Optimizing Distributed Data Stream Processing by Tracing☆

Zoltán Zvara[a,*], Péter G.N. Szabó[a], Gábor Hermann[a], András Benczúr[a]

[a]*Hungarian Academy of Sciences*
Institute for Computer Science and Control (MTA SZTAKI)
13-17 Kende u., H-1111 Budapest, Hungary

Abstract

In this paper, we illustrate how tracing in distributed data processing systems can be applied to improve system efficiency. By the tracing of individual input records, we may (1) identify outliers in a web crawling and document processing system and use the insights to define URL filtering rules; (2) identify heavy keys such as NULL that should be filtered before processing; (3) give hints to improve the key-based partitioning mechanisms; and (4) measure the limits of overpartitioning if heavy thread unsafe libraries are imported.

We achieve optimizations in data stream processing by our implementation of a distributed tracing engine for Apache Spark. We describe and qualitatively compare two different designs, one based on reporting to a distributed database and another based on trace piggybacking. Our prototype implementation consists of wrappers suitable in general for JVM environments, with minimal impact on the source code of the core system. Our tracing framework is the first to solve tracing in multiple systems across boundaries and to provide detailed performance measurements suitable for optimization not just debugging.

Keywords: Distributed Data Processing, Data Stream Processing, Distributed Tracing, Data Provenance, Apache Spark

1. Introduction

Monitoring cloud computing platforms is a complex task of high practical relevance [1]. In a cloud environment, one of the main challenges lies in understanding, troubleshooting distributed data processing systems (**DDPS**), and detecting causes of performance degradation.

In this paper, we propose and evaluate a tracing framework for DDPS, in which user programs are executed in numerous parallel tasks that process certain partition of the data. The user program consists of *user defined functions* (UDFs), first order functions plugged into second order functions such as *map* and *reduce*. For batch processing, data splitting is usually done in advance during storage, while for data stream processing, when partitioned data is sent to the first operators. In all cases, at the moment of the partitioning decision, the system is unaware of possible bottlenecks or skew distributions. Furthermore, the forthcoming, potentially dynamic execution graph is also unknown, which often leads to suboptimal decisions.

*Trace*s are per-record information that we collect at runtime to capture causality relations between past, present and future records at specific points of the topology. From the traces, we may reconstruct a *lineage*, which is another name of data provenance [2] to describe the origins and the processing history of an output record. After recording the traces, we build and analyze the lineages of individual records by an external service separate from the DDPS.

We describe and measure DDPS applications where we use lineage analysis to identify application or platform-wide problems such as outliers, load imbalance, or sub-optimal colocation between services. For example, we use our system to identify outliers such as very large or misformed Web documents in a streaming Web crawler, which we may then handle in additional preprocessing and filtering steps. Or we may measure the skewness in key-based partitioned processing and propose new partitioning heuristics to speed up computations. For overpartitioned load balancing solutions, we may also identify where heavy thread-unsafe libraries are included in each executor and find the best balance in the number of partitions.

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 both in runtime and in tail latency, the latter measured for a possible computational path accidentally overloaded with trace recording operations.

In order to efficiently trace individual records in both batch and streaming data processing systems, in this paper we propose and compare two alternatives. In the first solution, we directly report fine-grained events for a sample of records into an external distributed key-value store. And in the second solution, we piggyback sample records by the execution trace. In our experiments, both solutions are capable of implementing tracing in Apache Spark with low impact on the system and

☆Research was supported from the EIT Digital Activity 17186: HopsWorks and the "Momentum - Big Data" grant of the Hungarian Academy of Sciences.

*Corresponding author

Email addresses: zoltan.zvara@sztaki.mta.hu (Zoltán Zvara), peter.szabo@sztaki.mta.hu (Péter G.N. Szabó), gabor.hermann.dms@sztaki.mta.hu (Gábor Hermann), andras.benczur@sztaki.mta.hu (András Benczúr)

the application code base. In addition, direct reporting has desirable low computational overhead, while piggybacking turns out computationally expensive in practice for most use cases.

Previous works solely focus on single batch DDPS or 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. Our contribution is the following:

1. We present a generic tracing framework design for batch and streaming DDPS.
2. We provide two different prototype implementations, both built by a minimal code impact for Apache Spark [5].
3. We experiment with traced Spark appications to obtain low-level UDF metrics and detailed representation of causality of individual records.
4. We measure and compare the overhead of our tracing frameworks. We identify direct reporting as a low-impact solution for monitoring system efficiency.
5. 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.

The rest of the paper is organized as follows. In Section 2, we describe sample use cases that can be simultaneously identified by visualizing and analyzing the traces of the streaming system. Then, we outline our tracing mechanism and functional API in Section 3, while Section 4 contains the details of the implementation in Apache Spark. We measure the overhead of the tracing framework in Section 5 and demonstrate the performance gains that can be achieved by tracing, in three use cases, in Section 6. We conclude the paper with an overview of the related works in Section 7.

2. Applications

In this section, we describe several distributed data processing applications, where we may deploy tracing for performance optimization. In a DDPS, user programs compiled into a Directed Acyclic Graph (DAG) of computation, which represents a logical execution plan. The computation consists of parallel operations, where data is first split and partitioned to **operator instances**. After partitioning, the prescribed user defined function (UDF) is executed in parallel.

