomly as candidates to build lineages.

In order to apply a UDF f on a record (the wrapper payload), it must interact with the wrapper through a functional API. This is further detailed in Subsection III-C. We enforce immutability by returning a new wrapped record $w(f(d))$ on each function apply.

In the rest of the paper, we consider and evaluate two fundamental ways of tracing, **direct reporting** and **piggybacking** that we describe in detail in the following subsections. For direct reporting, lineage information is pushed down to the reporting service in each wrapper invocation and the lineage is reconstructed by an external tool. For piggybacking, the lineage information is carried along with the record packaged into the wrapper object. Both methods share the same wrapper mechanism and the underlying, low level tracking structure.

As another, orthogonal distinction, tracing can be **forward** or **backward**. Forward tracing tracks causality by connecting a record to its descendants, thus, it observes the DDPS from the perspective of its entry points. In contrast, backward tracing links records to their ascendants, thus views the topology from the perspective of its exit points. This is the right approach if we want to reconstruct the web of records that contributed to an output entry. With the sampling turned on, only partial backward tracing can be achieved: the backward lineages of output records will be incomplete because the contribution of untracked records will be inevitably lost.

The possible combinations of the tracing approaches and their main applicability and limitation is summarized in Table I.

TABLE I
THE COMBINATION OF DIFFERENT TRACING APPROACHES.

	Direct reporting	Piggybacking
Forward	possible	not possible
Backward	possible, partial	possible, partial

A. Direct reporting

In this section, we describe our first tracing solution, which produces reports directly by continuously pushing down the lineage information of tracked records to a reporting service. The service stores traces for later use by external analytic tools. Direct reporting supports both forward and partial backward tracing.

We internally identify each record by a unique trace ID that we store in the reporting service. It generates a new trace ID whenever it encounters a new record. Our tracing framework communicates with the reporting service by sending trace reports asynchronously at the predefined pivotal points of the DDPS.

To trigger checkpointing in the tracing-enabled DDPS, a *checkpoint*() function is called from the reporting API, either without arguments or with a sequence of trace IDs and a report.

In the general case, when *checkpoint*() is called with a list of parent trace IDs and a report r, the reporting service saves r in a key-value store, where the corresponding key will be a newly generated trace ID. The framework measures different metrics of the UDF (runtime, callsite, etc.), appends it to the report, then returns the trace ID supplied by the external service.

If *checkpoint*() is called without arguments, which usually happens at the entry points of the compute topology, then the reporting service decides whether the record should be tracked at all. Records are marked for tracking by a user-specified sampling strategy. If the record is selected, then the same process happens as above, with the exception that the report corresponding to the newly generated trace ID will be empty.

The reporting library is utilized in a DDPS to track record lineages as follows. When a record enters the DDPS for the first time, it is wrapped and registered to the reporting library by calling *checkpoint*(). If a trace ID is returned, it is set in the wrapper object. Wrappers without a trace ID will be treated as "untracked" and flow silently through the compute topology, without triggering any further reporting.

Next, suppose that there is an operator in the compute topology that applies a general UDF f that takes n records as input and produces a list of m records as output. Figure 3 illustrates the reporting process with $n = 1$ and $m = 2$. When f is called on the wrappers $w(d_1), w(d_2), \ldots, w(d_n)$, then first the result $f(d_1, d_2, \ldots, d_n) = [c_1, c_2, \ldots, c_m]$ is calculated, as would happen normally in the DDPS (this step is further detailed in Subsection III-C). Then, a report r is prepared and each c_i is wrapped with the trace ID returned by a call of *checkpoint*$((q_1, q_2, \ldots, q_n), r)$, where q_i is the trace ID of the wrapper $w(d_i)$. Finally, the whole UDF call returns with $[w(c_1), w(c_2), \ldots, w(c_m)]$, and the wrapped records are forwarded to the next operators.

