UNIVERSITY OF CALIFORNIA, SAN DIEGO

Newt: An Architecture for Lineage-based Replay and Debugging in DISC Systems

A Thesis submitted in partial satisfaction of the requirements
for the degree Master of Science

in

Computer Science

by

Soumyarupa De

Committee in charge:

Kenneth Grant Yocum, Chair
Geoffrey M. Voelker
Stefan Savage

2012

The Thesis of Soumyarupa De is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

_____

_____

_____
Chair

University of California, San Diego

2012

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ACKNOWLEDGEMENTS

I would like to thank my adviser Ken Yocum for his invaluable guidance. The journey to this thesis has been full of ups and downs, and he has supported me throughout with exciting ideas, enthusiastic encouragement and creative analogies for when things go wrong. I have immensely enjoyed working on my research and learned a lot throughout the process. I would also like to thank Dionysios Logothetis and Apurva Kumar who provided a great deal of help and advice on different aspects of this thesis.

Finally, I would like to thank my family and friends for their constant encouragement and best wishes, without which this would not have been possible.

ABSTRACT OF THE THESIS

Newt: An Architecture for Lineage-based Replay and Debugging in DISC Systems

by

Soumyarupa De

Master of Science in Computer Science

University of California, San Diego, 2012

Kenneth Grant Yocum, Chair

Data-intensive scalable computing (DISC) systems facilitate large-scale analytics to mine "big data" for useful information. However, understanding and debugging these systems and analytics is a fundamental challenge to their continued use. This thesis presents Newt, a scalable architecture for capturing fine-grain lineage from DISC systems and using this information to analyze and debug analytics. Newt provides a unique instrumentation API, which actively extracts fine-grain lineage across complex, non-relational analytics. Newt combines this API with a scalable architecture for storing lineage to accommodate the high throughputs of DISC systems. This architecture

enables efficient dataflow tracing queries across thousands of operators found in modern data analytics. Newt extends tracing with replay, enabling users to perform step-wise debugging or regenerate lost outputs at a fraction of the cost to execute the entire analytics. Newt further facilitates replay for re-executing analytics without bad inputs to produce error-free outputs. Finally, Newt also enables retrospective lineage analysis, which we use to identify errors in the dataflow using outlier detection techniques.

We illustrate the flexibility of Newt's capture API by instrumenting two DISC systems: Apache Hadoop and Hyracks. This API incurs 10-51% time overhead and 30-120% space overhead on workloads consisting of relational and non-relational operators, including a Hadoop-based de novo genomic assembler. Newt can also accurately replay selected outputs, which can reduce the time to recreate errors during debugging. We show that it incurs 0.3% of the original runtime when replaying individual outputs in a WordCount workload. Finally, this work shows the effectiveness of Newt's debugging methodology by pinpointing faulty operators in a dataflow.

Chapter 1

Introduction

The era of "big data" has arrived, ushered in by our ability to record and process large amounts of data. Users use data-intensive scalable computing (DISC) systems, which facilitate scale-out analytics, to mine this data. However, a major roadblock to utilizing the potential of big data with DISC systems is our ability to understand large-scale data analytics and debug them. This thesis investigates the challenges in analyzing and debugging data processing in DISC systems and introduces the notion of using *data provenance* as a viable approach to address this issue.

1.1 Motivation

In recent years, the amount of data in our world has exploded as increasing numbers of users and organizations generate and publish data through the Internet. This data being generated is used by a variety of different sectors, including social networking sites, real world events, scientific communities and businesses. For example, Google processes nearly 20 petabytes of data per day [9] and serves 35,000 queries each second [12] and Facebook generates over 130 terabytes of logs each day [13]. The Large Synoptic Survey Telescope is expected to generate terabytes of data every night and eventually store more than 50 petabytes [6], while in the bioinformatics sector, the largest genome

sequencing houses in the world now store petabytes of data apiece [11].

This information explosion has opened up a number of opportunities for organizations in different sectors to gain advantage from mining and analyzing big data. Businesses mine search trends, social media generated data, network traffic logs and financial data with the intent to extract market trends, improve ad quality in sponsored search [37], recommend relevant shopping ideas [15, 19], detect fraud [56] and botnets [32], and forecast economy [16]. Government agencies and scientific communities process large amounts of data for advances in healthcare [61], bioinformatics [28, 50, 44], climate predictions [4, 42, 57] and disease outbreak detection [14]. To mine such high volumes of continuously arriving information, large-scale data analytics are required. However, more than 80% of the data in the world is raw and unstructured [10], and the analytics used to process them are not purely relational. These requirements are different from those of traditional relational database management systems (RDBMSs), which assume well-defined structure in data and relational processing [59].

Instead, researchers now use DISC systems that are designed to scale to large volumes of data and accommodate sophisticated data transforms. The DISC architecture exploits parallelism from commodity clusters to achieve comparable performance to expensive servers. One powerful benefit of these systems is that they allow users to write custom functions, known as *user-defined functions* or UDFs, thus enabling a wide range of applications. Today, Google MapReduce [27], Microsoft Dryad [41], Apache Hadoop [1] (an open-source project) and Google Pregel [43] provide such platforms for businesses and users.

However, even with these systems, big data analytics can take several hours, days or weeks to run, simply due to the data volumes involved. For example, a ratings prediction algorithm for the Netflix Prize challenge took nearly 20 hours to execute on 50 cores, and a large-scale image processing task to estimate geographic information took 3 days to complete using 400 cores [25].