Typically, the first parallel operation is **flatmap** and **map**, where records shuffled, processed independently, and one output record is produced by map and multiple output records by flatmap. Shuffling mechanisms for map and flatmap can, for example, be round robin or random hashing, which usually result in even load balance.

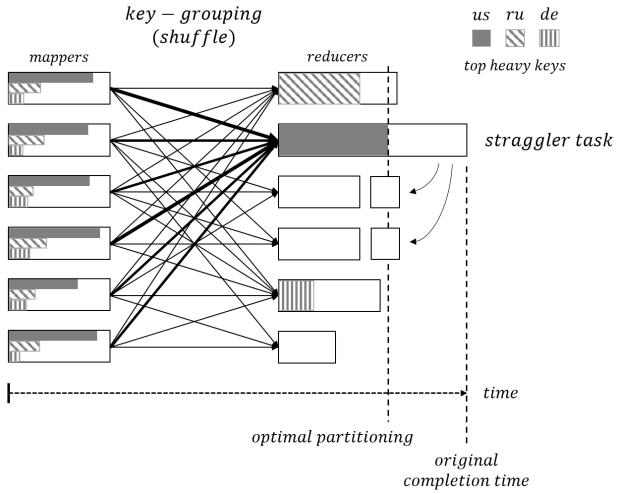


Figure 1: Key grouping operations may produce imbalanced partition sizes on the reducer side. In this example, the heaviest key, *us*, results in a straggler task with additional random keys (represented by empty areas) also mapped there. Since Spark synchronizes operations, a straggler task affects the runtime of the whole stage. Explicitly assigning *us* records to a partition alone and reassigning other keys reduces job completion time.

Other parallel functions such as **reduce**, **groupBy** or **join** typically consume the output of map or other operations by **key grouping**: data is represented by key-value pairs, and the records with the same key are assigned to the same instance. Key grouping may however be strongly affected by key imbalance and lead to straggler operator instances.

As we will illustrate in this section, the main reason for system inefficiency is suboptimal workload partitioning for operator instances. Through five different examples, we show why certain partitions may hinder execution either by containing too many records, containing certain records that require excessive computational time, overuse of resources such as exceeding memory limits, incur data errors that result in exceptions, or slow down due to suboptimal co-location with other resources such as external key-value stores. Given the motivating examples, our main goal in the rest of the paper will be to devise, illustrate and measure a tracking system that identify the cause of system inefficiency.

2.1. Data skew

Real world data usually follows power-law distribution. In a distributed processing step that groups records by keys, a key with high frequency will often end up in an overloaded partition. The task processing this partition will be identified as a straggler, and the synchronous DDPS can not progress with the next stage of computation, until all tasks finish. The problem exists in asynchronous DDPS as well. Wide array of specialized solutions (for example [6, 7]) have been developed in the past to capture key distribution during shuffle phase, which is necessary to provide uniform partitioning over all compute nodes.

Data skew may result in straggler operator instances in reduce, groupBy or join operations. The output of the preceding operation, typically map, is grouped by key. Without prior knowledge to data distribution, a hash-partitioner is used, which distributes keys uniformly among partitions and heavy keys may end up in excessively large partitions, as illustrated in Fig. 1.

The typical solution to mitigate the effect of heavy keys to include a map side **combiner**, an optional second order function that pre-aggregates records into key-groups before they are shuffled over the network. In certain cases, however, no map-side combiner can be applied. In the following code snippet builds an inverted index using `groupBy` operation, the entire sequence of word, file, count triplets have to be processed by the same operator instance to produce the search index.

```
countOfWordsPerFiles
  .groupBy { case (word, (file, count)) => word }
  .map {
    case (word, sequence) =>
      word -> sequence.map(
        case (_, (f, c)) => (f, c)
      ).mkString(‘‘, ’’)
  }
  .saveAsTextFile(...)
```

By tracing, 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, which may enable heuristics for online and balanced data partitioning.

2.2. Operator fission

When higher level programming language is used, for example SQL, the DDPS might use an SQL optimizer to further improve the logical plan. Several levels of internal optimization engines usually pipeline many UDFs into a single operator instance physically, which may then overuse memory or other resources, while not providing the necessary granularity of tasks for the cluster manager to make beneficial scheduling decisions. Instead of eager pipelining, several UDFs should be refactored into a separate 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 [8]. Fig. 2 shows an example of useful operator split and fission.

We illustrate complex execution pipelines by 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. During this phase, Apache Spark pipelined the whole operation into a single mapper, where multiple database connections and buffers

