UC San Diego
Technical Reports

Title

Scalable Lineage Capture for Debugging DISC Analytics

Permalink
https://escholarship.org/uc/item/2hm9z9d2

Authors
Logothetis, Dionysios
De, Soumyarupa
Yocum, Kenneth

Publication Date
2012-02-01

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

Scalable Lineage Capture for Debugging DISC Analytics

UCSD Computer Science and Engineering Tech Report CSE2012-0990

Dionysios Logothetis
Telefonica Research

Soumyarupa De
Microsoft, Inc.

Kenneth Yocum
Computer Science, UC San Diego

Abstract

A fundamental challenge for big-data analytics is how to efficiently tune and debug multi-step dataflows. This paper presents Newt, a scalable architecture for capturing and using record-level data lineage to discover and resolve errors in analytics. Newt's flexible instrumentation allows system developers to collect this fine-grain lineage from a range of data intensive scalable computing (DISC) architectures, actively recording the flow of data through multi-step, user-defined transformations. Newt pairs this API with a scale-out, fault-tolerant lineage store and query engine.

We find that while active collection can be expensive, real-world analytics often incur modest runtime overheads (<36%) and it enables novel lineage-based debugging techniques. For instance, Newt can efficiently recreate errors (crashes or bad outputs) or remove input data from the dataflow to enable data cleaning strategies. Additionally, Newt's active lineage collection allows retrospective analyses of a dataflow's behavior, such as identifying anomalous stages. As case studies, we instrument two DISC systems, Hadoop and Hyracks, with less than 105 lines of additional code for each. Finally, we use Newt to systematically clean input data to a Hadoop-based de novo genome assembler, significantly improving the quality of the output assembly.

1 Introduction

Modern data-intensive scalable computing (DISC) systems support analytics that use many non-relational transformations. We believe the next bottleneck to large-scale data analysis will be debugging and tuning these analytics. Errors arise at many levels—from semantic bugs, bad data, or hardware errors. With large input data sets, traditional debugging techniques are at best cumbersome, if not intractable. For example, MapReduce users run individual map or reduce tasks in isolation on small inputs to debug their code. However many pathological cases arise when running at scale on large input data sets. While it may be possible to manually prune the inputs to reproduce the bug, considerable programmer effort and additional machine time is required.

This paper explores how to capture and use fine-grain lineage, a record of how input records affect outputs, during the tuning and debugging of big-data analytics.[1] Though recent projects explore lineage in this very context, they focus on drill-down queries (tracing backward from outputs) [2, 19, 10], limit capture to one system (e.g., Pig/Hadoop) [10, 19, 2], or support coarse-grain or low-data rate collection [13, 19]. On the other hand, novel lineage models [15] and use cases, such as output refresh [11], offer sound theoretical underpinnings but lack scalable realizations.

This work presents Newt, an instrumentation-based approach for capturing lineage from DISC systems at runtime. A key contribution of this work is the introduction of novel debugging methodologies. For instance, Newt enables lineage data mining strategies, such as identifying changes in lineage to pinpoint suspect transformations, e.g., those that suddenly produce much more or less data. We also use Newt's ability to *remove* data from a pipeline to develop a general methodology for cleaning analytic pipelines of bad data. We employ this technique on non-trivial analytics, in particular to improve de novo genetic assembly when input data sets are polluted with DNA from multiple organisms [8, 18].

On the face of it, active capture of fine-grain dataflow may appear expensive and impractical, even for use during debugging and tuning. Such objections have driven the development of lazy[7] or forensic [19] capture systems that must re-execute analytics (perhaps multiple times) to produce lineage. This work questions that premise by studying the real-world costs of such an approach and weighing those penalties against the benefits of having the entire record-by-record dataflow immediately available to the user or data scientist. We believe that doing so provides both new abilities (mentioned above) and fast, arbitrary tracing queries and debugging replays.

In addition to those listed above, this work makes the following additional contributions:

- **Generic lineage instrumentation:** The Newt capture API supports the groupwise processing constructs (e.g,. MapReduce's reduce) that underly many DISC systems. It does so by offering *paired* capture (Section 3.2), which independently records the arrival of inputs and departure of outputs from the operator. Post capture, Newt

[1] Newt collects lineage in the spirit of Cui et al. [7], a specific form of why provenance [6].

reconstructs the lineage using the *timing* of data arrivals and departures. This approach simplifies instrumentation, reduces capture overheads, and improves lineage accuracy (Section 7.1). As case studies, we instrument both Hadoop and Hyracks [5] DISC frameworks.

- **Scalable capture and replay architecture:** Newt employs a scale-out, fault-tolerant architecture to store, query, and replay captured lineage. Newt can actively track fine-grain, record-level provenance with a modest run time penalty of 12-49% and a 34-69% output storage penalty for the PigMix benchmark (a set of Pig scripts that compile to Hadoop jobs). For more complex, non-relational dataflows, such as Contrail [17], a Hadoop-based de novo genome assembler, and Mahout [1], Hadoop-based machine learning, we observed run time slowdowns of 20-36%. Finally, Newt's efficient replay recreates individual outputs with a subset of input data, enabling step-wise debugging at a fraction of the cost of the original execution (Section 7.2).

2 Overview

This section introduces a simple scenario to illustrate how lineage helps tune and debug DISC analytics. Our goal is to enable system developers[2] to instrument the underlying DISC system once, and then extract lineage from any user-defined analytics that run on top.

Note that in some use cases, Newt's contribution is not the lineage facility itself, such as tracing or replay, but a general and efficient realization of it. For instance, while Ikeda et al. introduced a theoretical basis for correct replay [10], Newt develops the mechanisms for finding feasible and efficient replays (Section 3.4). The following dataflow scenario is inspired by actual analytics at a large mobile telecom.

2.1 The case for lineage

The telecom data scientist writes a Pig script [14] to produce aggregate statistics from daily mobile network traffic logs with tuples containing a uid, agent, and bytes transferred. The scientist wishes to consider only cellular network users (no tethering), so they remove log entries that contain agents suspected of tethering using a filter containing a regular expression. Downstream analytics join this intermediate output of (uid, agent) pairs with the original input to produce aggregate statistics for non-tethering users. Figure 1 shows the resulting sequence of MapReduce jobs.

[2]We distinguish system developers (or simply "developers"), those that create DISC architectures, from the data scientists or users that write programs, scripts, or dataflows that run on those DISC systems.

