Inspector Gadget: A Framework for Custom Monitoring and Debugging of Distributed Dataflows

Christopher Olston and Benjamin Reed
Yahoo! Research

ABSTRACT

We consider how to monitor and debug query processing dataflows, in distributed environments such as *Pig/Hadoop*. Our work is motivated by a series of informal user interviews, which revealed that monitoring and debugging needs are both pressing and diverse. In response to these interviews, we created a framework for custom dataflow instrumentation, called *Inspector Gadget* (IG).

IG makes it easy to write a wide variety of monitoring and debugging behaviors, and attaches seamlessly to an existing, unmodified dataflow environment such as Pig. We have implemented a dozen user-requested tools in Inspector Gadget, each in just a few hundred lines of Java code. The performance overhead is modest in most cases.

Our Pig-based implementation of IG, called *Penny*, is slated for public release in mid-2011, in conjunction with the upcoming Apache Pig v0.9 release.

1. INTRODUCTION

Most data processing scenarios consist of data items being routed through a network of data transformation operators. Such *dataflows* are sometimes compiled from declarative query expressions (e.g. SQL), and are sometimes programmed more directly (e.g. extract-transform-load (ETL) pipelines [12], data visualization builders [17], data stream processing engines (some approaches) [1], web mashup tools [20], and dataflow frameworks for map-reduce [14]). One of the reasons to program dataflows directly is to exert more control over, and have a better understanding of, their run-time behavior, e.g. to predictably satisfy service-level agreements (SLAs), or to facilitate debugging.

Unfortunately, real-world dataflow implementations often fail to achieve run-time visibility and ease of debugging, apparently for two main reasons:

- **Difficulty in providing useful status and error messages.** Status and error reporting is essential for usability of complex systems. Yet it is very challenging to ensure that status/error messages achieve the right balance between informativeness and brevity, and are expressed at the right level of abstraction for users to comprehend. Even relatively mature systems struggle with this issue. The problem is exacerbated in multi-layer systems (e.g. workflow middleware; Oozie-Pig-Hadoop [3, 4, 14]), due to the difficulties in linking status and error messages across layers, and in translating them from lower-layer to upper-layer terminology.
- **Expense of retaining intermediate data and capturing provenance.** For efficiency reasons, many intermediate results (data flowing between pairs of operators) are not materialized, making post-hoc examination of the data processing sequence difficult. There is a great deal of published work on capturing and querying *data provenance* [9], but that only solves a subset of users' debugging needs (see Table 1), and it presents the dilemma of balancing forward processing efficiency against post-hoc debugging capability.[1]

Besides, implementors of dataflow processing engines have enough on their hands ensuring correctness, achieving good scalability and performance, and supporting new applications. "Nice-to-have" usability enhancements like informative error messages and data provenance tend (sadly) to be perpetually pushed to future release cycles.

1.1 Users' Debugging Needs

There is great demand from users for dataflow monitoring and debugging capabilities. We conducted informal interviews of ten Yahoo employees from diverse product groups that use dataflow programming. Many of them use Pig [14], but a few use other proprietary dataflow tools. In the interviews we asked what monitoring and debugging capabilities would be helpful. The responses are summarized in Table 1, which shows the distinct capabilities mentioned, ranked by the number of interviewees who mentioned each one.

Most of these capabilities are not particularly daunting to implement, but adding all of them to the core code-base of a dataflow engine such as Pig would impose a great deal of complexity. Moreover, it seems likely that yet more capabilities would be needed over time, as more users are interviewed and as new scenarios arise.

A few (4/14) of the requested capabilities could be addressed—wholly or in part—via "taint tracking" ap-

[1]Fine-grained provenance capture imposes heavy time and space overheads on normal forward processing. To support fine-grained provenance querying the data must be heavily indexed, which is especially problematic in file-based environments like map-reduce [11].

Proceedings of the VLDB Endowment, Vol. 4, No. 12

# of users	desired capability	description
7	crash culprit determination	Determine which data record and/or processing operator triggered a crash.
5	row-level integrity alerts	Throw an alert whenever a record violates a given predicate (e.g. field X not null, numerical field Y ≥ 0, date-stamp field Z $\leq$ today).
4	table-level integrity alerts	Throw an alert whenever an intermediate data set violates a given predicate (e.g. cardinality > 0).
4	data samples	Show a few samples of data on each dataflow edge, as a sanity check of the dataflow semantics and to spot fishy data (e.g. a column filled with null values).
3	data summaries	Compute a statistical summary (e.g. a histogram) of data values on a particular dataflow edge, and perhaps automatically compare against histograms from previous dataflow runs (on the same or related data) to spot sudden data distribution changes that might indicate a processing error.
3	memory use monitoring	Monitor the memory used for materializing intermediate data sets, including custom state maintained by user-defined functions.
3	backward tracing	Find the chain of input and intermediate records that led to a given output record.
2	forward tracing	Find the chain of intermediate and output records that stem from a given input record.
2	golden data/logic testing	Given a set of "golden" input/output record pairs that are known to be correct, or a function known to contain correct logic for transforming an input record into an output record, compare the dataflow input/output data against the golden pairs.
2	step-through debugging	Set breakpoints and perform step-through debugging of user-defined functions running on remote "cloud" nodes.
2	latency alerts	Throw an alert if one record takes much longer to process through a particular operator than the average record (e.g. the record contains a very large nested data set to be processed, or induces a large number of interactions with an external service).
1	latency profiling	Show the distribution of record processing latencies, perhaps in relation to record-level SLAs (e.g. certain data items related to online advertising must be processed prior to their ad campaign start date).
1	overhead profiling	Report the per-operator breakdown of total dataflow execution time.
1	trial runs	Run the dataflow on a small sample of the input data, as a quick sanity check to see whether it crashes or succeeds, and if it succeeds whether reasonable-looking output is produced.

Table 1: Monitoring and debugging capabilities requested by users.

proaches, either in the system layer (see Section 1.3) or in the query layer (e.g., add a special "taint" column, and rewrite operators to propagate this column, or use compound data values [10]). A system-layer approach would perhaps be difficult to correlate with query-layer semantics, and also would preclude deployment on third-party "cloud" systems such as Amazon's Elastic Map-Reduce (which offers Pig/Hadoop as a service). A query-layer approach would require invasive re-writing of the user's original dataflow program, thereby potentially distorting error messages. In a debugging context that would be unacceptable. Moreover, the "extra column" approach is especially problematic in loose-schema and UDF-rich environments such as Pig, because there is no simple and robust way to "add a column."[2]

1.2 Our Approach

Motivated by the above considerations, we set out to develop a framework that makes it easy to layer diverse monitoring and debugging capabilities on top of an existing dataflow engine, as unobtrusively as possible. Specifically, our goals for the framework were:

- Exploit forward processing only, and not assume post-execution availability of intermediate data sets or provenance metadata.
- Not require any modifications to the dataflow engine.
- Not tamper with data flowing through the dataflow operators, e.g. inject special "taint" columns or bits.
- Incur low overhead relative to regular processing.
- Enable a wide variety of monitoring and debugging behaviors.