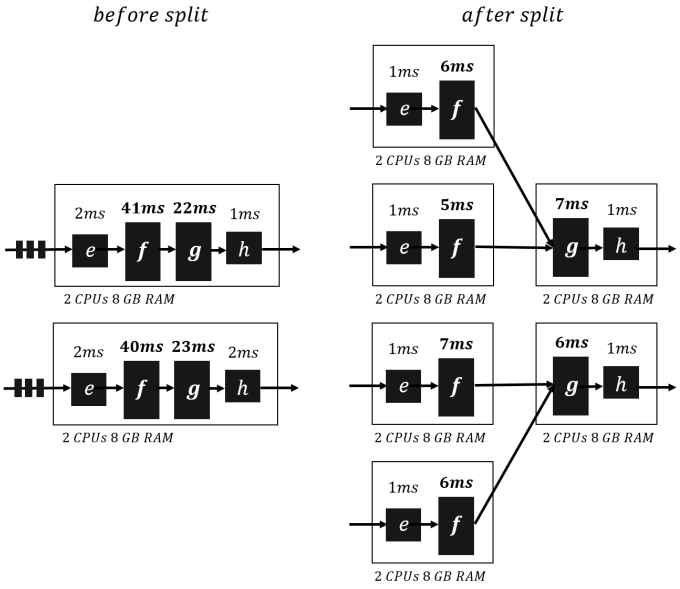


Figure 2: On the left hand side, we detect two heavy UDFs, f and g by examining their average runtime per record. Instead of naively scaling the region horizontally, two optimizations are detected. First, split the operators by refactoring f and g into separate containers. Second, increase the parallelism of the operator that includes the heaviest UDF, f.

caused frequent out-of-memory errors and resulted in inefficient resource utilization.

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, as seen on the following code snippet:

```
packetStream
  .map(toKeyValue)
  .mapPartitions(userLookupPartition) // f
  .mapPartitions(towerLookupPartition) // g
  .values
```

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.

2.3. Records with latent properties

Certain key partitions may grow exceptionally large by aggregations and external joins, causing problems at later stages of computation. Common scenarios include null items, which can only be detected by excessive manual data investigation. For example in a bank transactional analytic database, internal or so called proxy bank accounts are considered to be null items, which overload partitions and should be discarded from further computation. These outliers can be traced back, filtered out or partitioned efficiently in advance. In the code snippet below, the third line was added after, by tracing, we identified null account IDs as the reason for groupBy running out of memory regardless of the cluster configuration.

```
data = spark.read.load("transactions", format="parquet")
```

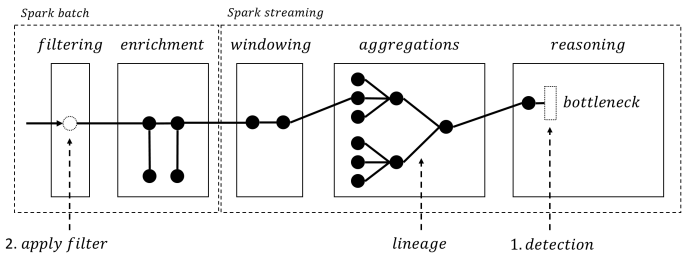


Figure 3: A typical aggregation workload, where a pipeline could be freed from a bottleneck by detecting the problematic UDF, then using the record's linage. Records that contribute to bottlenecks or failures could be filtered out.

```
data = data.join(supID_table,'acc_id',how='left_outer')
data = data.dropna(subset=['sup_id'])
data = data.groupBy(['sup_id','out_sup_id']).aggr(amount)
```

2.4. Records that lead to temporal deadlocks or exceptions

In a web crawling or unstructured text processing task, certain records may incur high-effort parsing or lead to inconsistent record state at later stages of the computation, which may lead to exceptions in user code. Programs including natural language processing are sensitive to misformed records. By collecting the lineage of such records, we can redirect 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. 3). 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 2.2. Then data is filtered, transformed and produced to other applications for further use. Afterwards, complex aggregations are applied on multiple dimensions and in multiple sliding windows. The 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. Further, problematic records are shuffled into the same partition during windowed aggregations due to key-grouping.

Because of these 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.

2.5. 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. When a processing pipeline access several external database systems at once, we may align all of the corresponding partitions onto the same machines. Using distributed tracing, inefficient communication patterns (hotspots) can be recognized across the platform, and an optimal placement can be provided.

3. 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 [9]. 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 4. 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, serialization overhead or time spent by a record in intermediate buffers or on network. Due to the flexible design of our checkpointing, detailed profiling of the DDPS internals become possible: checkpoints can be called at arbitrary points in the DDPS code path.

When a record d of type T enters a DDPS with tracing capabilities, it is automatically wrapped into a lightweight **wrapper** object $w(d)$ of type $W[T]$. To reduce tracing overhead, incoming records are **sampled** randomly as candidates to build lineages. We either sample probabilistically from incoming records, or pick one in fixed time-intervals.

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 3.3. 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. In direct reporting, lineage information is pushed down to a 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

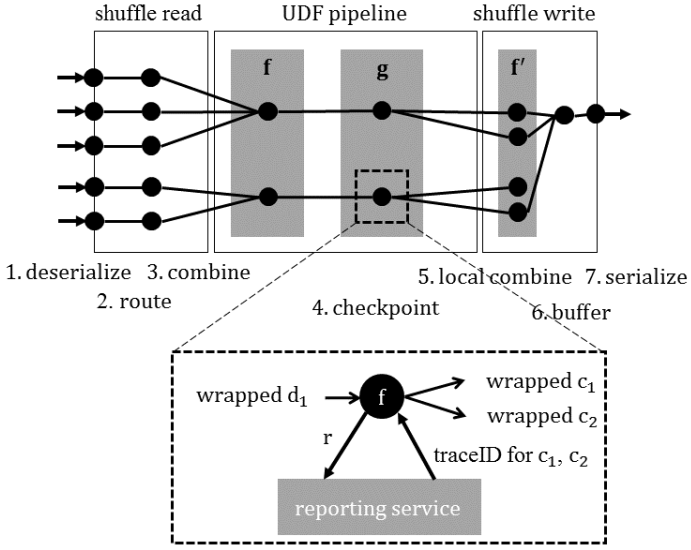


Figure 4: Records are traced through tasks, each consisting of a shuffle read, a UDF pipeline and a shuffle write phase. Traces are recorded at the pivotal points numbered 1–7. Each dot represents a checkpoint. For one checkpoint at UDF g, we show how the reporting service is invoked over the wrapped record.