B. Piggybacking

Next we describe piggybacking, our second tracing approach. Compared to direct reporting, now the lineage of a tracked object is piggybacked into its wrapper for the whole course of processing, until the record is served to other systems where traces are not relevant. This approach allows optimization engines to acquire traces quickly and to pre-aggregate the record lineages under different requirements. One notable drawback of this approach is, however, that it is practically not suitable for forward tracing.

The lineage graph is an immutable graph that represents the lineage of a record and stores metrics and additional metadata on its edges and nodes. It supports two operations, *merge*() that merges two lineage graphs along the overlapping paths, and *append*() that appends a new node to the graph and adds collected metrics to the new edges. Upon a UDF call, the lineage graphs of input records are retrieved from their wrappers and merged into a single graph G. Then for each output record c_i, a new node is appended to G along with the collected metrics, and this new lineage graph is placed in the output wrapper $w(c_i)$.

C. The functional wrapper interface

Below, we outline the functional interface that is used by UDFs to interact with wrappers. Let T and U be two arbitrary data types, and suppose that a UDF $f : T \to U$ is called on a wrapper of type $W[T]$. Because this cannot be done directly, f is instead handed to the wrapper, whose interface defines several *lift*() methods to handle the most common types of UDFs as follows:

- *lift*$(f : T \to U) : W[T] \to W[U]$ makes a unary UDF applicable on a wrapped record;
- *lift*$(f : T \to U^m) : W[T] \to W[U]^m$ makes a multi-valued unary UDF applicable on a wrapped record;
- *lift*$(f : T \to \{true, false\}) : W[T] \to W[T] \cup \{\emptyset\}$ makes a filtering criterion applicable on a wrapped record; the returned object is either a wrapper or an empty collection;
- *lift*$(f : T \to \emptyset) : W[T] \to \emptyset$ makes a side-effecting function applicable on a wrapped record;
- *lift*$(f : (T, T) \to T) : (W[T], W[T]) \to W[T]$ makes a binary UDF applicable on two wrapped records of the same type;
- *lift*$(f : (T, U) \to T) : (W[T], W[U]) \to W[T]$ makes a binary UDF applicable on two wrapped records of different types;

Note that filtering must return the actual result of the operation not just a boolean, in order to keep track of records being filtered out. The two *lift*() methods for binary UDFs are added to support folding operators, e.g. in Spark (Section IV). Any mutation of the tracked records are captured by the wrapper, and if tracing mechanism is used, we call the wrappers and

the records as *Traceables*. Traceables implement and extend the default wrapper interface.

To observe the characteristics of the underlying system on which the record is passing through, and to identify system-specific bottlenecks, the wrapper can be *poke*d on which event no mutation is going to occur:

- *poke*$(w(d) : W[T],\ e : \text{Event}) : W[U]$ applies a DDPS specific transformation on the wrapped record *w(d)*.

For example, when data is streaming through map-reduce architecture, tracked records are poked immediately before being written onto the shuffle system, or read back. In this way we may measure throughput and latency on arbitrary phases of data-passing (with no UDF involved), which helps to identify problems that are hard to diagnose otherwise.

IV. Implementation for Apache Spark

We integrated the tracing framework described in the previous section into Apache Spark in order to provide a prototype implementation[4] on top of an open-source data processing engine. As we have pointed out earlier, no code change in existing Spark applications is required for our distributed tracing system to work.

We wrapped all operators in Spark's rich functional API of second order functions such as map (one-to-one), flatmap (one-to-many), reduce (many-to-one), filter, foreach (side-effecting) and also key-grouping operators (e.g. groupByKey, reduceByKey, coGroup, etc.) that group records by key and then applies the UDF to the values only.