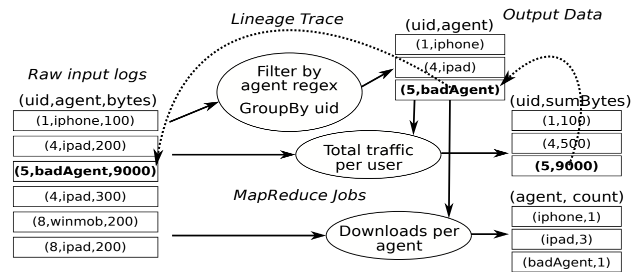


Figure 1: This example dataflow first filters network traffic logs looking for non-tethering user agents. The intermediate output is joined repeatedly with the original input to produce aggregate statistics, e.g., total traffic and downloads per user agent. Dotted arrows indicate the lineage of suspect user with a tethering agent.

Unfortunately, our scientist often encounters errors when new agents or agent spellings pass the initial filter. In Figure 1 "badAgent" ultimately pollutes the intermediate and final aggregate results. Without lineage the scientist must debug the analytic by writing additional Pig scripts. The first finds uid's with abnormally high download volumes, indicating tethering, and the second script scans the input records for those uid's, recording their agent fields. The scientist then updates the regular expression and re-runs the entire job. While time-consuming and fragile, this general process is common to many analytics. As we explain later, genome researchers attempt the same process to clean their input data sets.

Existing lineage-based facilities Fine-grain lineage simplifies this process. First, backward tracing, often called drill-down, allows a user to work backward from outputs to the inputs (and preceding transforms). A simple backward trace from the (**5, 9000**) tuple gives the scientist all bad inputs (and the offending agent) without running the second Pig script. In the opposite direction, forward tracing determines the outputs derived from the bad input records, alerting the scientist to any intermediate results that may be invalid.

A more challenging scenario occurs when the original Pig script silently fails due to malformed inputs. Now the data scientist must modify the MapReduce code to catch the offending record or manually scan for erroneous input records. Instead, with lineage-based crash-culprit determination [13] the scientist could record the offending inputs and issue a backwards trace to find the input records that caused the crash. Finally replaying [10] that backwards trace would re-run the dataflow only on the affected subset of input data, reducing debugging time and effort. Newt provides a practical implementation of these existing techniques.

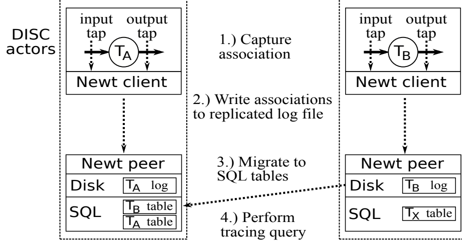


Figure 2: A Newt client sends associations to a peer node who writes them to local disk and then builds indexed SQL tables. The peer backs up associations by lazily recording them on another peer's disk. Newt peers and clients may also be co-located.

Facilities unique to Newt This work explores two new facilities that address other potential problems for our data scientist. First, after identifying the "badAgent" records with a trace, typically the scientist wishes to rerun the script with clean inputs. To do so, Newt provides *exclusive* replay (ex-replay). While ordinary replay runs the analytic on the subset of inputs indicated by a tracing query, ex-replay *excludes* those inputs. In particular, ex-replay can exclude or squash data from any stage in a dataflow, not just the beginning.

Our example assumed the scientist could find bad outputs or experienced a crash. This, though, is not always the case. Without known bad outputs, users need to analyze the dataflow for suspicious behavior in general. To that end, Newt provides a set of debugging interfaces (Section 4.1) that assist a user in inspecting lineage for dataflow anomalies, as well as identifying faulty actors given a bad output.

2.2 Design

Newt employs a scale-out architecture of Newt peers colocated with each node in the DISC cluster (Figure 2). System developers instrument each logical *actor* in the DISC environment to capture lineage *associations*, the relationship between data inputs and outputs. An actor is an entity that transforms data; it may be a Dryad vertex, a MapReduce job, individual map and reduce operators, or an entire dataflow.

The lifecycle of a single instrumented dataflow consists of three primary phases: capture, tracing, and replay. In the capture phase, Newt acts as a scalable, parallel sink for lineage associations, delivering them directly to local disk (possibly replicating the log file on a different peer). Once the capture phase has ended and the analytic is finished, the system imports each actor's association log into an indexed SQL table, called an *association* table. At this point Newt can perform tracing queries by issuing a sequence of joins across the association tables, matching the outputs of one actor with the inputs of another (Section 3.3).

A logically centralized Newt controller (not shown) distributes the load of ingesting associations, maintains a directory of association tables stored in the cluster, and manages failures by rebuilding tables as needed. For instance, the controller decides the peer that will store the SQL association table (chosen to optimize tracing queries). The controller also performs the tracing queries and orchestrates replay by restarting the necessary actors.

Beyond scalability, Newt must address two other key challenges. First, a key feature of many DISC systems is the ability for users to incorporate non-relational or user-defined functions (UDFs). This makes tracking fine-grain lineage in a transparent fashion difficult. Second, the definition of lineage is a set of inputs sufficient to recreate an output. However, lineage is not guaranteed to be minimal; there may be unnecessary input items associated with the output. Improving lineage accuracy can minimize storage overheads, narrow the scope of input data considered when debugging, and minimize dataflow replay processing.

We recognize that recording record-level lineage is space intensive. However, storage is relatively inexpensive versus compute, and we leave minimizing storage overheads as future work. In addition, while Newt can capture lineage from a non-deterministic operator, it can only accurately replay deterministic transforms.

3 Lineage capture, tracing, and replay

This section discusses how Newt captures, queries, and replays lineage. We first describe Newt's notion and representation of lineage. Next, we present the capture API and how it accommodates common operators found in DISC environments. The remainder of this section describes Newt's Tracing function, as well as the semantics, operation, and limitations of dataflow replay and ex-replay.

3.1 Data lineage in Newt

Each actor in the DISC environment reads data items from an input I and creates a set of output items O. Given $O = P(I)$, where P is an actor transform, provenance associates input records $i \in I$ with output records $o \in O$. Newt allows the programmer to associate a subset of input records $I' \subseteq I$ with each o. An association takes the form of (I', P, o). To Newt, these associations represent data lineage, a specific form of provenance developed to perform tracing queries for arbitrary data warehousing transforms [7].