Table 1: Possible combinations of different tracing approaches.

	Direct reporting	Piggybacking
Forward	yes	no
Backward	only partial	only partial

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 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 1.

3.1. 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 is stored by 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 4 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. Then, a report r is prepared and each c_i is wrapped with the trace ID returned by one 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 $[w(c_1), w(c_2), \ldots, w(c_m)]$, and the wrapped records are forwarded to the next operators.

3.2. Piggybacking

Next we describe our second tracing approach, piggybacking. As opposed to direct reporting, now the lineage of a tracked record is piggybacked into its wrapper for the whole course of processing, until the record is served to other systems where traces are not relevant. The lineage is represented in the form of a trace graph, with metrics and additional metadata stored on the nodes and edges as node and edge *weights*. These weights support pre-aggregation, which is needed when trace graphs are merged.

The trace graph API consists of two methods:

- *append(v: Vertex, vw: VertexWeight, ew: EdgeWeight): TraceGraph* creates a new trace graph by appending vertex v with vertex weight vw to the head of the trace graph. The edge weight ew will be attached to the newly created edge. Finally, the appended vertex is set to be the new head.
- *merge(other: TraceGraph): TraceGraph* merges *other* with the present trace graph. Identical nodes are merged

and weights of merged vertices and edges are added. The heads of the two trace graphs are required to be identical, so that they can be merged together to form the new head. Merge can happen, for example, when records with overlapping lineages are combined together.

Traces are not reported at checkpoints, but instead a new node with the appropriate metrics is appended to the trace graph. More precisely, when a general UDF f (as above) is called we do the following for each output record c_i. First, the trace graphs of input records are retrieved from their wrappers and a new node is appended to each of them with the collected metrics and metadata. After that, they are merged into a single trace graph G (if there are more then one of them), which is then placed in the emitted wrapper $w(c_i)$.

The advantage of the piggyback approach is that tracing metrics can be aggregated continuously, which allows embedded optimization engines to acquire record lineage information more quickly. Further, there is no need for an extra service to handle tracing reports. On the other hand, piggybacking is not suitable for forward tracing and it may add a considerable overhead in latency. Due to the fact that piggybacking mechanism is easier to adapt in existing systems, it is more suitable for offline debugging tasks rather than in a real-time feed to provide optimization heuristics.

3.3. 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 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. *reduce* in Spark (Section 4). 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 *Traceable*s. 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:

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

For example, when data is streaming through a 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.

4. Implementation for Apache Spark

We integrated the tracing framework described in the previous section into Apache Spark in order to provide a prototype implementation[1] 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 *PiggybackTraceable* for piggybacking.

4.1. Direct Reporting

First, we discuss the direct reporting approach. We give schematic Scala codes for the wrapped Map operation below.

```
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,
  attachment: Attachment): Traceable[U] = {
    new Traceable[U](f(this.payload),
      // report returns a new traceID
      report(attachment + ("operator" -> f.toString)))
}
```

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

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 3 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 reporting must be suspended during the intermediate processing steps of such operators and enabled again 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++. This design helps to achieve high performance and minimum overhead. Native bindings to this library have been developed for certain runtimes, for example for JVM and Python.

4.2. Piggyback Tracing

The piggyback tracing mechanism is integrated into Spark in a similar way as direct reporting, thus, we will only highlight the differences. In the place of Traceable, a custom wrapper implementation, PiggybackTraceable is used, which wraps the payload with a TraceGraph object, that stores metrics and additional metadata on its nodes such as creation time, cpu usage or record count, while the edge weights contain the deltas of these metrics.

The wrapping of major Spark operators and UDF lifting works as described in Section 4.1. The only major difference is that the traceable is poked *before* folding operators and not after them. Poking is also necessary before co-grouping, where data-mutation happens without a UDF call. The following code sample shows the implementation of a map UDF lifting from PiggybackTraceable.

```
override def apply(f: T => U,
  attach: Attachment): PiggybackTraceable[U] = {
    val newPayload = f(this.payload)
    val newTrace = this.trace.map( gr =>
      // appending new node to trace graph
      val vw = new TracingVertexWeight()
      gr.append(
        new TracingVertex(attach("id"), attach),
        vw, new TracingEdgeWeight(gr, vw)))
    new PiggybackTraceable[U](newPayload, newTrace)
}
```

5. Experiments I: tracing overhead

The performance overhead introduced by our tracing framework is negligible in practice, but depends among others on the cluster environment, the complexity of the jobs, as well as the sampling rate. For most use cases, we have measured an overhead of 5% or lower using direct reporting. The piggybacking approach, however, incurs in higher overhead for most use cases. Piggybacking can dramatically increase memory consumption as well as the serialization overhead for sending the trace graph to the next operator. Aggregating trace graphs also increases the computation complexity on-the-fly.