The framework we created, called *Inspector Gadget*, provides abstractions for observing data passing through dataflow edges, tagging pieces of data then viewing tags at downstream observation points, and exchanging messages between pairs of observation points and with a central coordinator node. Using Inspector Gadget we have successfully implemented most of the behaviors in Table 1, each in very few lines of code (low hundreds). Our contributions lie not in the specific behaviors we implemented, but rather in the programming framework itself, and in demonstrating the ability to layer such a framework on top of an unmodified dataflow engine. Our implementation of Inspector Gadget for Pig, called *Penny*, will be packaged with the upcoming v0.9 release of Apache Pig (scheduled for mid-2011).

1.3 Related Work

Inspector Gadget focuses on a specific, but important, class of applications: distributed data processing. In that context, IG aims to enable a wide variety of behaviors (most of the ones in Table 1) with simple coding, and to avoid intrusive modifications to the underlying dataflow system and the data it manages. We believe those goals largely set it apart from other work. That said, there are several prior projects that do overlap in some ways with ours. Many of them focus on a narrower set of behaviors (e.g. just forward

[2]Prepending a column would shift column positions and cause error messages to give the wrong column number, thereby confusing users who think of their data in positional terms. Appending a column is not compatible with "jagged" rows that have different numbers of fields, and could transform "missing data" errors into "wrong type" errors or, worse, defer or even suppress errors. Another challenge would be in obtaining the correct taint-propagation semantics for user-defined functions (UDFs).

tracing, latency profiling and overhead profiling) and embed in the underlying systems at a much lower level, thereby potentially achieving better performance (for those behaviors) at the expense of more intrusiveness.

One class of mechanisms for achieving a few (4/14) of the behaviors in Table 1 is tainting with tracing [2, 7, 6, 13, 8, 18]. These approaches annotate data with special markers that enable it to be tracked as it moves through complex system(s). Perhaps the most relevant of these is X-trace [13], which can track data in and between nodes of distributed systems, including dataflow systems such as Hadoop. X-trace allows monitoring agents at data transformation points to affect the taint of generated data, to establish causal relations between the original and derived data.

Causeway [7] also uses taint markers to establish causality in distributed systems, with custom callbacks that intercept, manipulate and propagate taint among modules in an instrumented kernel. A follow-on project called Whodunit [6] added the ability to accumulate a stack-trace-like history of the procedures that have dealt with a piece of data, across multiple machines.

Aguilera et al. [2] focuses on network traces, and seeks to identify causal relationships and measure latency for chains of RPC calls. It makes a simplifying assumption that latency is mostly due to the network. Inspector Gadget is also interested in latency and causality, but because our dataflow programs can be computation and I/O intensive, we cannot rely on such an assumption. Instead, IG benefits from a different kind of simplifying assumption: the set of possible dataflow operators and control flow situations is small and known a priori.

Magpie [5] uses events generated by operating and middleware components to address some issues similar to the ones listed in Table 1. It relies on these core components to properly generate events with enough information such that events can be collected and correlated.

To instrument dataflow programs, Inspector Gadget injects code into them before they are executed by the dataflow engine. IG uses techniques similar to aspect oriented programming [16] to instrument the programs after they have been submitted but before they are executed. Other projects have used this approach for general-purpose languages, but because we are particularly interested in instrumenting dataflow programs we have a more structured and limited set of operators to deal with than general purpose languages.

1.4 Outline

The remainder of this paper is structured as follows. We present the Inspector Gadget programming model, and describe a suite of applications implemented in the IG model, in Sections 2 and 3 respectively. Then, Section 4 describes IGs semantics in the presence of distributed/parallel dataflows. We describe an implementation we have developed in the context of Pig in Section 5. Section 6 discusses the limitations of our approach. Performance experiments are reported in Section 7.

2. PROGRAMMING MODEL

This section describes our Inspector Gadget dataflow monitoring/debugging framework, from the point of view of a user of the framework (i.e. someone who wishes to create a particular monitoring or debugging application).

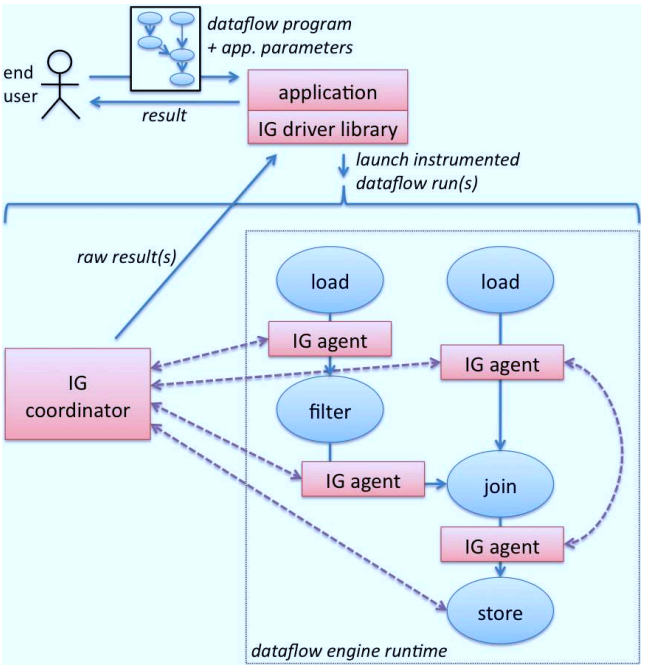

Figure 1: Instrumented dataflow.

Inspector Gadget provides abstractions for inserting *monitor agents* (agents, for short) along dataflow edges to observe data records flowing through. The agents may communicate with each other and/or with a central *coordinator*. The bottom portion of Figure 1 shows how a running dataflow (shown as interconnected ovals) is instrumented with IG monitor agents (small boxes), linked to a coordinator (large box on left). The main data processing flow (the ovals) behaves as normal—from the dataflow engine runtime's point of view the monitor agents behave as no-op functions—and a separate processing and communication plane for the IG application is layered on top.

IG applications supply code that runs inside the monitor agents and coordinator. For example, in a simple (but inefficient) implementation of the **crash culprit determination** application (Table 1), each agent sends a copy of each record it sees to the coordinator; the coordinator keeps track of the latest record sent from each agent, and if a crash occurs the last-received records are flagged as candidate culprits in triggering the crash.

Each IG application has a *driver* module, shown in the top portion of Figure 1, that receives instructions from the end user (e.g. "please provide clues about why my dataflow program crashes"), configures and launches one or more IG-instrumented dataflow runs, and composes a response to the user (e.g. "your dataflow program crashes when it tries to process records X, Y and Z").

2.1 APIs

IG application implementations consist of three pieces of code: (1) one or more monitor agent classes; (2) a coordinator class; (3) a driver class that orchestrates the overall execution. We describe the API associated with each class.

`init(args)`	Initialize the agent, given application-specific arguments.
`tags = observeRecord(record, tags)`	Observe a record on the dataflow edge being monitored, with incoming tags; assign outgoing tags or request record to be suppressed.
`receiveMessage(source, message)`	Process an incoming message from another agent (instance).
`finish()`	All records have been seen; perform any final actions.