To enable replay, Newt requires the instrumented actor to report *complete* data lineage. In this case, I' con-

Function	Description
Standard Capture API	
capture(id,H_{in},H_{out}) → filter	Create provenance association (H_{in}, p_{id}, H_{out}). If supporting replay, if `filter` is true, drop output.
Paired Capture API	
addInput(id,H_{in},T) → filter	Add H_{in} to current association set D_T, where T is an optional tag. If supporting replay, if `filter` is true, drop input.
addOutput(id,H_{out},T)	Add provenance association (D_T, p_{id}, H_{out}).
reset(T)	Reset association set D_T to $\emptyset$.
Management	
register(name,g,α) → id	Register actor of type g with Newt before processing. Returns unique identifier, id.
commit(id)	Inform Newt that this actor has completed processing.
flow_link(id$_{src}$,id$_{dst}$)	Inform Newt that actor id$_{dst}$ receives data from actor id$_{src}$.
Debugging Interface	
trace(r[],dir,p_{root}) → F	Trace elements in r[] produced by actor p_{root} in dir direction. Returns a set of tracing dataflows, F.
mine(p_{root},o_{bad},O_{good}[]) → R	Mine the dataflow contained by actor p_{root} to identify anomalies. See Section 4.1
fail(id,H_{in},H_{out})	Records culprit input when actor encounters exception—H_{out} is optional.

Table 1: Newt API. Note that if tags (T) are not supplied in the paired API, sys will use a default tag $*$.

tains all inputs considered by actor P when it created o during P's execution, and this ensures $o \in O'$ when $O' = P(I')$ [7]. Note that this I' may not be unique; there may be many other possible witness sets from which o can be generated using P. While prior work tried to discover all such witness sets [16], here we are interested in the one that occurred during a dataflow's execution.

To collect data lineage, Newt instrumentation captures individual association pairs, (d^{in},d^{out}), that relate input data d^{in} to output data d^{out} for an actor instance p, where each p is given a unique instance identifier. In practice d^{in} and d^{out} are typically cryptographic hashes on byte ranges or short, specific data fields that the system developer knows the system will preserve (e.g., grouping keys or file-offset-length triples). When the dataflow completes, Newt creates an association table $\mathtt{A}_p$ for each actor instance p with a row per association and columns for inputs and outputs.

Newt depends on the system developer to indicate a connected set of actors to correctly perform tracing and replay. Figure 3 illustrates a single Hadoop job that invokes instrumentation for record readers, record writers, and map and reduce tasks. System developers use the flow_link API call (Table 1) to tell Newt the *capture dataflow*, a DAG of actor instances. Recording the dataflow is often straightforward: DISC job controllers know the explicit dataflow and actor identifiers.

Figure 3 shows the job and task-level captured dataflows. However, to answer queries that span levels, such as "which input records did a MapReduce job read?", Newt captures *containment* relationships. To do so, an instrumented DISC system declares a set of actor and data types and arranges each into a logical *containment hierarchy*. For example, MapReduce jobs contain maps and reducers, and files contain records. In addition, Newt records *instance containment* relationships (e.g, record y is contained in file x) at run time.

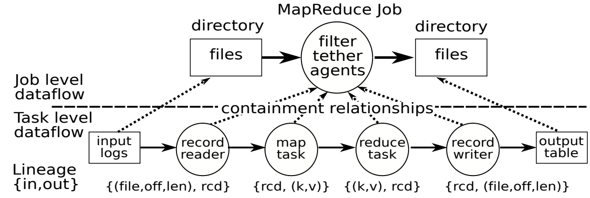


Figure 3: Newt records a MapReduce job and task level dataflow, containment relationships that unify them, and the flow of data between actors.

Here Newt extends the concept of containment, introduced by the Ibis multi-granularity provenance model [15], to support replay. In this case one actor type contains another if it initiates that actor (directly or transitively) and cannot complete until it completes. For instance, each MR Job spawns record readers, maps, reduces, and record writers that must finish for the job to complete. Newt depends on this semantic to garbage collect lineage from speculative tasks (Section 5.3).

3.2 Capture API

Developers use the capture APIs in Table 1 to create lineage associations. We found one-to-one or one-to-many operators, such as the map operator, relatively easy to instrument with a single call to capture, which associates an input element hash H_{in} with an output element hash H_{out}.[3] In these cases, all the outputs of such an operation can be associated with the same input.

[3]Developers define a set of data types (Section 3.1) for input and output elements and these types determine how to compute H, the hash of the data element.

However, groupwise or many-to-many operators are more challenging because they inherently buffer inputs before emitting results. These transforms typically call a user-defined function with an array of input data elements. For example, Hadoop's reduce task calls the user's reduce function with an iterator containing all values for a reduce key k. The downside is that such iterator-based functions force us to associate *all* input values for a given reduce key k with the reduce output $k', v'[]$. The left side of Figure 4 shows the $(N * M)$ associations (arrows) that the standard capture API produces—each output associates with each input value.

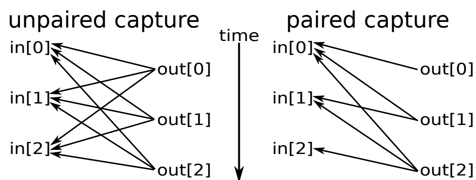


Figure 4: Paired capture creates fewer associations (edges) when used within an iterator-based function such as reduce. An output only associates with inputs that occur earlier in time (e.g, happened-before).

However virtually all physical DISC operators execute on a single machine and many are pipelined [15]. Thus, the temporal order of outputs relative to inputs gives an approximation of lineage—an output can only appear if its inputs have been read. These observations lead to a second API we call *paired* capture (Table 1). Here a developer can independently instrument the input and output sides of an actor using addInput and addOutput respectively. During capture Newt internally timestamps these observations, and issues them to the lineage storage peer without further processing. Thus an input is only sent once to Newt, not once for each associated output as in capture. This approach improves accuracy for pipelined transforms, and reduces network traffic and storage overheads.

Only after capture does Newt reconstruct the associations. For each actor in the trace, Newt simply associates an output item with any input with a lesser timestamp (i.e., it happened-before). Intuitively, each call to addInput adds an input element to an *association set* D_*. When the programmer invokes addOutput, Newt associates all elements in D_* with the output. However actors should flush D_* (by calling reset) when the next output does not depend on any prior inputs in D_*. For instance, our reduce task instrumentation calls reset when processing the next key k. The right side of Figure 4 shows how this paired API reduces the number of associations.

However, paired capture remains problematic for operators that buffer all inputs before releasing outputs. For instance, the Hyracks HashGroup operator groups all input into a hash table before processing the found keys. To assist in these cases, Newt provides *tags*. Developers add a tag T to input items in addInput, and Newt associates an output item with T with all inputs in D_T.[4] Note that Newt does not propagate T; it must either be an operator artifact or communicated by other means (e.g., Inspector Gadget agents [13]). We use tags in our Hyracks instrumentation (Section 6.2) to provide fine-grain lineage. Figure 5 illustrates tag-based associations.