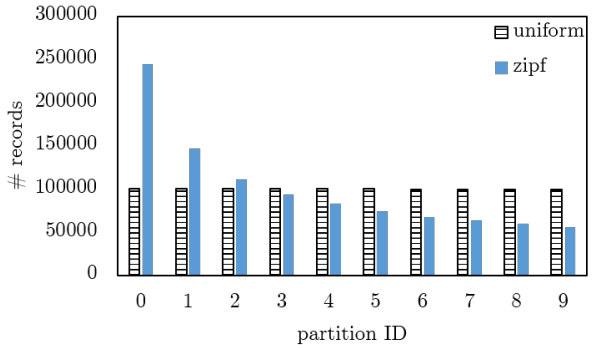
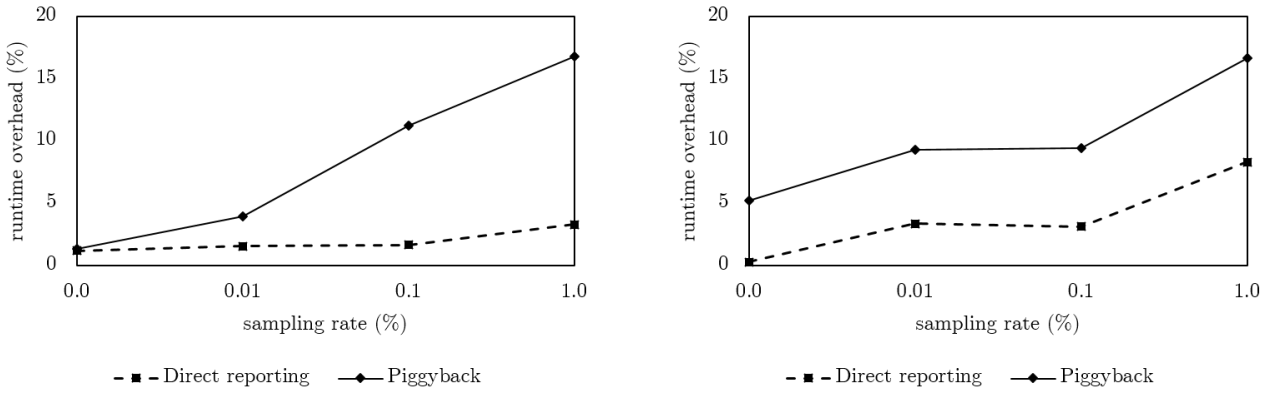


Figure 5: Number of records per partition in the RV experiment, for uniform and zipf key distributions.

We measured the runtime overhead of the direct reporting and piggyback approach under varying sampling rates. We generated a random vector (RV) dataset with distribution shown in Fig. 6. The RV dataset consists of 100,000 randomly generated double vectors of dimension 10. We ran a Spark batch job of 10 partitions on a single container with 8 GB RAM and 4 CPU cores. The job consisted of two stages, both containing heavy mapper tasks: first, we apply a geometric transformation on the vectors, then label them, group them by label as key and finally, sort each key group by vector norm. Labels were picked from a set of size 100 such that they follow a specific distribution, which was uniform distribution in the first experiment (Fig. 6a) and zipf distribution of exponent 1 in the second experiment (Fig. 6b). Figure 5 shows the number of records in each partition after the shuffle phase. Observe that there is a notable partition skew in the zipf case, where the heaviest partition is roughly 2.5 times larger than it is in the uniform case.

The overhead was measured with sampling rates $r \in \{0\%, 0.01\%, 0.1\%, 1\%\}$, which covers the corner cases as well as the most frequently recommended sampling rates from different papers. Note that 0% sampling means no tracing at all, hence we measure the overhead of the code modification with no effect. In the experiments, we averaged ten consecutive runs of the Spark job. The minimum, the maximum and the average overhead is shown in Figure 6, relative to the average runtime of the same job with tracing turned off.

It can be seen that the piggyback tracing architecture with 0% sampling rate added an 5% and 1% overhead for uniform and zipf key distribution, but it surpassed 100% in both cases if the sampling rate was close to 100%. Using direct reporting occurs in substantially less overhead for low sampling rates. Interestingly, we measured that for impractical sampling rates (for example 10% or 50%) direct reporting resulted in 3 times more overhead than piggyback tracing. Moreover, the overhead



(a) Uniform key distribution

(b) Zipf key distribution

Figure 6: Runtime overhead of direct-reporting and piggyback tracing in the RV experiment, with varying sampling rates.

was generally 6-14% larger for uniform keys than for zipf keys (depending on the sampling rate) in case of piggyback tracing. A sampling rate of 0.01% caused 9% and 4% overhead, respectively, which is affordable in many use cases, where high accuracy lineage tracking can solve serious performance problems. For example, the reference value (the average runtime with tracing switched off) was 21117 ms for uniform and 23622 ms for zipf distribution, which means that the data skew in the latter case was responsible for approximately 12% runtime overhead. Under these conditions, using our piggyback tracing framework to detect and handle data-skew adaptively would be the more efficient solution even with a sampling rate of 0.1%, where the average runtime was 23105 ms.

Figure 6 also shows that for small sampling rates ($r < 1\%$), the overhead grows in an exponential manner, while for higher sampling rates, the growth becomes linear. This is caused by the fact that the ratio of traced records can jump up significantly in a combiner operation: if even one traced record gets into a key group in the combiner phase, than the result of the operation will be traced as well. Hence, when we increase the sampling rate r starting from 0%, the overhead added by the transformations after the grouping step increases steeply. Then, when r is increased further, the key-groups gets saturated by traced records, and the rapid growth of the overhead stops, changing to a moderate linear tendency as was expected.