Table 2: Monitor agent API.

`sendToCoordinator(message)`	Send a message to the coordinator.
`sendToAgent(agentId, message)`	Send a message to the agent associated with a particular dataflow edge.
`sendUpstream(message)`	Send a message to the agent located immediately upstream in the dataflow.
`sendDownstream(message)`	Send a message to the immediate downstream neighbor.

Table 3: Message sending API.

`init(args)`	Initialize the coordinator, given application-specific arguments.
`receiveMessage(source, message)`	Process an incoming message from an agent instance.
`output = finish()`	All dataflow processing has ceased; finalize processing and emit some application-specific output.

Table 4: Coordinator API.

`parsed_dataflow = parse(dataflow_spec)`	Parse a dataflow specification (e.g. Pig Latin script) into a graph representation with edge labels.
`output = launch(parsed_dataflow, agent_map, coordinator, coordinator_args)`	Instruct IG to launch a dataflow, instrumented according to `agent_map` (a set of ⟨edge id, agent class, agent args⟩ triples) and the given coordinator class and arguments. Returns the coordinator's output (if any).

Table 5: Driver API.

2.1.1 Monitor Agent

Monitor agents implement the API in Table 2. The "workhorse" method is `observeRecord()`, which is invoked each time a record passes through the dataflow edge to which the agent is attached. For example, the **row-level integrity alerts** application would use `observeRecord()` to run integrity checks on each record, and report violations to the coordinator.

Records are annotated with zero or more *tags*, which are optional record annotations used by applications that need to determine which downstream record(s) are influenced by a given upstream record, e.g. for provenance determination or to compare actual input/output pairs with "golden" pairs (detailed tagging use-case examples are given in Section 3.5). The `observeRecord()` method has access to the tags associated with the record as it enters the agent, and can select which tags are associated with the record as it exits. Although the tagging abstraction makes it appear as if we are altering the dataflow records (and hence going against one of our goals from Section 1.2), our implementation (Section 5) does not, in fact, alter the records.

`observeRecord()` has a special return option (a reserved `tags` value) that instructs the framework to suppress the record, i.e. not inject it back into the dataflow. This feature is used, e.g., in the **trial runs** application for running the dataflow on a sample of the input data, and in **overhead profiling** to isolate the cost of executing just a prefix of the dataflow. Of course, this feature does tamper with the data in the dataflow by dropping some data (and hence goes against our goals stated in Section 1.2), but only in explicitly-requested cases.

The `init()` method can be used to pass application-specific parameters to the agent, e.g. a set of data values to look for in the observed records, which initiate forward tracing of the record. `finish()` is called when it can be guaranteed that the dataflow edge will not see any further records. It can be used, for example, to signal the coordinator that this part of the dataflow has completed its processing successfully (i.e. no crash).

The `receiveMessage()` method is invoked each time a message is received from the coordinator or another agent (to be precise: agent instance, as explained in Section 4.1). For sending messages, agents have access to the methods listed in Table 3, which may be invoked from within any of the four API methods listed above, although most commonly from within `observeRecord()`. These message sending methods have very specific semantics, in terms of whether messages are delivered synchronously, asynchronously, or not at all. These semantics are described in Section 4, coupled with a requisite discussion of how dataflows execute in distributed/parallel settings.

2.1.2 Coordinator

Coordinators implement the API in Table 4. The `init()` method receives parameters from the outermost layer of the application (e.g. the golden data input/output pairs against which to compare the actual input/output pairs from the dataflow). The `finish()` method is called after all dataflow processing has completed (or crashed); it returns a result back to the outermost application layer (e.g. a "diff" of the golden data versus the actual data). The coordinator method implementations have access to the `sendToAgent()` method from Table 3.

2.1.3 Driver

Application drivers interface with IG via a simple library whose API is given in Table 5.[3] A typical driver has four steps: (1) parse the dataflow specification, (2) select edges on which to deploy monitor agents, (3) execute the monitored dataflow, (4) process the results returned by the coordinator. Some drivers execute the dataflow multiple

[3]Of course, the specific representations of `dataflow_spec`, `parsed_dataflow` and dataflow edge labels will depend on the underlying dataflow environment (e.g. Pig).

times, with different monitor agent arguments each time, e.g. our iterative **crash culprit determination** implementation described in Section 3.3.

3. EXAMPLE APPLICATIONS

We have implemented most of the user-requested behaviors in Table 1 as IG applications. The only two behaviors from Table 1 that do not fit neatly into our framework, and which we did not implement, are: (1) memory use monitoring and (2) step-wise debugging. Obtaining a detailed breakdown of memory usage would require access to the internal data structures of the dataflow system and/or user-defined functions, which is expressly outside the capabilities of our framework. Step-wise debugging can be accomplished without our framework by attaching a conventional debugger to a remote process, which poses logistical and security challenges unrelated to our framework.

Table 6 lists the applications we implemented, and gives the code sizes (all code is in Java). Our implementations of some of the applications (especially the ones marked with *) are rather basic. For example, our implementation of **row-level integrity alerts** just checks for `null` values in a user-specified field of a user-specified dataflow edge.[4] Our goal in developing these implementations was to assess the ease with which the IG-related part of the code could be written. The additional code required to expand them into more fully-fledged implementations (e.g. handling other types of integrity checks) would be orthogonal to IG. In the same vein, all of our implementations have simple command-line user interfaces, and moving to more sophisticated interfaces would of course involve additional non-IG-related code.

This section describes our implementations of the most in-demand applications: ones that were requested by two or more users interviewed (the top nine items in Table 6), grouped into five categories.

3.1 Basic Applications

Row-level integrity alerts, **data samples** and **data summaries** are all extremely simple to implement in our framework. Most of their logic resides in the monitor agent code, which occasionally transmits some data (alerts, samples, summaries) to the coordinator. In these applications the coordinator does very little, other than propagate data to the driver.

Table-level integrity alerts require the coordinator to aggregate information gathered by the agent instances, much like the reduce phase of a map-reduce job. For example, to detect when the number of records flowing on a given dataflow edge is zero, each agent maintains a counter and sends its count to the coordinator in its `finish()` method; the coordinator sums the counts across instances and then checks the constraint.

[4]The specific functionality implemented in the other cases marked with * is: **table-level integrity alerts** just checks the table cardinality on a specified dataflow edge; **data summaries** just builds a histogram of a specified field of a specified edge; **backward tracing** does not implement the weak inversion static analysis optimization to limit the scope of tagging (see Section 3.5); **golden data/logic testing** implements golden logic testing with respect to a user-supplied golden logic class (not golden data testing).

application	lines of Java code			
	driver	coord.	agents	total
crash culprit determ.	72	29	40	141
row integrity alerts*	31	23	35	89
table integrity alerts*	31	33	35	99
data samples	31	28	38	97
(with savepoints)	(35)	(119)	(118)	(272)
data summaries*	36	36	58	130
backward tracing*	45	57	135	237
forward tracing	41	28	45	114
golden logic testing*	39	69	80	200
latency alerts	33	28	107	168
latency profiling	31	52	53	136
overhead profiling	72	22	30	124
trial runs	32	22	39	93

Table 6: IG application code size.