The core of our Spark implementation consists of a thin functional layer for the wrapper API and two wrapper implementations, *Traceable* for direct reporting and *Piggyback-Traceable* for piggybacking. Here, we will only discuss the direct reporting approach. We give schematic Scala codes for the wrapped Map operation next.

```
def map(f: T => U): RDD[U] = {
  new MapPartitionsRDD[U, T](this,
   (self, iterator: Iterator[Wrapper[T]]) =>
    // wrapper.apply is called instead of f
    iter.map(wrapped => wrapped.apply(f,
     // attaching metadata to report
     new Attachment() + (callSite))))
}

override def apply(f: T => U,
  attchmnt: Attachment): Traceable[U] = {
    new Traceable[U](f(this.payload),
      // report returns a new traceID
      report(attchmnt + ("op" -> f.toString)))
}
```

UDF lifting in the wrapper API has been implemented by Scala *apply*() methods. All of these methods can receive an *attachment* in addition to the UDF. The attachment is a general purpose key-value map, which transmits additional UDF metadata to the reporting system.

The wrapper mechanism works the same as described in Section III with the addition of some trace-specific side effects. For example, Spark's aggregation based operators and their keyed counterparts work in a folding manner, processing their input in a series of intermediate steps, each merging two consecutive records. Implementing tracing for such functions was a challenging task, because we had to suspend reporting during the intermediate processing steps of such operators and report only when the final result of the aggregation is known. We solved this problem by designing a set of Spark-specific *poke* events.

Finally, the direct reporting mechanism relies on a closed source reporting library implemented in C++. Native bindings to this library have been developed for certain runtimes, for example for JVM and Python. This design helps to achieve high performance and minimum overhead.

[4]Implementation is available at https://github.com/zzvara/spark/tree/tracing

V. Experiments

A. Performance overhead

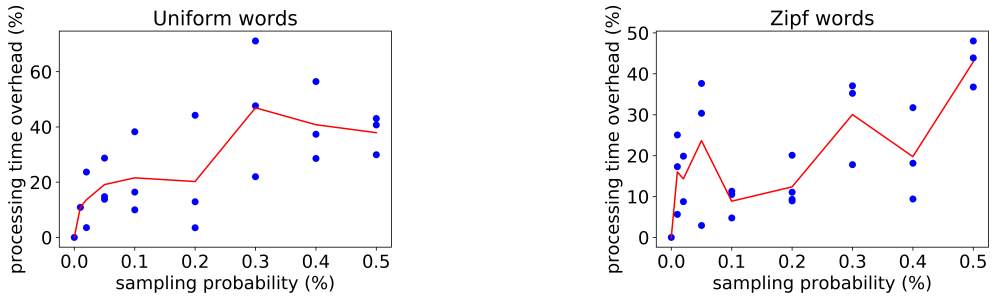
The performance overhead introduced by our tracing framework is negligible in practice, when well configured. For most use-cases we have measured an overhead of 10% or lower using direct reporting. The piggybacking approach, however, is impractical as it incurs more than 70% overhead for most use-cases. It can dramatically increase memory consumption, computation complexity (since trace graph is aggregated on-the-fly) as well as the serialization overhead for sending the trace graph to the next operator. Therefore we will focus on tracing with direct reporting in the rest of the performance evaluation.

Generally, the overhead depends on several factors. The cluster environment, the complexity of jobs and the sampling rate all affect the performance. Tracing every record might incur a 300% overhead, but lowering the sampling rate achieves better performance. In practice, tracing every 10,000th record (a sampling rate of 0.01%) could be enough for detecting bottlenecks. However, the tracing framework allows adaptive sampling rate in order to trade-off between its overhead and accuracy.

We have measured the overhead of different sampling rates using a streaming windowed WordCount job in Spark (see Fig. 4). We generated random sentences between 3 and 10 words in length, sampled from uniform and Zipf distributions (with a vocabulary of size of 8,000). Frequencies of words were computed in tumbling windows of 20,000 sentences. Experiments were conducted on a cluster of 8 machines with 4 CPU cores and 8 GB RAM each. We show the processing time of multiple jobs for the same sampling rates, and the average processing time. We can see that even a higher sampling rate of 0.5% (every 200th record) has an overhead of around 50%. Even though our topology could sustain performance under a high sampling rate such as 0.5%, it resulted in unnecessary resource consumption in our cluster.