Another consequence of this phenomenon is the *problem of low cardinality*: the number of distinct keys and the reduction rate of any operator instance affects the tracing overhead considerably. Our solution to the problem is the following: we approximate the cardinality using traces and input metrics gathered from the DDPS. If high reduction rate and substantial overhead is detected, we reduce the sampling rate adaptively.

6. Experiments II: performance gains in use cases

In this section, we describe three use cases where we applied our tracing system to identify bottlenecks. We show optimization options and compare the performance of the unoptimized and optimized systems to show the gains that we achieved by the insights of the tracing system.

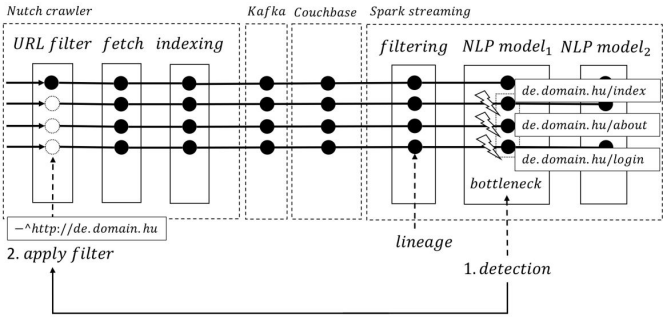


Figure 7: Web crawler and NLP pipeline.

Our experiments were conducted on four Intel HPC nodes, each with E5-2650v4 (22 thread) CPUs and 128 GB RAM on board, in a single rack.

6.1. Outlier filtering

In the first use case, we process Web content, a prominent source of unexpected, skewly distributed, and malformed input. We apply natural language processing (NLP) tools over Web crawl data. Using NLP targeted for a specific language, textual content not suitable for analysis can significantly affect the performance of the whole processing pipeline. Distributed tracing can help to debug these applications and to identify new corner-cases that lead to slowdowns or exceptions. Using web crawl, content from a site of a foreign language or content with remaining HTML tags all affect processing time considerably in NLP models (such as morphology). We used tracing to adaptively update URL filter rules by identifying the documents that slow down NLP pipelines in advanced stages of the computation.

In our experiments, we used two data sources: we process Hungarian language content from the public Twitter stream, and use the Apache Nutch Web crawler to download content from Hungarian websites into Couchbase, a NoSQL database.

We measured Apache Spark while reading over a million document updates from Couchbase nodes using cross datacenter replication protocol. The Spark pipeline then supplied these documents to several NLP models, as in Fig. 7), including language detection and morphology libraries as seen in the following code snippet.

```
couchbaseStream[Long, Document]("crawl")
  .transform { microBatch =>
    if (!microBatch.isEmpty())
      Morphology.batchMorphed {
        contentFiltered(microBatch)
      }
  }

def contentMorphed[I, D <: Unique[I]](
  mb: RDD[(I, E)]): RDD[(I, E, Option[Morph])] = {
    mb.mapPartition {
      partition =>
        val model = Morphology.borrow()
        partition.map(model.predict)
        Morphology.return(model)
    }
  }
```

The `Morphology.contentMorphed` assigns free model instances dynamically to running tasks using a pooling technique. The average time needed to predict a document in our models is 126 ms, whereas outliers can take more than 3 seconds - where a timeout is reached or exception is thrown. In our representative crawl dataset of 13.5 million records, roughly every 500th document is considered to be an outlier. Adaptively updating Nutch URL filter rules, we can increase throughput by an average of 6% in contrast to using the naive URL-filters. For example, we adaptively update filter rules for domains, where a sub-domain of foreign language version would be included for fetching: if we see that records with ancestors of *de.domain.hu* cause models to throw exceptions, we add a rule `-^http://de.domain.hu`.

6.2. *Handling data skew*

In our next use case, we applied distributed tracing to identify elephant keys at parallel regions where key-grouping is used without a map-side combine. We experimented with data stream and batch processing of 3 GB LastFM data of (user, artist, tags, timestamp) triplets. As seen in the following code snippet, the job consists of two stages: parsing on the mapper side, then grouping by tags and building an index within the reducers. In this task, map side combiners cannot reduce the data for the reduce operator instances, since every record is enriched by tags that makes the records unique.

```
driver.textFile("/timeseries.idm")
      .map(parse)
      .flatMap(r => r.tags.map(t => t -> r))
      .groupByKey()
      .mapPartition(index)
       // We allow users to use wrappers
          from application code
```

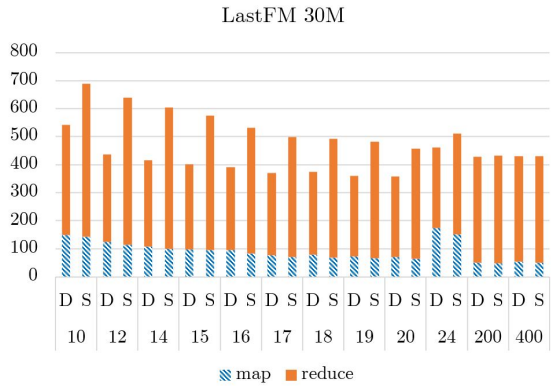


Figure 8: Runtime (in seconds) of map and reduce functions for default hash-partitioning (denoted by S) and adaptive repartitioning (denoted by D) on different number of partitions. Total number of available compute slots were set to 20.