3.2 Applications that Pause the Dataflow

As stated in Table 1, the purpose of the **data samples** application is to examine some intermediate data records as a "sanity check" for spotting any obvious problems with the data or processing. This feature is geared toward ad-hoc analytics, in which the user is deploying untested analytics code and needs to oversee the processing closely.

Sometimes the spotted problem takes the form of a faulty processing step that occurs after a sequence of expensive and/or selective steps that were performed correctly. In such cases, once the problem is noticed it is useful to save the result of the (correct) processing prefix, so the user can then repair the faulty step and resume processing from there.

To this end, we implemented an advanced version of **data samples** that includes a *savepoints* feature. It works as follows: Each time a sequence of pipelined operators is encountered (i.e. the operators in a map or reduce phase), just a few records are released into the pipeline for the purpose of displaying samples to the user; then processing pauses until the user responds (or a timeout is reached). If the user feels the samples "look okay," she signals that processing should proceed. If, on the other hand, the user spots a problem with one of the processing steps, she can request the dataflow to terminate early and produce an output corresponding to a prefix of the current pipeline (i.e., the steps before the problematic one).

Our implementation buffers a copy of each "preview" record at the start of the pipeline, and replays those records in the event that a savepoint is requested. The savepoint itself is achieved by simply opening and writing to HDFS files.

Our simple savepoints implementation enables a "try-before-you-buy" option for all non-blocking (pipelined) operators. Unfortunately this feature does not apply to blocking operators such as group-by and join, but those tend to be less problematic because they seldom involve custom user code.

As future work, we plan to explore other behaviors that pause the dataflow in order to request user input, e.g. to substitute a user-specified data value in the event of an integrity violation, or to handle unanticipated corner cases in a user-defined function.

3.3 Iterative Applications

Our implementation of **crash culprit determination** invokes `launch()` n times, each time narrowing the scope of

possible records "responsible" for the crash.[5] In each iteration the agent instances report every kth (record number, record) pair to the coordinator, starting at record number s. The coordinator keeps track of the highest record number received from each agent before the crash. In each iteration s is set to the last record number seen in the prior iteration[6], and k is progressively reduced (e.g. $k = 100, 10, 1$).

We ran our crash culprit determinator on a real Pig Latin script that caused Pig to crash with error message "ERROR 2106: Error while computing count in COUNT" followed by some detailed information that nonetheless left the exact cause of the crash a mystery (for one thing, the offending record was not printed). The script was attempting to count the number of incoming links to a particular web site, starting with a large data set of the form (url, site, inlinks, ...), by first counting the number of items in each "inlinks" field, and then grouping by site and summing the per-url inlink counts to produce per-site counts. Our crash culprit determinator produced from among millions of input records a handful of candidate crash culprits. Upon inspection one of those records turned out to contain a `null` value in its inlinks field, which turned out to be the cause of the crash. The problem was resolved by adding a filter expression to the original Pig Latin script that bypasses `null` values.

3.4 Applications that Use Inter-Agent Messaging

In our experience most applications only exchange messages between agents and the coordinator. One exception is our **latency alerts** implementation, which exchanges messages among agent instances in a peer-to-peer fashion.

Recall that the goal is to throw an alert if, on a particular point in the dataflow, a given record takes a long time to process compared to a typical record. Our implementation generates an alert if the processing time for the current record is greater than some factor F times the average processing time. The average processing time is based on measurements of records that have already been processed.

An important nuance is that while the first few records are seen, a reliable average processing time is not yet available. Let us assume that the average processing time statistic is considered reliable if it incorporates measurements from at least k records. In our implementation the monitor agents buffer the first k records, and when the $(k+1)$st record arrives the buffer is drained and any alerts are thrown retroactively. To improve the convergence on a reliable average, each agent instance also broadcasts the processing time measurements of the first k records to all peer instances, using `sendToAgent(`A`)` where A is the identifier of the edge being observed.

Although we have only described one application that explicitly leverages peer-to-peer messaging among agent instances, note that all applications that make use of the record tagging feature (Section 3.5) also use peer-to-peer messaging, albeit indirectly—our framework's implementation of tagging (Section 5.3) uses `sendDownstream()` under the hood.

3.5 Applications that Use Tagging

Recall from Section 2.1.1 that the `observeRecord()` method provides the opportunity to associate free-form *tags* with a record, which "follow" the record as it is transformed by downstream processing steps.

A simple example of an application that uses tagging is **forward tracing**, which uses two kinds of monitor agents: an *injection agent* that injects a certain tag when it observes a record of interest to trace, and a *detection agent* that notifies the coordinator whenever it observes a tagged record. The coordinator simply keeps track of received notifications and returns them to the driver at the end. The driver inserts a tag injection agent at the point in the dataflow from which tracing is to originate (typically one of the dataflow inputs), and tag detection agents at all downstream positions.

Backward tracing (determining the *provenance* of output record o) can be implemented as a sequence of two steps: (1) use a form of static dataflow analysis called *complete weak inversion* [19] to determine a superset I of input records that constitute o's provenance; (2) use Inspector Gadget with tagging to trace the path of each input record $i \in I$; the ones that "hit" o constitute o's true provenance.

Since our goal is to understand the complexity of the portion of an application that uses the IG framework, we focused on implementing the tagging step and used a trivial, conservative variant of weak inversion that places every input record in the candidate set I.

As shown in Table 6, the tagging step of backward tracing requires more code than forward tracing (described earlier). This discrepancy stems from the fact that forward tracing associates a single tag with all records being traced, whereas for backward tracing we need to assign a distinct tag to every input record in order to see which one(s) "hit" the output record of interest o.

As a final example of applications that use tagging, consider **golden data testing**. Recall that this application compares actual input/output record pairs produced by (a portion of) the dataflow, against "golden" input/output pairs supplied by the user as baselines. Tagging is used to construct the actual input/output pairs by discovering which output(s) stem from a given input.

4. PARALLELISM AND MESSAGING SEMANTICS

Inspector Gadget has somewhat nuanced message delivery semantics, developed in view of distributed/parallel dataflow execution environments such as map-reduce.

4.1 Execution Environment

The dataflow execution environment assumed by Inspector Gadget (illustrated in Figure 2) is as follows: A dataflow program is compiled into a series of *execution stages* (stages, for short), with each stage running a portion of the dataflow. In map-reduce, each map phase and reduce phase constitutes a stage. Stages are executed in serial: stage i does not begin until stage $i-1$ has completed. Execution of stage i is realized via a collection of n separate, and potentially concurrent, processes called *stage instances*. Data is partitioned among the stage instances such that each record is

[5]Our implementation assumes that the crash is being caused by a particular, problematic record. It further assumes that the order in which records are read, and the way in which they are partitioned among stage instances, are both deterministic (these are reasonable assumptions, e.g. they hold for Pig/Hadoop).

[6]There are some details in the handling of parallel agent instances that we omit for brevity.