B. Operator-fission

Data processing engines usually pipeline as many UDFs as possible into a single operator to reduce unnecessary data transfer over network. Such greedy operator compositions are



(a) (b)

Fig. 4. Overhead measurements.

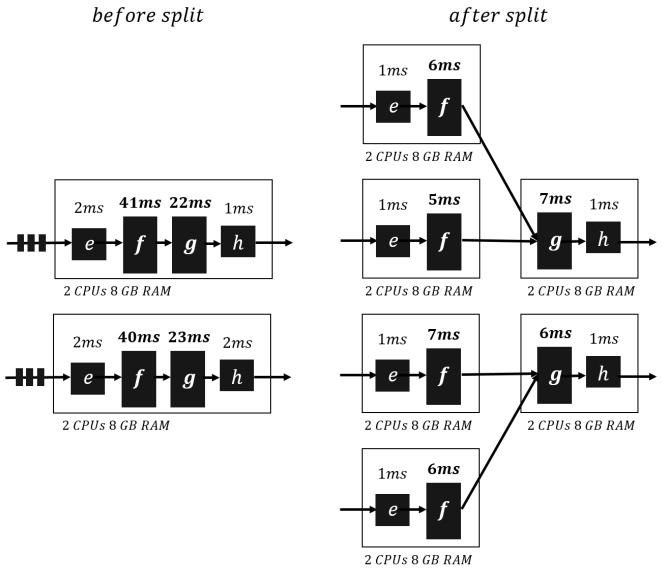


Fig. 5. On the left hand side, we detect two heavy UDFs, f and g by examining their average runtime per record. Instead of naively scaling the region horizontally, two optimizations are detected. First, split the operators by refactoring f and g into separate containers. Second, increase the parallelism of the operator that includes the heaviest UDF, f.

prone to introduce bottlenecks because of computationally heavy UDFs that are not identified in advance. Instead of eager pipelining, several UDFs should be refactored into a single parallel region (stage) in the physical execution plan. Moreover, the weight of certain UDFs can change over time, which calls for a dynamic, online fission [7]. Fig. 5 shows an example of useful operator split and fission.

Next we describe a telecommunication analytic use case, where we fetch and aggregate contextual information for user, communication tower, region, radio cell, gateway, session, device and operator, in order to compute user specific business KPIs such as network stability, download speed and latency. On Apache Spark, we split heavy flattener operators during the enrichment phase of an analytic workload. From the communication packet log stream, we compute user KPIs periodically by enriching with contextual data. In the analytic workload, a Spark parallel region with 20 partitions and 8 pipelined UDFs, tracing reported 18,000 records/second throughput, with 8% time spent on deserializing and serializing data. Each UDF called an external data store and performed a simple lookup for each record to retrieve contextual information, for example user or tower record. Retrieved data are then attached to the records. Each parallel operator ran in a container with 2 VCPUs and 8GB RAM. UDFs roughly spent the same amount of time, 0.093 milliseconds, processing each record.

Our adaptive monitoring system operates as follows. Whenever high latency is detected, it raises an alert. In this case we query the trace database for each stage of all DDPS in the compute topology. Stages with the most compute complexity are examined in a decreasing order, to identify heavy UDFs subject to horizontal splitting.

Using traces of the stage, we identified that 8 complex UDFs could be split into two operators. We utilized Spark's feature that provides a simple elastic scale-out of the system (Dynamic Allocation), which dynamically adds or removes new executors to the streaming job. After each operator instance was splitted into two containers of the same dimensions (4 UDFs in each), total serialization overhead of the workload increased by 34% on rack (due to another network hop introduced), while throughput increased to roughly 25,000 records/second on both newly created operators.