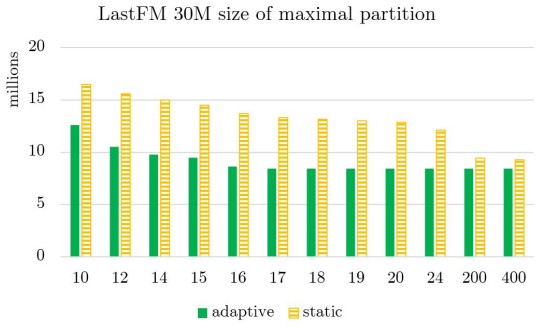


Figure 9: Maximal number of records that are shuffled into a single partition for static hash-partitioning (denoted by S) and adaptive partitioning (denoted by D).

```
.foreachPartition(ds =>
  dumpPartition(ds.collectWithWrappers))
```

Figure 8 shows runtimes for map and reduce stages for the default hash partitioning as well as for the adaptive partitioning mechanism with tracing enabled. Mapper runtimes include the overhead of our tracing framework in case of adaptive partitioning. In Spark, the performance of a parallel region is determined by the slowest task (partition). The most benefit has been observed while computing with 14 partitions, the size of the maximum partition has been reduced from 14.99M to 9.78M records at the reducer side, as shown in Figure 9. Gain slightly demolishes if number of partitions decreased: the contribution of the heaviest key is reduced in the biggest partition. For 10 partitions, the gain is 23.42%, and for 14 partitions 37.04%. A naive idea to mitigate data skew is overpartitioning, which however increases reducer time to more than 378 seconds.

We used our tracing system to detect imbalance by examining the traces at stage boundaries. Using the key-histograms, we constructed hybrid hash functions introduced by Gedik [10]. The new hash function is supplied to the shuffle writer module of the operator. Apache Spark is capable of changing hash functions, and in a streaming scenario, due to the micro-batch nature of Spark Streaming, operators may migrate their state automat-

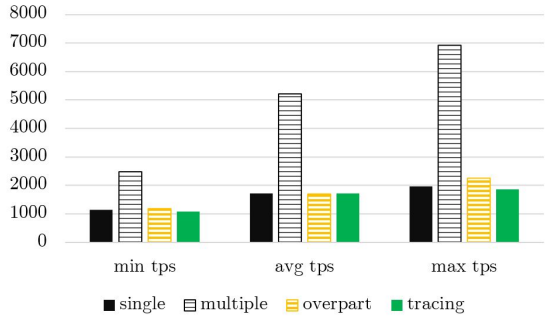


Figure 10: Throughput per micro-batch in 4 scenarios of non-thread safe NLP model parallelism.

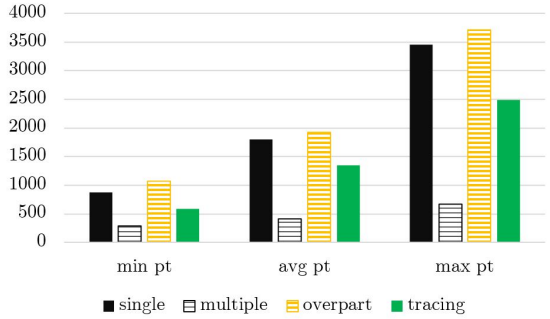


Figure 11: Processing time per micro-batch in the 4 scenarios of non-thread safe NLP model parallelism.

ically. State migration can result in unexpected downtime or increased latency and decreased throughput, therefore we use the cost model introduced by [10] in decision making.

6.3. Overpartitioning in the presence of heavy libraries

Overpartitioning is a general solution for data skew mitigation in streams where the key distribution is unknown or changes in time. However, in the natural language processing use case of Section 6.1, we ran into memory overuse when combining NLP tools with a high number of partitions. By tracing, we identified loading the NLP tools as the main reason for high memory consumption: these tools are not thread safe and hence have to upload their model data for each operator instance, even if ran in shared memory.

By tracing, we measured that each NLP processing object consumed 4.28 GB of RAM and took 5.6 seconds to load on average. These models store a state that we update for each record based on its source, i.e. the domain name in case of websites. We use the *groupBy* operator to shuffle and collect websites from the same source into a partition, then we supply the data to the model, as shown previously in Subsection 6.1. We set the batch interval for Spark Streaming to 1.5 seconds. The average processing time for each micro batch is expected to be below 1.5 seconds, so that backpressure is not required at the sources.

Figure 10 shows throughput and Figure 11 shows processing times of micro batches in multiple scenarios, when using:

- A single model in each executor, that is served to one task at a time (total memory consumption of 46 GB);
- Multiple models in each executor, so that each executor can run multiple tasks in parallel (total memory consumption of 102 GB);
- A single model, but we overpartition to mitigate data skew (total memory consumption of 47 GB).
- A single model and using tracing to capture data characteristics and construct an optimal hash function (total memory consumption of 46 GB).