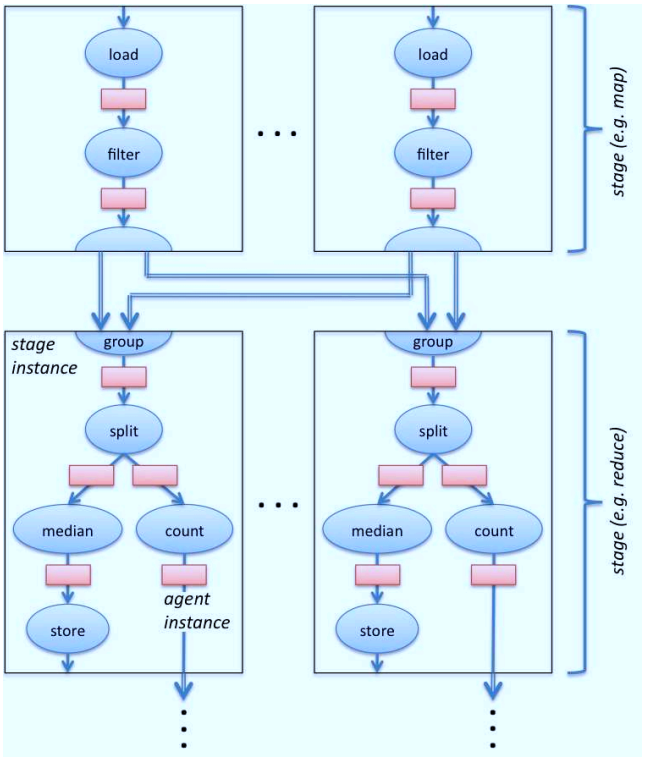

Figure 2: Distributed/parallel dataflow execution model.

handled by exactly one instance.[7] Although concurrency is permitted, there is no guarantee that all n stage instances are active at the same time—for example Hadoop [3] executes map stage instances in a series of "waves" of size k each, where oftentimes $k < n$.

Each dataflow edge occurs within a particular stage, and if the stage has n instances there will be n instances of the edge, each seeing a portion of the overall data on that edge. Correspondingly n instances of a monitor agent deployed on that edge will be run, each seeing a subset of the records on that edge. The only guarantee is that if no crash occurs, each record is seen by exactly one agent instance.

Consider a dataflow graph with edges E_1 and E_2 such that E_2 lies immediately downstream from E_1 (i.e. there is one vertex V that is the target of E_1 and the source of E_2). Suppose that monitor agents are deployed on both E_1 and E_2, called A_1 and A_2 respectively. There are two cases: either (1) A_1 and A_2 execute as part of the same stage, or (2) A_2 is part of the stage immediately following A_1's stage. In Case 2 there is a simple temporal relationship between the execution of A_1 instances and A_2 instances: no instance of A_2 can begin until all instances of A_1 have completed. In Case 1, for a given instance of A_1 there must be a corresponding instance of A_2 executing at the same time inside the same process; but other instances of A_2, as well as peer instances of A_1, are part of other processes that may have already completed, may still be active, or may not yet have started. This distinction has implications for inter-agent messaging, as we discuss next.

[7]If a stage instance crashes and gets re-started, already-handled records may be handled again in the re-started instance. We discuss the implications of this situation to applications briefly in Section 6.

4.2 Messaging Semantics

Suppose a message is aimed at a particular monitor agent instance I. There are four possible scenarios:

1. I has already run to completion—it is no longer running.
2. I is currently running, and is executing in the same process as the sender (i.e. both sender and recipient are monitor agent instances running in the same stage instance).
3. I is currently running, in a different process (possibly on another machine).
4. I has not yet started.

In Scenario 1 the message of course cannot be delivered. In Scenario 2 the message is delivered to the recipient (via I.`receiveMessage()`; see Table 2) prior to the next invocation of I.`observeRecord()`. In Scenario 3 an attempt is made to deliver the message to the recipient in a timely manner, but with no guarantees about the interleaving with I.`observeRecord()` or that it is delivered at all (i.e. the recipient might terminate before the message arrives). In Scenario 4 the message is delivered via I.`receiveMessage()` prior to *any* invocations of I.`observeRecord()`.

Turning to the message sending API available to monitor agents (Table 3), the semantics of the four methods are:

- `sendToCoordinator()` transmits a message to the coordinator node asynchronously (i.e. the method invocation may return before the message is delivered).
- `sendToAgent()` attempts to transmit the message to all instances of a given agent, with the delivery timing and success dictated by the four scenarios described above.
- `sendUpstream()` just transmits to any same-stage-instance (i.e. same process) agent instances deployed on the immediate upstream edge, according to Scenario 2 above.
- `sendDownstream()` behaves like `sendUpstream()` (except of course targeted at downstream neighbors) for neighbors that are part of the same stage. For cross-stage neighbors the message is delivered to every instance of the downstream agent according to Scenario 4.

5. IMPLEMENTATION FOR PIG: PENNY

We describe our implementation of the IG abstraction for Pig [14], an open-source dataflow engine originally developed at Yahoo. Our implementation is written in Java (to match Pig and its underlying processing platform Hadoop [3]) and is called *Penny*. While parts of our implementation are, by necessity, Pig-specific (e.g. use of wrapper UDFs to embed monitor agents in a Pig dataflow), much of it is entirely separate from the details of Pig itself and likely reusable in other dataflow settings (e.g. our messaging implementation). Our tagging implementation falls in-between these two extremes:

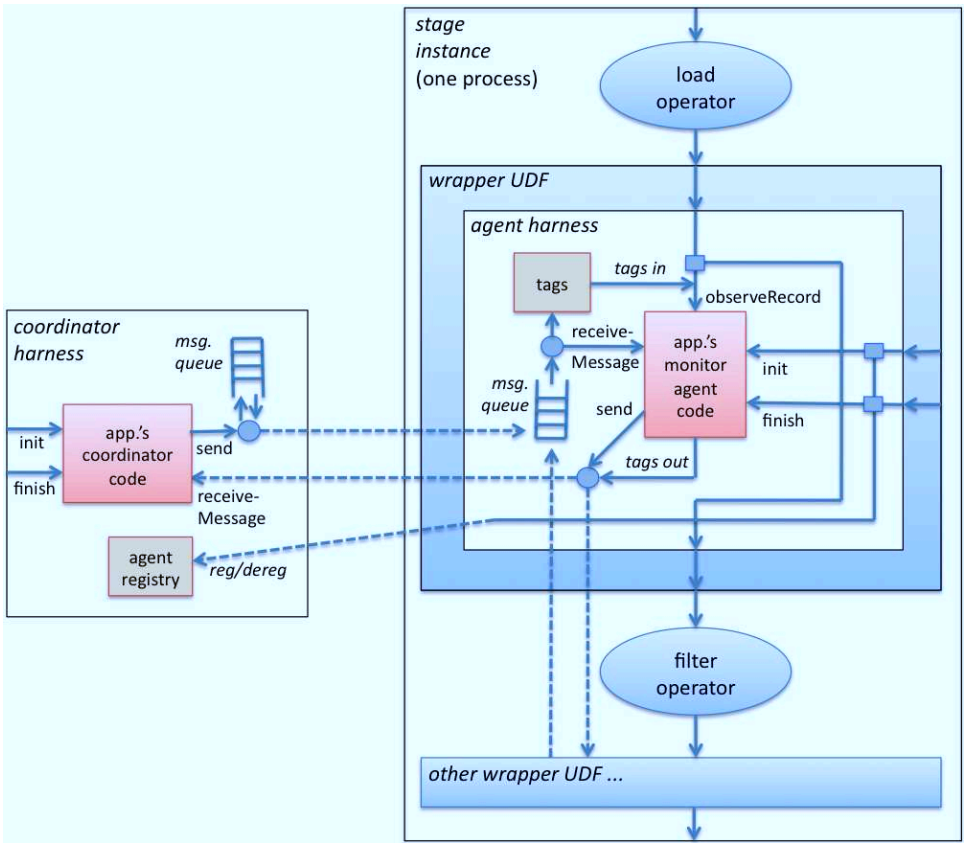

Figure 3: Penny implementation details.