C. Outlier filtering

Incoming records with unreliable, latent properties can cause problems in later stages of the compute topology. By collecting the lineage of such records, we redirected them to another processor pipeline right at the entry point of the traced topology. As an example, when problematic input can only be identified via its lineage consider incoming TCP packet logs from a mobile network operator's monitoring system (Fig. 6). In order to compute KPIs of user communication, first, each record is enriched with session, user, cell, region and application information during several stages, by joining records with data from different external sources as in Subsection V-B. Then data is filtered, transformed and produced to other applications for further use. Complex aggregations are applied on multiple dimensions and in multiple sliding windows. The

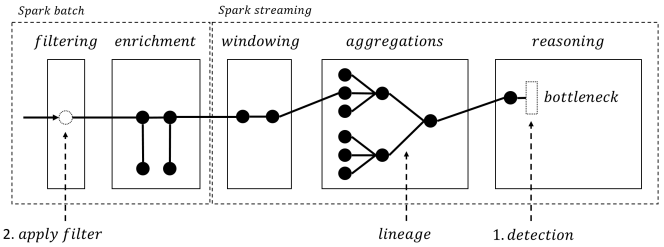


Fig. 6. A typical aggregation workload, where a pipeline could be released from a bottleneck by detecting the problematic UDF, then using the record's linage. Records that constitute bottlenecks or failures could be filtered out.

characteristics of the TCP packet data received for a certain mobile network cell can heavily influence the computational complexity of a UDF, which is hard to recognize. Due to aggregations, properties that otherwise would characterize outliers will already be eliminated by then. In such scenarios, we lose the ability to trace back problematic records, but it is achievable by exploiting a distributed tracing framework.

We identified outliers for certain workloads by analyzing the lineage graph of many traces at once. First, we identify problematic records during online trace analysis, then we extract their lineages, which are then traversed in a backward direction until topology or application entry points has been reach. Usually the algorithm which looks for entry points goes through many DDPS's. When the problematic input records have been traced back, we update the filter rule to match those records therefore discarding them from processing. In this setting we reduced tail-latency from 14 seconds to 9 seconds of the processing pipeline (Fig. 6) implemented in Apache Spark, by detecting and filtering out 0.01% of the records.

D. Handling data skew

Using distributed tracing we can identify elephant keys at any given parallel region where key-grouping is used without a map-side combine. In Spark, typical operations sensitive to data-skew include *groupBys* and *joins* as seen on Fig. 7.

We detect imbalance by examining the traces at stage boundaries. During trace analysis, a separate module inspects trace reports for every operator instance in the topology. Combiners in general perform pre-reduction, which wipe out the effects of data skew. However aggregation is not possible if every record is enriched by additional data that makes the records unique. In this case, for each record, its key is extracted from the report. As more reports arrive to the same operator instance, we build a global key-histogram periodically in a time decaying window. Using the key-histograms, we construct hybrid hash functions introduced by Gedik [8]. The new hash function is supplied for the shuffle writer module of the operator. Apache Spark is capable to change hash functions, and in a streaming scenario, due to the micro-batch nature of Spark Streaming, operators may migrate their state automatically.

For common workloads, data skew mitigation can lead to a 38 to 60 percent speedup in case of power-law distributions.

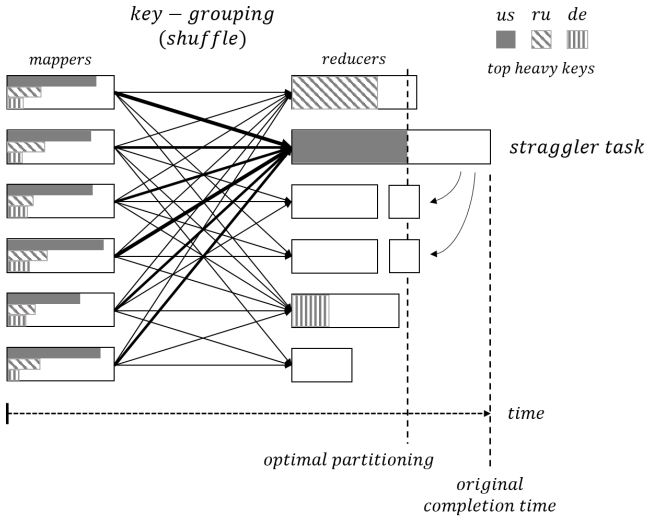


Fig. 7. Key grouping operations may produce imbalanced partition sizes on the reducer side. In this example, the heaviest key, *us*, results in a straggler task with additional random keys (represented by empty areas) also mapped there. Spark being a synchronous DDPS, a straggler task affects the runtime of the whole stage. Explicitly assigning *us* records to a partition alone and reassigning other keys reduces job completion time.