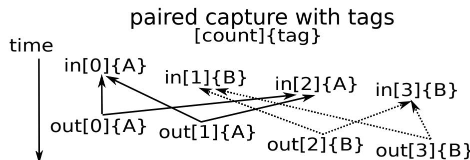


Figure 5: When the actor buffers inputs tags can tell Newt about finer-grain associations.

3.3 Tracing and anonymous transforms

To express tracing queries, Newt provides an imperative function $F = \text{Trace}(r[], \text{dir}, p_{root})$ that takes as input a set of target data elements $r[]$, a direction, and an actor instance p_{root} that produced $r[]$. Trace returns F, a set of discovered *tracing dataflows* whose granularity is equal to or contained by p_{root}. Each tracing dataflow is a topologically sorted set of actors that processed the traced data, its precursors, or its derivations. The tracing dataflow contains per-actor *tracing tables* $\mathtt{T}_p$ with the data inputs (a backward trace) or outputs (a forward trace) found during the trace. Tracing dataflows and tables are sub-graphs and sub-sets of the original dataflow and association tables, respectively.

Tracing uses the original dataflow for actor p as a scaffold on which to perform the tracing query. In most cases, tracing is simply a recursive sequence of equality joins between the association tables of directly connected actors in the dataflow graph. However, because DISC systems often lack formal schemas; downstream transforms often *re-interpret* input data in a different way than it was written. For instance, an output web page may be read as a sequence of input strings. Such *anonymous transforms* confound tracing—the instrumentation will not hash identical bytes or fields.

Here we observe that many transforms in DISC dataflows read and write to data types that are *locatable*. A locatable data type, such as a file, allows actors to reference sub-parts of an instance of that type. For example, Hadoop record writers thus identify their outputs with both a hash of the record and a *location specifier* L consisting of a (file, offset, length) triple. Newt can

[4] While not shown, addInput and addOutput can take sets of tags.

then perform tracing queries by comparing overlaps in location specifiers. Other locatable data types include network (TCP) streams and table-based stores. We discuss efficient ways to perform this tracing procedure in Section 5.1.

3.4 Replay, ex-replay

Newt allows users to replay backward tracing queries to recreate specific outputs or ex-replay a backward trace to avoid errors by removing data. In both cases, Newt improves performance by turning off lineage capture. In Newt, replay is guaranteed to produce the target output elements of a backwards trace.[5] However, there may in fact be additional records in the output. In fact, if all transforms are *monotonic*, i.e., if for any two inputs sets $I' \subseteq I$ then $P(I') \subseteq P(I)$, then these additional records will be members of the original output.

However, replaying a dataflow with *non-monotonic* transforms must be done with care. In this case, a replay of a dataflow with one non-monotonic function will produce the traced outputs o, but potentially create additional, unseen outputs. A replay with two or more non-monotonic functions may not produce the original output altogether. In order to correctly replay these dataflows, any additional outputs must be filtered at each transform during replay [10]. In Section 5, we describe how Newt instrumentation provides *input* filtering for each actor, ensuring that an actor only processes the lineage of the data to be replayed.

Note that in the case of ex-replay, these filters ensure that actors exclude data found in the tracing dataflow. Currently an ex-replay only excludes data at *one* logical stage in the dataflow. Unlike regular replay, we want to incorporate changes to outputs, even in the presence of non-monotonic operators. Thus, the dataflow will execute with the original universe minus the lineage at actor instance p: $\bigcup_p -\mathrm{T}_p$. We use such replays as the basis for data cleaning in Section 7.3.2.

4 Analytic debugging with Newt

Beyond tracing and replay, Newt provides additional APIs for debugging. First, Newt supports a fail API call for error or crash culprit determination. Developers may insert this call into exception handlers, allowing Newt to record fault-causing inputs (stored as a separate table within Newt). We use this, for example, in the Hadoop Map exception handler that runs when inputs cause faults. Users may specifically query for failed actors and identify error-inducing inputs.

[5] This is not true for forward-trace replays; such traces do not contain the complete set of inputs required for a downstream record.

4.1 Mining lineage for anomalies

In complex analytics, a user may be suspicious of the dataflow whether or not they have a known bad output. To address these cases, Newt provides mine interface to collect a set of lineage associates for data mining. In particular, our hypothesis is that changes in output multiplicity (the number of outputs associated to each input) can identify anomalous actors. For instance, a map task may incorrectly parse an input record (producing zero outputs) or correctly parse a bogus input record (producing many more outputs) [19]. Thus our anomaly detection algorithm (Section 7.3.1) calculates output multiplicity for actors in a dataflow, runs an outlier detection algorithm, and returns a ranked list of suspect actors.

However, this anomaly detection process is sensitive (both in accuracy and computation cost) to the *set* of lineage associations considered. The purpose of mine is to create an informed subset of lineage to mine for anomalies. Users call mine($p_{root}, o_{bad}, O_{good}[]$) with a root actor instance and two optional parameters. These further refine the number and size of returned association tables. If the user only specifies p_{root}, mine behaves like trace; it identifies one or more *mine dataflows* rooted or contained by p_{root} and produces per-actor *mine tables* that contain all of the original associations.

To reduce the number of suspect actors, the user can provide a known bad output, o_{bad}. In this case, Newt performs a backward trace on o_{bad} and only returns the actors found in the resulting tracing dataflows. The mine tables for each found actor still contain all original associations. Finally, the user can supply a set of known good outputs, $O_{good}[]$, to find baseline behavior. This version refines the mine table associations in the bad output trace to those found in the union of lineage of o_{bad} and $O_{good}[]$. Figure 6 illustrates this winnowing process.

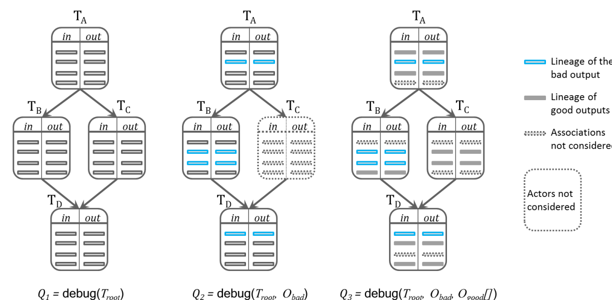


Figure 6: Users use mine to find a relevant subset of lineage to study. The inclusion of bad and known good outputs further winnows the lineage returned.

4.2 Data cleaning with ex-replay

In our original telecom analytic (Figure 1), the presence of malformed or unexpected inputs led to erroneous results and dataflow crashes. Combining the backward trace of a bad output with ex-replay, Newt makes it trivial to re-run an analytic without faulty inputs. For instance, consider iterative machine learning algorithms, which may execute tens or hundreds of processing steps [1]. Many such analytics produce a set of outputs that can be ranked by quality, including clustering algorithms, collaborative filtering, and item recommendation. Input data may be polluted by data with corrupt measurements, that came from untrusted sources, or simply contained unwanted outliers.