it exploits knowledge of Pig compilation and execution behavior, but the strategies it employs can likely be adapted to other systems.

Figure 3 shows the implementation structure and communication pattern during dataflow execution. The right-hand side of the figure shows a single stage instance (one process running one or more dataflow operators and monitor agents). The left-hand side shows the coordinator. The application's coordinator code and monitor agent code are both wrapped with *harnesses*, which is where most of Penny's implementation of the Inspector Gadget semantics takes place; the harnesses and communication patterns are described in detail below. Before we proceed, note that the monitor agent harness is enclosed in a *wrapper UDF*, which is a Pig Latin user-defined function (UDF) that interfaces between the harness and Pig's UDF API for processing records. The wrapper UDFs are inserted into the dataflow via simple Pig Latin script manipulation prior to execution (in the driver `launch()` method).

5.1 Basic Processing

The harnesses take care of initializing (with arguments) the application-provided coordinator or monitor agent class running inside them, and exchanging messages with other harnesses (including ones in the same process and ones on different processes/machines; see Section 5.2). The coordinator harness maintains a registry of currently running agent instances. Monitor agent harnesses register and deregister themselves by sending messages to the coordinator and waiting for acknowledgments.

When a wrapper UDF is handed an input record from the Pig runtime, the wrapper in turn hands the record to the harness, which then hands it to the agent code (by invoking its `observeRecord()` method), possibly after associating one or more tags (Section 5.3 describes how tags are determined). After `observeRecord()` returns (its return value specifies new tags, whose treatment is described in Section 5.3), the harness hands the original input record back to the wrapper UDF, which then gets handed back to the Pig runtime as the UDF output (unless `observeRecord()`'s return value requested that the record be suppressed). Barring suppression, from the Pig runtime's point of view the wrapper UDF is a no-op.

5.2 Messaging

The messaging abstractions between pairs of agent instances and between agent instances and the coordinator is implemented via a combination of mechanisms, mainly for efficiency purposes.

`sendToCoordinator()`: The simplest case is when an agent instance sends a message to the coordinator (the `sendToCoordinator()` method in Table 3), which is handled via a straightforward network message to the coordinator. For all network messaging, Penny harnesses use thread pools on both the sending side and the receiving side to allow concurrency with other processing and messaging events. (When a harness is ready to shut down it first waits for all pending messages to be delivered.)

`sendToAgent()`: When the coordinator sends a message to an agent via `sendToAgent()`, a copy is immediately transmitted to all currently registered instances of the agent. Additionally, the message is placed in the coordinator's message queue, awaiting any instances that register in the future. If an agent instance invokes `sendToAgent()` the message is first relayed to the coordinator, which in turn handles it in the same way as messages originating from the coordinator, as just described.

`sendUpstream()`: As specified in Section 4.2, this method only applies to transmitting a message to an upstream agent instance running in the same stage instance, and hence the same process. Our implementation leverages this fact by simply inserting the message into the recipient agent instance's local queue (this data structure permits insertion by the harnesses of other within-process agent instances, for this purpose).

`sendDownstream()`: Recall from Section 4.2 that a given invocation of `sendDownstream()` falls into one of two cases: (1) if the immediate downstream neighbor agent is part of the same stage, then the implementation matches that of `sendUpstream()` (direct insertion into the recipient's message queue); (2) if the immediate downstream neighbor agent is part of the subsequent stage, then the message is relayed via the coordinator, where it will wait in the coordinator's message queue until downstream agent instances come online and register.

When an agent instance registers with the coordinator, any enqueued messages destined for that agent are copied from the coordinator's message queue to the agent instance's message queue, after which registration is considered completed. Each agent instance guarantees to process any locally enqueued messages (via `receiveMessage()`) before processing an incoming record (`observeRecord()`). Hence, messages sent from an upstream agent instance to a downstream instance across a stage boundary are guaranteed to be handled by the downstream agent instance before it handles its first dataflow record. This guarantee is important for correct semantics of applications that cascade initialization state down the dataflow, and is also relied upon by our tagging implementation, described next.

5.3 Tagging

Tagging is implemented on top of our messaging abstraction (tagging-related messages are kept separate from application messages using metadata in the message headers). Consider a dataflow sequence $A_1 \rightarrow O \rightarrow A_2$ which has a first monitor agent A_1 followed by a Pig operator O followed by a second monitor agent A_2. Suppose A_1 emits record r_i with associated tags τ, and suppose that when r_i passes through O it contributes to a (possibly empty) set of output records R_o. The tagging implementation must ensure that whenever a record $r_o \in R_o$ is passed to A_2 the tags τ are passed along with it.

Our implementation strategy relies on knowledge of the rules for compiling a dataflow script into a sequence of stages (map and reduce stages, in the case of Pig), which in Pig are simple, deterministic, and have remained the same for several years. In particular, our implementation handles the following two cases differently:

1. O is a non-blocking operation (e.g. filter, project, or user-defined functions (UDFs)[8]) and is executed as part of a single stage. In this case, the Pig compiler rules guarantee that A_1 and A_2 execute in the same stage with O.

2. O is a blocking operation that spans a stage boundary (for Pig, one of: group-by, co-group, join or sort, all of which exploit the shuffle step that occurs between a map phase and reduce phase). In this case, the Pig compiler rules guarantee that A_2 executes in a later stage than A_1.

Our implementation strategy for Case 1 exploits the fact that Pig, like many dataflow systems, uses the *iterator model* [15] for pulling data through operators within a given stage instance. For our purposes the important aspect of the iterator model is that there is no queueing of records along dataflow edges. Our implementation works as follows: Before A_1's harness releases record r_i to downstream processing, it signals to A_2 (using `sendDownstream()`) that any subsequent records arriving at A_2 should be tagged with τ. When A_1 receives its next input record (or finishes, if there are no more input records), it signals to A_2 to stop using τ to tag arriving records. The set of records received by A_2 in between the two signals from A_1 are exactly R_o.