Having multiple models in each executor clearly wins in terms of throughput and latency, but wastes memory, thus we do not consider it as an efficient workaround. Our starting point however, is the *single* slot and model scenario, from which increasing the number of partitions to mitigate data skew will increase processing time per micro batch: this is because tasks launched in multiple rounds introduce undesirable scheduling overhead. The scenario in which tracing is enabled (*tracing*) can reduce processing time needed per micro batch significantly, so that the 1.5 seconds SLA can be met without wasting memory in a *multiple model* scenario.

7. Related work

The starting point of our work is [11, 4] 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 [11]. In our work, we complement monitoring by enabling all Spark execution steps to serve as monitoring access points.

IDRA [12] provides adaptive breakpoints to distributed, long running and data intensive applications. In addition, with IDRA, the developer is able to change the code of the application in three phases: handling exceptions and breakpoints; reconstruct these breakpoints on remote virtual machines; fixing bugs and committing.

As another solution for Apache Spark, Titian [13, 14] 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, 15] 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 [16, 17]. However, their techniques are not feasible for production streaming jobs, because they rely on the capability to replay stages of computation.

X-Trace [18] 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 [9] 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 [19] takes a similar non-intrusive approach for monitoring request-response systems. Dapper [20] 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 [21] 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 [22] wraps map and reduce functions in Hadoop to achieve backward and forward tracing. Newt [23] aids batch jobs with a generic lineage instrumentation that allows the replay of captured lineage and support offline analytics on captured traces. We havenot considered about replays in our streaming setting, and we have found offline analytics impractical. Finally, Facebook's The Mystery Machine [24] and lprof [25] target batch systems with offline analysis of monitoring data.

8. Conclusions

Our distributed tracing framework for interconnected DDPS simplifies monitoring and aids confident reasoning on performance issues. Compared to existing tracing and debugging solutions, the key features of our framework design and implementation is that it is suitable for both batch and streaming workloads in 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 Apache Spark batch and streaming integration with real world use cases, where we showed how to identify and mitigate 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.

[1] J. S. Ward, A. Barker, Observing the clouds: a survey and taxonomy of cloud monitoring, Journal of Cloud Computing 3 (1) (2014) 24. doi:10.1186/s13677-014-0024-2.

[2] P. Buneman, S. Khanna, 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, M. Kim, Bigdebug: Debugging primitives for interactive big data processing in spark, in: Proceedings of the 38th International Conference on Software Engineering, ICSE '16, ACM, New York, NY, USA, 2016, pp. 784–795. doi:10.1145/2884781.2884813.

[4] D. Géhberger, P. Mátray, G. Németh, Data-driven monitoring for cloud compute systems, in: Proceedings of the 9th International Conference on Utility and Cloud Computing, UCC '16, ACM, New York, NY, USA, 2016, pp. 128–137. doi:10.1145/2996890.2996893.

[5] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, 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), USENIX, San Jose, CA, 2012, pp. 15–28.

[6] J. Fang, R. Zhang, T. Z. Fu, Z. Zhang, A. Zhou, J. Zhu, Parallel stream processing against workload skewness and variance, arXiv preprint arXiv:1610.05121.

[7] M. A. U. Nasir, G. D. F. Morales, N. Kourtellis, M. Serafini, When two choices are not enough: Balancing at scale in distributed stream processing, in: Data Engineering (ICDE), 2016 IEEE 32nd International Conference on, IEEE, 2016, pp. 589–600.

[8] N. Hidalgo, D. Wladdimiro, E. Rosas, Self-adaptive processing graph with operator fission for elastic stream processing, Journal of Systems and Software 127 (2017) 205–216. doi:https://doi.org/10.1016/j.jss.2016.06.010.

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

[10] B. Gedik, Partitioning functions for stateful data parallelism in stream processing, The VLDB Journal 23 (4) (2014) 517–539.

[11] J. Mace, R. Roelke, R. Fonseca, Pivot tracing: Dynamic causal monitoring for distributed systems, in: Proceedings of the 25th Symposium on Operating Systems Principles, SOSP '15, ACM, New York, NY, USA, 2015, pp. 378–393. doi:10.1145/2815400.2815415.

[12] M. Marra, Idra: an out-of-place debugger for non-stoppable applications.

[13] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, T. Condie, Titian: Data provenance support in spark, Proc. VLDB Endow. 9 (3) (2015) 216–227. doi:10.14778/2850583.2850595.

[14] M. Interlandi, A. Ekmekji, K. Shah, M. A. Gulzar, S. D. Tetali, M. Kim, T. Millstein, T. Condie, Adding data provenance support to apache spark, The VLDB Journal (2017) 1–21.

[15] M. A. Gulzar, M. Interlandi, T. Condie, M. Kim, Debugging big data analytics in spark with bigdebug, in: Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD '17, ACM, New York, NY, USA, 2017, pp. 1627–1630. doi:10.1145/3035918.3058737. URL `http://doi.acm.org/10.1145/3035918.3058737`

[16] C. Olston, R. Benjamin, Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows, Proc. VLDB Endow. 4 (12) (2011) 1237–1248.

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

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

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

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

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

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

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

[24] M. Chow, D. Meisner, J. Flinn, D. Peek, 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, OSDI'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 217–231.

[25] X. Zhao, Y. Zhang, D. Lion, M. F. Ullah, Y. Luo, D. Yuan, 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, OSDI'14, USENIX Association, Berkeley, CA, USA, 2014, pp. 629–644.