We experimented with data stream processing of 3 GB LastFM data of (user, artist, timestamp) triplets. Tracing yielded a 24% map-side overhead, and running time increased from 78 seconds to 95 seconds. After manually assigning heavy keys, reducer time was reduced from 116 to 55 seconds. In Spark, the performance of a parallel region is determined by the slowest task (partition). On 17 partitions, the size of the maximum partition has been reduced from 13.30M to 8.43M records at the reducer side. A naive idea to mitigate data skew is overpartitioning, which however increases reducer time to 240 seconds.

VI. Related work

The starting point of our work is [4], [9] where heterogeneous distributed computing workflows are monitored by attaching monitoring tags to sample records. In order to tag and monitor data, access points are necessary, which consist of connectors between different systems in [4] and low-level I/O operations and external RPC calls in [9]. In our work, we complement monitoring by enabling all Spark execution steps to serve as monitoring access points.

As another solution for Apache Spark, Titian [10] adds data provenance extension to Spark's dataset API abstraction (RDD) to ease debugging. Such a data lineage could be useful for offline reasoning, but it is unsuitable for identifying bottlenecks, sub-optimal processing pipelines in production environments. We argue that the most time is spent on optimizing and reasoning about online data processing systems. However, our solution provides the same debugging capabilities on a lower level, if required.

BigDebug [3] provides real-time debugging primitives for batch jobs with deep modifications of Apache Spark's RDD

primitive. BigDebug supports several distributed debugging features such as simulated breakpoints, fine-grained tracing and latency monitoring, and real-time quick fixes to running jobs. These features are proved to be useful for batch jobs [11], [12]. However, their techniques are not feasible for production streaming jobs, because they rely on the capability to replay stages of computation.

X-Trace [13] is a framework for inter-system tracing. It provides a holistic view on the data movement on the network between different applications. However, it cannot trace records inside the same DDPS that does not involve network transfer.

Magpie [6] provides end-to-end tracing of request-response systems. It supports only non-intrusive monitoring without modifying the monitored system. However, this non-intrusive approach does not allow low-level monitoring. Also, due to focusing on request-response systems, Magpie cannot trace more complex data transformations like joins. Pinpoint [14] takes a similar non-intrusive approach for monitoring request-response systems. Dapper [15] extends the ideas in X-Trace and Magpie with sampling and additional monitoring. However, the analysis of monitoring data has a larger (10 minutes) latency, which is impractical for streaming scenarios.

Several systems provide tracing for various batch systems that is not suitable for streaming applications. Arthur [16] selectively replays parts of the computation on map-reduce dataflow systems. While this enables debugging with minimal overhead, leaves a wide variety of bottlenecks undetected and optimizations harder to employ. RAMP [17] wraps map and reduce functions in Hadoop to achieve backward and forward tracing. Newt [18] aids batch jobs with a generic lineage instrumentation that allows the replay of captured lineage and support offline analytics on captured traces. We are not considered about replays in our streaming setting, and offline analytics are impractical. Finally, Facebook's The Mystery Machine [19] and lprof [20] target batch systems with offline analysis of monitoring data.