Our strategy for Case 2 exploits the semantics of specific cross-stage Pig operators (there are four: group-by, cogroup, join and sort). In this scenario A_1 is in the stage prior to A_2, and `sendDownstream()` invocations from A_1 broadcast messages to all instances of A_2, which they receive at registration time prior to seeing any records (see Section 4.2). If the operator between A_1 and A_2 is group-by on field f, then A_1 simply notifies all A_2 instances to associate tags τ with the grouped record with group key $r_i.f$. For example, if grouping web crawl records by f = web site, if r_i's web site is `amazon.com` then the group formed by O with group key `amazon.com` will be tagged with τ. Of course, in many-to-one operations like group-by a single output record may collect a large number of tags if many input records are tagged.

Co-group, join and sort are handled similarly and we omit the details. A caveat is that operations such as join and sort do not produce a field that acts as a unique key for tag propagation (unlike group-by and co-group, which do produce unique group keys). Instead, for those operations one has to either: (1) leverage a field that is known, through separate means e.g. a system catalog or an assertion from a user, to constitute a unique key (e.g. URLs or SSNs), or (2) use all available record fields in combination as record identifiers, and accept tag cross-overs among identical records (e.g. if the input stream contains two identical records ⟨`Joe Smith`, `Los Angeles`⟩ and we wish to tag only one of them and trace it through the dataflow, we may not be able to do so).

6. LIMITATIONS

Inspector Gadget is a simple and powerful way to add monitoring and debugging capabilities to an existing dataflow system such as Pig. However, our strategy of not

[8]Our implementation only handles UDFs that are stateless, i.e. the UDF does not buffer any data, and consequently the result it produces upon seeing input record r_i is a function of r_i alone.

modifying the underlying dataflow system or tampering with its data induces some limitations:

- IG assumes the translation from the user's original dataflow script to the dataflow execution graph is direct. Aggressive query optimization, e.g. reordering operators, can get in the way of IG, and vice-versa (IG's monitor agent UDFs can interfere with operator commutativity). Pig's current query optimizer is rather limited, but future releases or other dataflow systems may be problematic. One possible workaround is for IG to instrument the post-optimized dataflow graph, which requires a way to view and modify the post-optimized graph and has implications for the user interface.
- Since IG rests at a high level of abstraction, it can be difficult to correlate observations made in the IG layer with lower-layer observations. For example, it is difficult to match a given IG agent instance with a particular underlying Hadoop map or reduce task that failed.
- Stage instance re-execution (due to a crash) may result in the same record being processed multiple times by a monitor agent. Our framework does not provide special support for this situation, and leaves it to be handled by the application. In all of the applications we have implemented (Table 6), re-starts do not cause any serious ill effects: Applications that transmit summaries to the coordinator upon agent completion (e.g. table-level integrity alerts) are unaffected, and ones that transmit messages on the fly experience non-harmful redundancies (e.g. the same row-level integrity alert is thrown multiple times).
- Our tagging implementation (Section 5.3) relies on messaging and is designed for applications that trace a small number of records. Tracing a large number of records with this mechanism incurs excessive overhead.
- The correctness of our tagging implementation relies on several assumptions about the dataflow engine, operators and data (e.g. no buffering/queueing inside or between operators; availability of unique keys—see Section 5.3), which hold in many contexts but are not universally valid. Bear in mind that the majority of our application scenarios (8/12 rows in Table 6) do not use tagging.

7. EXPERIMENTS

The key evaluation metrics for Inspector Gadget are: (1) applications enabled and their code size (reported earlier in Table 6), and (2) performance overhead (this section). In a debugging context, users are generally more interested in the added functionality offered by the debugging tools than their performance impact—up to a point. Since IG is a very general framework, it is possible for applications to use its APIs in ways that generate very high performance overheads. In this section we show that the overheads incurred by actual debugging applications requested by users are acceptable: In some cases (e.g. integrity alerts), the overhead is so small that it could be used to monitor a production deployment. For others, the overhead is not negligible, but nonetheless small enough for debugging purposes. Even in a few "bad" cases, the execution time remains within a factor of two, which is probably acceptable for debugging.

Except where noted, our performance experiments use Hadoop 0.20 with Pig 0.7. We used a cluster of 15 machines connected to a common switch with 1G network links. Each machine has two 7200 RPM SATA drives with dual core 2.13 GHz Xeon processors and 4G of memory. We dedicated one machine to running the Hadoop JobTracker and NameNode.

Our experiments use four Pig Latin scripts selected for their different optimization and compilation properties. Each script runs over a small 10GB, 10 million record, sample of web crawl data, in which each record represents a web page and contains, among other fields, the URL, site, language, spam score, inlinks, outlinks, and anchor text. The scripts are described in Table 7, which indicates whether each script can benefit from certain optimizations that Pig performs (projecting unused columns early; using the Hadoop combiner for early partial aggregation) and how many map-reduce jobs the script gets compiled into.

Our goal is to evaluate IG's overhead for the applications listed in Table 6. We consider two baselines: (1) regular Pig execution (without IG); (2) execution with a no-op IG application, which deploys a no-op agent at every dataflow edge. For comparison against those baselines, we take applications from Table 6 that perform a single pass and do not filter any data.[9] (Our iterative crash culprit determination application is studied separately in Section 7.1.) For applications that monitor a particular dataflow edge (e.g. integrity alerts), we monitor the first edge (i.e. right after the loading step).

Figure 4 shows the running time of each baseline and application, averaged over ten runs (the standard deviations are shown as error bars). The application abbreviations are as follows:

- **RI:** row-level integrity alerts, which checks for a null value in one of the fields on one dataflow edge.
- **TI:** table-level integrity alerts, which checks for an intermediate table (the set of records passing along an edge) that is smaller than expected.
- **DS:** data samples, with five samples requested from each instance of each edge.
- **DH:** data summaries, in the form of a histogram of data values in one particular field on one edge.
- **FT:** forward tracing of one input record as it passes through the rest of the dataflow.
- **LA:** latency alerts on all dataflow edges.
- **LP:** latency profiling, which tracks the latencies of five input records per load instance, as they pass through the rest of the dataflow.

Result analysis.

The no-op IG application baseline performs somewhat worse than the regular Pig baseline in three of the four scripts, reflecting the fact that adding monitoring agents on all dataflow edges impedes the early projection and/or combiner optimizations. The **DS**, **FT**, **LA** and **LP** applications place monitoring agents on all dataflow edges, and hence their performance is bounded by that of the no-op application on all scripts. The remaining applications (**DH**, **RI** and **TI**) only monitor the first edge (the one immediately following the initial loading of the data) and hence do not

[9]We did not include backward tracing in our experiments, because its performance will depend almost entirely on the effectiveness of the weak inversion analysis phase that we did not implement.

script name	description	applicable optimizations: early projection	applicable optimizations: combiner	# of map-reduce jobs
Distinct Inlinks	Projects the data to just site and inlinks, groups by site, and for each site finds the distinct inlinks and counts them.	NO	NO	1
Frequent Anchor Text	Groups the data by site, and for each site uses a non-combinable UDF to extract the frequent terms from the anchor text of all its pages.	YES	NO	1
Big Site Count	Filters out records not belonging to one particularly large web site, counts the inlinks of each page of that site, groups the counts together, and adds them up.	YES	YES	1
Linked by Large	Finds the number of distinct inlinks each site has that originate from a large site. It projects to just url, site and outlinks, then groups by site, filters out sites with fewer than k pages, projects and flattens the resulting records to site and outlinks, groups by outlink, and for each outlink gets the count of the distinct set of sites with that outlink.	NO	YES	2

Table 7: Pig Latin scripts used in experiments.