This indicates a general approach that ranks outputs by quality, finds the input data that created the lowest quality outputs, and ex-replays the analytic to improve the results. Note that during ex-replay Newt avoids the majority of Newt overheads since it does not re-collect any lineage. We build a proof-of-concept of this idea by applying it to the area of de novo genome assembly, which we describe in Section 7.3.2.

5 System details

Before Newt can be used, the system developer must first declare a set of actor and data types and arrange each into logical containment hierarchies (Section 3.1). When an actor starts, it registers with the Newt controller by providing an actor type g and a set α of unique instance identifiers or *id*'s of parent actors. The controller inserts this information into the actor instance table `AInstance`, returns a unique *id*, and chooses a peer on which to replicate the association logs (if desired).

When the actor finishes processing it calls `commit`, which tells Newt that this actor's log is complete, and the controller may begin to import the log as a SQL `A` table. Note that Newt can only answer tracing queries after all actors commit. To do so, Newt periodically sweeps the actor instance table for actor commits, indicating that the contained actors have finished and that they may be arranged into a dataflow. Note that Newt uses an exponential weighted moving averages of CPU and disk utilization to determine peer selection and SQL table creation respectively.

5.1 Tracing query optimization

Newt includes a distributed tracing query engine that federates the set of MySQL stores across the cluster. It performs the output-input matching required by tracing queries with a simple sequence of relational joins. This engine also performs the lineage analysis queries we discuss in Section 7.3. Newt optimizes query execution by placing A tables according to the discovered dataflow. The policy should try to minimize network traffic by co-locating tables while leveraging the combined processing capacity in the cluster. We found that assigning tables to the same (randomly chosen) node if directly connected in a dataflow outperformed strategies that grouped tables by actor type for all tested tracing queries.

Finally, as mentioned in Section 3.3 Newt uses locatable data types to create dataflow connections across anonymous transforms. To do so, Newt builds an association table to connect the actor instance p_{out} with p_{in}. We call this a ghost table, since an unseen actor (Newt) has translated between them. Ghost tables are given unique actor ids, and are otherwise treated as actor instances. Newt scans both actors' `A` tables, comparing the location specifiers L for each pair of records (H_{out}, L_{out}) and (L_{in}, H_{in}). In case of overlap, Newt inserts the row (H_{out}, H_{in}) into the ghost table.

5.2 Replay

Here we describe how Newt ensures accurate and efficient replay. A corner stone of accurate replay is the ability for each actor to filter data inputs. By filtering input data, the actor only processes inputs that existed in the prior execution, ensuring Newt's ability to correctly replay dataflows that contain non-monotonic transforms (Section 3.4). Newt filters data by installing the tracing table at the client when a replayed actor registers with the controller. A Newt actor filters data by observing the boolean return value of calls to the capture API, `filter`. If `filter` is true, then the actor must drop the data in question, i.e., it was not found in the tracing table. This arrangement allows replay to proceed with no further interaction with the Newt controller. As we show in the evaluation, the tracing table's row count is often a small fraction of the data observed in the original run.

Users may either manually re-execute the dataflow in replay mode or have Newt restart actors[6]. For each replayNewt uses the tracing query output to decide how to replay the dataflow. Newt checks for actor instances that are both *restartable* and that take *materialized* input data. Our current replay engine determines the minimum number of actors to restart by finding the highest-level restartable actors with materialized input. While this replay enables step-wise debugging through each actor, it may not be the fastest replay to recreate lost outputs. We intend to explore other replay strategies in the future.

[6] Restartable actors must implement a `restart(name,conf)` RPC method. The `name` is the same name recorded in the `AInstance` table when the original instance registered and `conf` is Newt-opaque actor configuration data.

5.3 Fault tolerance

It is important for Newt to provide fail-over capability since long-running dataflows are expensive to re-execute. First Newt attempts to ensure complete lineage capture if a Newt peer fails. Note that clients backup lineage logs to a secondary peer, and on peer failure, the Newt controller will make the secondary peer the primary for those log records. The controller will also re-distribute any lost SQL tables by re-importing them from the new primary. Note that our current design assumes that failed DISC tasks restart from the beginning, a common design for many architectures.

The Newt controller manages a number of tables in a local SQL database, but it can rebuild its state from local storage and information stored across the peers. Peers will store a copy of each actor's registration in the actor instance table. If the controller's failure is catastrophic, it can rebuild its entire state, including `AInstance` and the committed actors from information on the peers. To prevent lineage capture from stalling during reconstruction, one could employ a hot standby.

Note that Newt must also gracefully deal with DISC systems that run speculative actors, such as Hadoop. Successful actors call commit, and the Newt controller places a committed mark in the `AInstance` table indicating that all child instances started by this actor must also have committed. In these cases, the DISC system only commits the "winning" actor and Newt will only import and index that actor's logs. When the parent actor commits, Newt can garbage collect any un-committed actor's log files.

6 Instrumenting DISC systems

This section describes how we instrumented Hadoop and Hyracks [5] to capture fine-grain provenance.

6.1 Hadoop

Job controller First, the Hadoop job controller uses the Newt capture API (Table 1) to register and link instances of record reader, map, reduce, and record writer actors. The job controller calls commit when those instances complete successfully. Finally, the job controller implements the restartable API, allowing Newt to resubmit jobs.

Map and reduce tasks Record readers implement a simple interface `next(k,v)` that consumes a portion of the raw data (e.g. a line in a text file) and outputs the next (k,v) pair. Record readers use a *File Locatable* input data type, which uses a location specifier of the form L=(file, offset, length). The output data type is a *Record*, though Hadoop treats it as a (k,v) pair. We use the capture API to send L and a hash of the (k,v) pair as provenance. Record writer instrumentation is symmetrical.

Maps read input M_{in}=(k,v) pairs from the record reader and call the user-supplied map function. A user's map function reads M_{in} pairs and emits M_{out}=(k,v) pairs to the output using Hadoop's `Context` interface. We instrumented the Context mechanism to intercept emitted M_{out} pairs and record the pair (hash(M_{in}), hash(M_{out})) as provenance, using the capture interface.

Reducers call a user-supplied reduce function for every group of values. Unlike the map function, the reduce function takes a key and value list, $(k,v[])$. As described in Section 3.2, Hadoop passes the value list as an iterator to the user's reduce function. Thus a reduce output may depend on any prefix of input values, and we capture this dependence using the paired capture interface. We have instrumented the Context mechanism to (i) intercept input values and hash and record them using the addInput interface and (ii) intercept emitted output (k,v) pairs and record them using the addOutput interface. Upon exit from the `reduce` function, we call reset, to prepare the system for the next `reduce` call.