VII. Conclusions

Our distributed tracing framework for interconnected DDPS simplifies monitoring and aids confident reasoning on performance issues. In contrast to the state of the art, our framework design and implementation is suitable for batch and streaming workloads as well on any DDPS. In addition, compared to previous work, by capturing record lineage with low level UDF metrics across all connected systems, most bottlenecks of complex compute topologies become tractable. We also demonstrated our system design by providing an Apache Spark batch and streaming integration with real world use-cases and bottlenecks. In contrast to specialized frameworks designed to solve one bottleneck at a time, we showed that distributed and holistic tracing of records can solve many critical issues in complex user-facing applications, under one framework.

References

[1] J. S. Ward and A. Barker, "Observing the clouds: a survey and taxonomy of cloud monitoring," *Journal of Cloud Computing*, vol. 3, no. 1, p. 24, 2014.

[2] P. Buneman, S. Khanna, and T. Wang-Chiew, "Why and where: A characterization of data provenance," in *Proceedings of the 8th International Conference on Database Theory*. Springer, 2001, pp. 316–330.

[3] 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," in *Proceedings of the 38th International Conference on Software Engineering*, ser. ICSE '16. New York, NY, USA: ACM, 2016, pp. 784–795.

[4] D. Géhberger, P. Mátray, and G. Németh, "Data-driven monitoring for cloud compute systems," in *Proceedings of the 9th International Conference on Utility and Cloud Computing*, ser. UCC '16. New York, NY, USA: ACM, 2016, pp. 128–137.

[5] 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," in *Presented as part of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI 12)*. San Jose, CA: USENIX, 2012, pp. 15–28.

[6] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: Online modelling and performance-aware systems." in *HotOS*, M. B. Jones, Ed. USENIX, 2003, pp. 85–90.

[7] N. Hidalgo, D. Wladdimiro, and E. Rosas, "Self-adaptive processing graph with operator fission for elastic stream processing," *Journal of Systems and Software*, vol. 127, pp. 205–216, 2017.

[8] B. Gedik, "Partitioning functions for stateful data parallelism in stream processing," *The VLDB Journal*, vol. 23, no. 4, pp. 517–539, 2014.

[9] J. Mace, R. Roelke, and R. Fonseca, "Pivot tracing: Dynamic causal monitoring for distributed systems," in *Proceedings of the 25th Symposium on Operating Systems Principles*, ser. SOSP '15. New York, NY, USA: ACM, 2015, pp. 378–393.

[10] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie, "Titian: Data provenance support in spark," *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 216–227, Nov. 2015.

[11] C. Olston and R. Benjamin, "Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows," *Proc. VLDB Endow.*, vol. 4, no. 12, pp. 1237–1248, 2011.

[12] M. A. Gulzar, X. Han, M. Interlandi, S. Mardani, S. D. Tetali, T. D. Millstein, and M. Kim, "Interactive debugging for big data analytics." in *HotCloud*, 2016.

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica, "X-trace: a pervasive network tracing framework," in *Proceedings of the 4th USENIX conference on Networked systems design & implementation*, ser. NSDI'07. Berkeley, CA, USA: USENIX Association, 2007, pp. 20–20.

[14] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks*, ser. DSN '02. Washington, DC, USA: IEEE Computer Society, 2002, pp. 595–604.

[15] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, and C. Shanbhag, "Dapper, a large-scale distributed systems tracing infrastructure," Google, Inc., Tech. Rep., 2010.

[16] A. Dave, M. Zaharia, S. Shenker, and I. Stoica, "Arthur: Rich post-facto debugging for production analytics applications," 2013.

[17] H. Park, R. Ikeda, and J. Widom, "Ramp: A system for capturing and tracing provenance in mapreduce workflows," 2011.

[18] S. De, "Newt: an architecture for lineage-based replay and debugging in disc systems," 2012.

[19] M. Chow, D. Meisner, J. Flinn, D. Peek, and T. F. Wenisch, "The mystery machine: End-to-end performance analysis of large-scale internet services," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 217–231.

[20] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, and M. Stumm, "Lprof: A non-intrusive request flow profiler for distributed systems," in *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, ser. OSDI'14. Berkeley, CA, USA: USENIX Association, 2014, pp. 629–644.