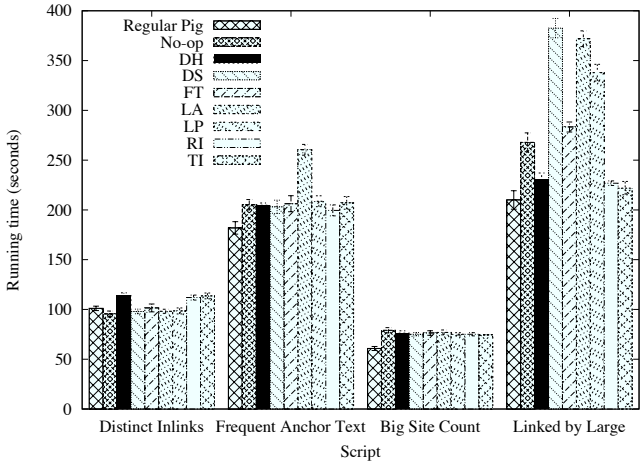

Figure 4: Running times of two baselines and seven IG applications, for each of the four scripts.

interfere with the combiner optimization, and only interfere slightly with the early projection optimization.

Although **DH**, **RI** and **TI** do not suffer from optimization-related slowdowns, they do exhibit other slowdowns not related to optimization, as we can see clearly with the Distinct Inlinks script (to which neither optimization applies). **DH** and **TI** require their agent instances to send a message to the coordinator in their `finish()` method (**DH**'s histograms, and **TI**'s record counts), which delays shutting down the process (shutdown waits for all enqueued messages to be sent). **RI** checks whether the `inlinks` field contains a null value, and our naive implementation does this by first deserializing the data in the field and then comparing it with null—since some records have very large inlink sets the deserialization process incurs a measurable overhead.

Most of the performance discrepancies are explained by the aforementioned factors. The remaining cases, each of which exhibits a fairly large performance degradation, are:

- The Frequent Anchor Text and Linked by Large scripts exhibit significant variability in per-record latency, which causes **LA** to generate a large number of alerts; in our implementation each alert transmits the entire content of the offending record, which amounts to quite a bit of data because many of the records contain large nested inlink sets.
- The poor performance of **DS** on the Linked by Large script is also due to transmitting a large amount of data to the coordinator: since that script deals only with records having large inlink bags, every sampled record (except those on the edge prior to the initial filter step) is large.
- **LP** performs poorly on the Linked by Large script, because it tags many records (whereas **FT** only tags one) and the script's group-and-flatten sequence causes the tags to spread to a large number of downstream records. One could presumably improve the **LP** implementation to repeatedly trim the number of tagged records.

7.1 Crash Culprit Determination

We also measure the performance of our iterative **crash culprit determination** application, using the scenario mentioned at the end of Section 3.3 in which the Big Site Count script failed because of bad data. For this test we revert to an earlier version of Pig (version 0.6), because version 0.7 automatically converts null values into empty sets for the purpose of counting. We also configured Hadoop to not retry failed tasks, which is how a crash culprit determination application would configure Hadoop in practice—however note that our application is able to handle retries, and the relative performance difference between our application and regular Pig is not affected much by this configuration change.

The ten-run average of the running time of the plain Pig script (i.e. time until the crash) is 24.5 seconds. Our **crash culprit determination** application configured to make three passes ($k = 100, 10, 1$) takes, on average over ten runs, 81.4 seconds to find candidate culprit records, which is not much more than three times the plain Pig running time.

8. SUMMARY

This paper presented *Inspector Gadget*, a framework that layers highly customizable monitoring and debugging capability on an existing (distributed) dataflow engine such as Pig. Inspector Gadget enabled us to implement 12 of the 14 monitoring/debugging capabilities requested by users that

we interviewed, each in just a few hundred lines of code. Experiments showed that our IG implementation, called *Penny*, incurs only modest overhead for most real-world use-cases. Penny is scheduled for public release as part of the v0.9 release of Apache Pig, in mid-2011.

9. REFERENCES

[1] D. J. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 12(2):120–139, 2003.

[2] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proc. SOSP*, 2003.

[3] Apache. Hadoop: Open-source implementation of MapReduce. `http://hadoop.apache.org`.

[4] Apache. Oozie: Hadoop workflow system. `http://issues.apache.org/jira/browse/HADOOP-5303`.

[5] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier. Using magpie for request extraction and workload modelling. In *Proc. OSDI*, 2004.

[6] A. Chanda, A. L. Cox, and W. Zwaenepoel. Whodunit: Transactional profiling for multi-tier applications. In *Proc. Eurosys*, 2007.

[7] A. Chanda, K. Elmeleegy, A. L. Cox, and W. Zwaenepoel. Causeway: Support for controlling and analyzing the execution of web-accessible applications. In *Proc. Middleware*, 2005.

[8] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet services. *Proc. DSN*, 2002.

[9] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1(4):379–474, 2009.

[10] B. Davis and H. Chen. DBTaint: Cross-application information flow tracking via databases. In *Proc. USENIX Conference on Web Application Development*, 2010.

[11] J. Dean and S. Ghemawat. MapReduce: Simplified data processing on large clusters. In *Proc. OSDI*, 2004.

[12] Extract, transform, load. `http://en.wikipedia.org/wiki/Extract,_transform,_load`.

[13] R. Fonseca, G. Porter, R. H. Katz, S. Shenker, and I. Stoica. X-trace: A pervasive network tracing framework. In *Proc. NSDI*, 2007.

[14] A. F. Gates, O. Natkovich, S. Chopra, P. Kamath, S. M. Narayanamurthy, C. Olston, B. Reed, S. Srinivasan, and U. Srivastava. Building a high-level dataflow system on top of map-reduce: The Pig experience. In *Proc. VLDB*, 2009.

[15] G. Graefe. Encapsulation of parallelism in the volcano query processing system. In *Proc. SIGMOD*, 1990.

[16] M. Haupt and H. Schippers. A machine model for aspect-oriented programming. In *Proc. ECOOP*, 2007.

[17] M. Stonebraker, J. Chen, N. Nathan, C. Paxson, and J. Wu. Tioga: Providing data management support for scientific visualization applications. In *Proc. VLDB*, 1993.

[18] E. Thereska, B. Salmon, J. Strunk, M. Wachs, M. Abd-El-Malek, J. Lopez, and G. R. Ganger. Stardust: tracking activity in a distributed storage system. *SIGMETRICS Perform. Eval. Rev.*, 34(1):3–14, 2006.

[19] A. Woodruff and M. Stonebraker. Supporting fine-grained data lineage in a database visualization environment. In *Proc. ICDE*, 1997.

[20] Yahoo!, Inc. Pipes: Rewire the web. `http://pipes.yahoo.com`.