6.2 Hyracks

Hyracks is a generalized dataflow engine roughly in the same spirit as Dryad [12] where programs are DAGs of operators. However, the majority of the instrumentation is similar to Hadoop's. For instance we instrument the Hyracks job controller to track the lifecycle of operators with calls to register,flow_link,and commit. In addition, we instrumented the Hyracks operators that are analogs to Hadoop's record readers and writers: the `FileScan` and `FileWriter` operator respectively.

The interesting Hyracks operator is `HashGroup`, which implements a hash-based grouping operator and applies a user-defined aggregation function per group. `HashGroup` first reads an un-ordered set of input tuples and builds a hashtable based on a grouping attribute. After reading all input tuples, it then updates the aggregate (e.g. a count) in each hashtable entry and emits an output tuple. Thus, `HashGroup` buffers all inputs before emitting a single output, and we use the tagged capture API to logically partition input tuples as they are read. As before, we intercept input reads, but now we associate every input tuple with a tag that is a hash of the grouping key, calling addInput(r_{in},tag). For every output tuple, we call addOutput(r_{out},tag) to associate with input tuples that correspond to the specific tag only, and subsequently call reset(tag) to reset the association for the tag.

7 Evaluation

Our evaluation first establishes the impact of provenance capture on a range of workflows in terms of space and time overheads. A series of experiments quantify the accuracy of provenance Newt collects, and how that translates into efficient replay. We then highlight the use of the mine API to pinpoint faulty stages in a multi-join dataflow. And finally we demonstrate how tracing and ex-replay can be used to clean an input data set for the Contrail de novo genome assembler, ultimately improving output assembly quality.

We instrument version 0.20.2 of Hadoop, adding 53 lines of instrumentation to the job controller, 9 lines to the map object, and 11 lines to the reduce. For Hyracks (version 0.1.8) we added 60 lines to the job controller, 10 lines for FileWrite, 15 lines for FileScan, and 20 lines for HashGroup. In our Newt prototype, each peer uses MySQL version 5.5 as the peer relational engine. The Newt controller is written in 3.1k lines of Java.

Unless noted otherwise, all experiments use a 17-node cluster of Dual Intel Xeon 2.4GHz machines with 4GB of RAM, a single SCSI disk, and connected by gigabit Ethernet. One node runs the job tracker and Newt controller while the other 16 nodes run Newt peers and Hadoop slaves. While we do not report Hyracks results, our experiments confirmed similar runtime overheads as Hadoop for our microbenchmarks.

7.1 Capture overheads and selectivity

We first measure the impact of recording fine-grain provenance on execution performance. In general, the observed overhead is a complex function of how many records an actor creates from an input, as well as the amount of time the instrumentation overhead will be amortized against, such as per-record CPU processing and data shuffling on the network. For example an identity map-only job is a worst-case situation for Newt, exhibiting a 124% run time overhead. However, most jobs actually process inputs. To get a feel for a range of common Hadoop transforms, we first ran PigMix (version 1 on Pig 0.9.1) on our instrumented Hadoop cluster. PigMix generates a variety of Pig queries, including joins, orderby, and unions.

Figure 7 illustrates the runtime overhead for both timed and non-timed capture when compared to an unmodified Hadoop. For all queries timed capture outperforms non-timed capture. In the cases where non-timed capture overheads were very large, we noticed that the Pig jobs reduce tasks had few groups with very large numbers of values, causing Newt instrumentation to create many explicit associations. Timed capture defers that computation (and space usage) to post-capture, and exhibits overheads of 12-49%.

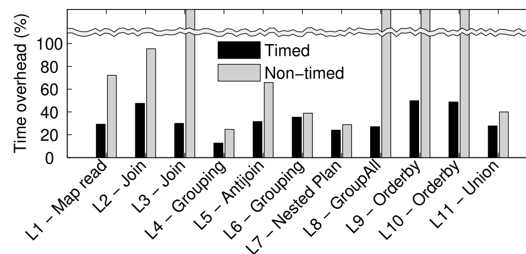


Figure 7: The run time overhead of individual PigMix jobs with and without timed capture. The break indicates where non-timed capture overheads exceed 100%.

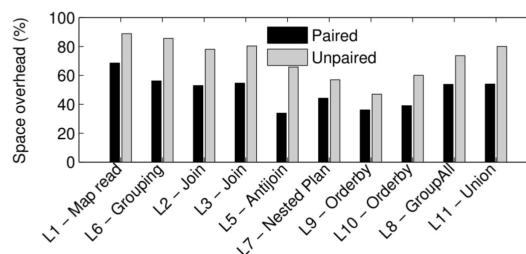


Figure 8: Space overhead (relative to total output data) of storing provenance for individual PigMix jobs.

Next we study the benefits of using paired vs. unpaired capture. Figure 8 shows the storage overhead as a percentage of the size of the output data for the same benchmark. Here we calculate the total number of bytes Newt uses to store all association A tables as compressed, serialized SQL tables. Note that this includes two indices for each A table (on input and output), which adds an average 10% overhead on table size. Here paired capture creates fewer associations, improving lineage accuracy and reducing storage overhead by 31% on average.

For many scenarios, we believe these overheads are reasonable when Newt is used for tuning and debugging. While space overheads approach 70% for paired capture, storage is cheap relative to additional CPU time. However PigMix experiments consist of short dataflows (2-3 jobs) that mostly use relational transforms. Many interesting "big data" analytics, such as processing large-scale graphs, involve non-relational tasks and dataflows with many stages.

To test Newt on real-world dataflows, we ran Contrail [17], a de novo genomic assembler, and two collaborative filtering analytics from the Mahout machine learning library [1]. Contrail attempts to recreate the original genome from a large set of short DNA sequences called "short reads." It uses an existing assembly method, de Bruijn graph assembly, to build a large graph and then repeatedly refine it to discover large contiguous regions

Analytic	# jobs	# actors	overhead(%)
Contrail	145	34927	20%
Item Similarity	7	21	31%
Recommender	10	30	36%

Table 2: Time overhead for real-world analytics. We run Contrail and two collaborative filtering applications from the Apache Mahout project.

of the genome. Contrail experiments leverage Amazon's EC2 service, running on 64 large instances with 7.5GB memory, 4 EC2 compute units, and "high" I/O performance. Mahout jobs ran on our 17-node cluster described previously.

Table 2 shows the number of jobs, captured actors, and time overheads for these analytics. The Contrail dataflow produce over 20 times as much lineage per second as the most intensive PigMix program. Assembly of our bacterial genome used 145 MapReduce jobs and Newt recorded provenance from 34927 actor instances. However, running with Newt increased run time by only 20% using the same hardware footprint as the original run. The assembly created 306GB of intermediate and final data; relative to this amount Newt incurred space overheads of 86%.

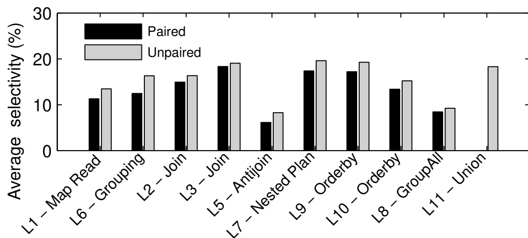


Figure 9: The tracing selectivity for PigMix jobs.

Last we study *trace selectivity*, the per-actor ratio of the tracing table size to the A table size, for both PigMix and for Contrail. Trace selectivity is a metric for determining lineage accuracy and gives us a lower bound on the amount of work that would be performed to replay the dataflow. Figure 9 shows the average percentage of input items that a randomly selected output depends upon for PigMix. By this measure paired capture improves accuracy (12% over original), with all traces selecting less than 18% of the original input.

7.2 Replay

This section investigates the ability of Newt to accurately and efficiently reproduce outputs. These experiments used a Word Count job with an input of 635,000 lines that created 1,173,443 output records. We choose output records to recreate at random, and verified that the reproduced output was indeed exact. Beyond running time, we use a range of metrics to measure the "accuracy" of the replay. Recall that a replayed dataflow may produce more records than those that were originally requested. Those records will be in the original output, but they represent potentially unnecessary work. To capture accuracy, we observe the relative number of these unnecessary records the replay creates.

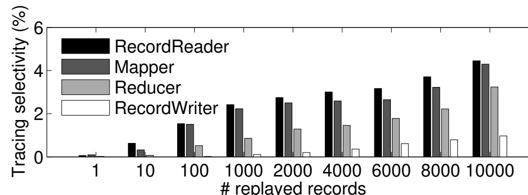


Figure 10: The percentage of data required to replay Word Count for each logical actor in the dataflow.

Figure 10 summarizes tracing selectivity for each logical actor. As expected, the selectivity increases as we move from the output to input. For instance, the record reader must replay at least 4.5% of the input to reproduce 10E3 records, while the record writer processes less than one percent (the exact size of the output).

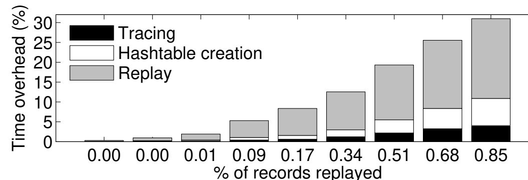


Figure 11: The percentage of normal time required to reproduce a fraction of the output data set. Note the absolute count of reproduced records is identical to Figure 10.

Lastly, we look at replay running time. Figure 11 plots the percentage of normal running time as a function of the fraction of the output data set to regenerate. While the replay is not as fast as a scaled version of the original execution, for small numbers of records—less than a 100—this holds true for Word Count. In fact a single record replay can execute in 0.3% of the original execution time. Thus for debugging, where quick testing is important, or regenerating small sets of output (from a disk latent sector error), it can be vastly less time intensive to regenerate the outputs with Newt.

7.3 Dataflow debugging

This section explores the effectiveness of our anomaly detection facility using a microbenchmark three-stage join dataflow. We also present a proof-of-concept ap-

plication of our data cleaning methodology to clean and improve a genome assembly.

7.3.1 Finding selectivity anomalies

Recall that we perform actor anomaly detection by calculating the output multiplicity (number of outputs per input) across a subset of lineage records found through the mine API (Section4.1). To identify anomalous actors, Newt uses a modified version of Grubbs' outlier detection technique [9] that can find multiple outliers in the dataset. Our anomaly detector works in two phases. It first examines individual actors for changes in multiplicity across their inputs. It then compares the average multiplicity of each actor instance to other instances of the same actor type. Newt ranks and places these outliers in the suspect set. We note that more sophisticated methodologies may leverage other statistics or compare across dataflow executions.

We test our anomaly detection technique on a dataflow that uses a sequence of three MapReduce jobs to join four different datasets. The dataflow has a total of 42 actor instances and we instrument two map instances (one in the second and one in the third MR job) to produce spurious outputs for specific input keys. The number of spurious outputs produced is called the *error magnitude*.

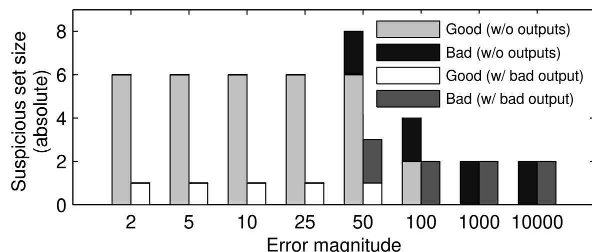


Figure 12: This figure shows the size of suspect actors ("Good") and found faulty actors ("bad") for anomaly detection with and without a known bad output. Using a bad output to winnow the analyzed lineage results in smaller suspect sets (there are 42 actors in total).

We measure the efficiency of anomaly detection by the size of the suspect actor set and the presence of the faulty actor. Figure 12 shows the size of the suspect set (the "Good" bar) and the number of bad actors present (the "Bad" bar) as we vary the error magnitude. We run the detector on two sets of lineage: "w/o outputs" calls mine without any bad outputs and "w/ bad output" calls mine with a known bad output. First, no test finds a bad actor until they inject over 50 spurious outputs. At this point, mining either lineage set discovers the faulty actors, but winnowing the lineage by specifying a bad output reduces the number of false positives (suspect but not guilty actors) from six to one.

Finally, we performed a similar experiment, but further reduced the lineage set returned by mine by also providing *good* outputs. This improved the ability of Newt to *rank* the suspected bad actor, moving the faulty actors (when found) from the 50th percentile to the 80th. Moreover, providing more information (bad and good outputs) limits the number of actors and associations Newt considers. This decreases the anomaly detection runtime by 85% relative to the general anomaly detection query.

7.3.2 Data cleaning genome assembly

A key software tool for building and understanding the human genome is de novo genome assembly [4]. Because it is difficult for gene sequencing machines to read the entire 3 billion character human genome string, they instead read short pieces of a cell's DNA. These are called short reads and they are typically ≤ 200 base pairs (bp) in length (i.e., A, C, T, or G). For the human genome, de novo assembly takes a set of 200-300 billion short reads and tries to re-create the original genome. In reality, assemblers output a handful of *contigs*, the assembler's best guess as to the original genome, and these output contigs can vary greatly in quality and length.

Like our telecom example, though, the input data set can contain bad data. In this case, the pollution is DNA from organisms other than the target genome [18], and even small amounts of pollution (<5%) affect assembly [8]. Here we use Newt to clean the input dataset and re-execute the assembly, with the goal of improving output contig quality.

Our first task is to produce a polluted input data set. Here we use Contrail to assemble a bacterial genome while simultaneously polluting the input data set with DNA from a viral genome. Using freely available software to simulate the input short reads [3], we produced an input data set with a 2.8% contamination rate, lower (and more challenging) than the 5% rate used for a recent assembly bake off [8].

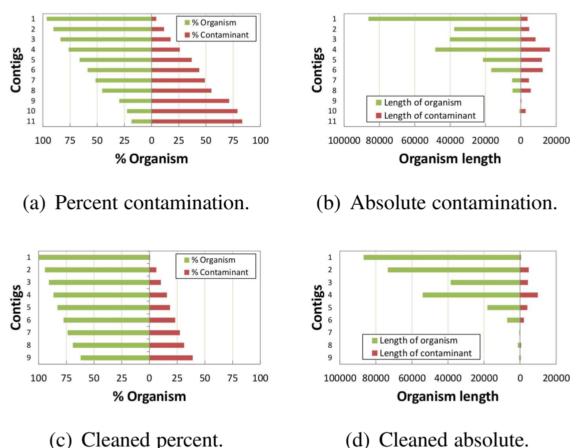
Next we run the assembly on the simulated data set and assess output contig quality. For each output contig, we used Nucmer (mummer.sourceforge.net), a package for aligning large DNA sequences, to measure the percentage of bases that match to the original bacterial or viral sequence. We flagged any output contigs whose DNA was < 30% bacterial as "bad." We then use Newt to backward trace these bad output contigs to identify bad input data throughput the pipeline.

Unsurprisingly, we found that each bad output contig depended upon over 95% of the original input short reads. Thus we needed to identify an intermediate stage from which to remove bad data. To do so, we analyzed the lineage at each successive MapReduce job, looking for a stage where there was minimal overlap between the

lineage of the bad output contigs and the lineage of the very best output contig. Through this empirical process we determined that it was best to remove data and restart from the 35th job (out of 51).

These experiments were performed on an AWS EC2 cluster of 15 large instances (with 1 master). The original execution took 6626 seconds, used 51 MapReduce stages, and produced 11 contigs, three of which we flagged as bad. Restarting without removal from that point took 2471 seconds, with ex-replay it took 1986 seconds, a 19% reduction in runtime.

Figure 13 shows the results of this process. We look at the percent and absolute amount of viral contamination in the output contigs before and after we used ex-replay to clean the data pipeline. Comparing the before picture (Figure 13(a)) with the after ex-replay (Figure 13(c)) (the other figures show absolute contig lengths), data cleaning not only removes large percentages of bad data, but also improves the average length of the returned contigs.



(a) Percent contamination. (b) Absolute contamination.

(c) Cleaned percent. (d) Cleaned absolute.

Figure 13: This percent and absolute contamination of the assembled contigs with bacterial and viral DNA before (top) and after (bottom) we use Newt to remove bad data. Removal improves contig quality and length.

8 Conclusion

Understanding, debugging, and managing the increasing volumes of derived data may soon become the limiting factor for data-intensive analytics. While Newt's use of timed capture dramatically lowers the overhead of active collection, it remains relatively expensive. However, the power of having all lineage "at hand" allowed us to rethink traditional techniques, and introduce new lineage-based data mining and data cleaning strategies. We hope Newt can serve as a platform for further refinement of active capture (e.g., through sampling) and exploring novel debugging techniques.

9 Acknowledgements

This work was supported in part by National Science Foundation grant CCF-1048296.

References

[1] Apache Mahout: Scalable Machine Learning and Data Mining. http://mahout.apache.org.

[2] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, and J. Stoyanovich. Putting lipstick on a pig: Enabling database-style workflow provenance. In *Proc. of VLDB*, August 2011.

[3] F. E. Angly, D. Willner, F. Rohwer, P. Hugenholtz, and G. W. Tyson. Grinder: a versatile amplicon and shotgun sequence simulator. *Nucleic Acids Research*, pages 1–8, 2012.

[4] M. Baker. De novo genome assembly: what every biologist should know. *Nature methods*, 9, 2012.

[5] V. Borkar, M. Carey, R. Grover, N. Onose, and R. Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of ICDE*, 2011.

[6] J. Cheney, L. Chiticariu, and W.-C. Tan. Provenance in databases: Why, how, and where. *Foundations and Trends in Databases*, 1, 2009.

[7] Y. Cui and J. Widom. Lineage tracing for general data warehouse transformations. *VLDB Journal*, 12(1), 2003.

[8] D. Earl, K. Bradnam, and J. S. John. Assemblathon 1: A competitive assessment of de novo short read assembly methods. *Genome Research*, 2011.

[9] F. Grubbs. Procedures for detecting outlying observations in samples. *Technometrics*, 11:1–21, 1969.

[10] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. In *Proc. of CIDR*, January 2011.

[11] R. Ikeda, S. Salihoglu, and J. Widom. Provenance-based refresh in data-oriented workflows. In *ACM Conference on Information and Knowledge Management*, 2011.

[12] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly. Dryad: Distributed data-parallel programs from sequential building blocks. In *Proc. of the European Conference on Computer Systems (EuroSys)*, March 2007.

[13] C. Olston and B. Reed. Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows. In *Proc. of VLDB*, August 2011.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proc. of ACM SIGMOD*, Vancouver, Canada, June 2008.

[15] C. Olston and A. D. Sarma. Ibis: A provenance manager for multi-layer systems. In *Proc. of CIDR*, January 2011.

[16] A. D. Sarma, A. Jain, and P. Bohannon. PROBER: Ad-Hoc Debugging of Extraction and Integration Pipelines. Technical report, Yahoo, April 2010.

[17] M. Schatz, A. Gupta, R. Gupta, D. Kelley, J. Lewi, D. Nettem, D. Sommer, and M. Pop. Contrail: Assembly of Large Genomes using Cloud Computing. http://sourceforge.net/apps/mediawiki/contrial-bio.

[18] J. Simpson and R. Durbin. Efficient de novo assembly of large genomes using compressed data structures. *Genome Research*, 2011.

[19] W. Zhou, Q. Fei, A. Narayan, A. Haeberlen, B. T. Loo, and M. Sherr. Secure network provenance. In *Proceedings of 23rd ACM Symposium on Operating System Principles (SOSP)*, December 2011.