The massive scale and unstructured nature of data, the complexity of these ana-

lytics pipelines, and long runtimes pose significant manageability and debugging challenges. Even a single error in these analytics can be extremely difficult to identify and remove. While one may debug them by re-running the entire analytics through a debugger for step-wise debugging, this can be expensive due to the amount of time and resources needed. Auditing and data validation are other major problems due to the growing ease of access to relevant data sources for use in experiments, sharing of data between scientific communities and use of third-party data in business enterprises [55, 31, 54, 23]. These problems will only become larger and more acute as these systems and data continue to grow. As such, more cost-efficient ways of analyzing DISC system analytics are crucial to their continued use.

1.2 A case for provenance

To address such challenges, what we need is a means to track the lifecycle of each piece of data as it is ingested, processed and output by the analytics. This provides visibility into the analytics pipeline and simplifies tracing errors back to their sources. It also enables replaying specific portions or inputs of the dataflow for step-wise debugging or regenerating lost output (which can happen due to disk errors [20]). In fact, database systems have used such information, called *data provenance*, to address similar validation and debugging challenges already.

This thesis investigates the use of data provenance to analyze and debug DISC systems analytics. Data provenance broadly refers to a description of the origins of a piece of data and the process by which it was derived [23]; essentially, it allows users to associate inputs of a process to its outputs. DBMSs have explored the use of provenance to validate scientific databases, debug queries, clean data in data warehouses, understand and correct complex data integration transformations, and to understand the value of data in curated databases [60, 34, 22, 29].

The benefits of provenance in database systems suggest its use in DISC systems

to facilitate similar analysis and insight into the flow of data, or *dataflow*, through these systems. Provenance at the granularity of each piece of data, or *fine-grain* provenance, allows users to identify the individual sources of anomalous outputs [34, 55, 33]. The ability to trace data to their derived outputs is useful for auditing and ensuring policy compliance, while tracing data to their sources is useful in debugging [38, 48, 55]. Provenance can also enable fine-grain replay in DISC systems, i.e. replaying specific portions or inputs of the dataflow, which can be used for regenerating lost data, step-wise debugging, simulating what-if scenarios and data cleaning [36, 63, 65, 33].

But using provenance for DISC systems is challenging because unlike DBMSs, DISC systems are inherently designed to deal with unstructured data and complex computations, and their scale of operations can be arbitrarily large. DISC systems have many possible processing architectures, and a provenance collection interface must accommodate them all to enable wider applications. User-defined functions make these systems more challenging, as it can be hard to predict how they derive outputs from inputs, thus making it difficult to extract fine-grain provenance from them. Therefore, a provenance system designed for DISC systems must be able to scale, be generic and handle UDFs and arbitrary data, while still capturing fine-grain provenance and providing useful facilities for tracing, replaying and debugging dataflows.

This thesis describes the design and architecture of a scalable provenance collection system that addresses these challenges and enables fine-grain provenance capture from generic DISC systems and UDFs, supports efficient tracing and replay, and explores new approaches for using provenance for debugging complex DISC analytics.

1.3 Contributions

In summary, this thesis makes the following contributions:

- A generic instrumentation API that supports common operators in DISC systems and captures accurate provenance from arbitrary operators.

- A scalable architecture for efficiently collecting provenance from and tracing through DISC system dataflows. This architecture also enables fine-grain replay for regenerating specific outputs and step-wise debugging.

- Realization of the provenance model proposed by Ibis [46], with optimizations for scalability and accurate lineage capture.

- A technique to identify potential anomalies using dataflow characteristics observable from its provenance, such as selectivity[1]. We also show that our debugging methodology can identify multiple anomalies in a dataflow with reasonable accuracy.

- Instrumentation of two real-world DISC systems - Hadoop [1] and Hyracks [21], which demonstrates the flexibility of our capture API.

- Evaluations to measure performance overheads of provenance capture and replay on Hadoop and Hyracks. We incur 10-51% time overheads and 30-120% space overheads on simple PigMix benchmarks [8], a Hyracks [21] WordCount program, and a Hadoop-based genomic assembler [53]. We also incur less than 0.3% overheads on individual output replays, enabling efficient step-wise debugging.

[1] Selectivity is the number of outputs produced per input.

Chapter 2

Background

This chapter gives an overview of concepts and related work relevant to this thesis. Section 2.1 gives an overview of current approaches to processing big data and common DISC architectures. Section 2.2 describes data provenance, and the form of provenance we believe is required for analyzing DISC analytics. Finally, Section 2.3 summarizes the challenges this work must address to use provenance for DISC systems.

2.1 Big data processing today

To mine big data, users can write analytics and execute them in a DISC system. DISC systems abstract away the challenges of writing fault tolerant, load balanced and parallel code for users' programs. They leverage the parallel compute capacity of thousands of cores on commodity clusters by partitioning the input data into small chunks, and processing them in parallel over the entire cluster [27, 41, 43, 21, 1].

The basic unit of work a DISC system accepts is called a *job*, which consists of data and operators that process the data. DISC systems split each job into smaller *tasks*, typically consisting of a single logical operator, which are then executed in parallel on small chunks of data. Failures in DISC systems are handled at the level of each task, which may fail independently and are restarted [27, 41, 43, 21, 1]. This section describes

the different challenges to capturing and using provenance in DISC systems.

2.1.1 Layering of DISC systems

There are several different DISC systems available today, including Google MapReduce [27], Pregel [43], Apache Hadoop [1], Dryad [41] and Hyracks [21]. Each employs different architectures and abstractions to facilitate different kinds of analytics. To enhance the ease of deploying large-scale analytics on these systems, programming platforms have been developed, which define higher-level languages that compile to DISC programs and provide a wide range of pre-built operators that users can use and extend. Examples of such platforms include Pig [49], DryadLINQ [62], Hive [58] and SCOPE [24]. These, in turn, are combined in workflow systems like Nova [47] or Oozie [2].

Big data analytics can span several DISC systems and compile to long *multi-stage* dataflows (a stage is a logical step or group of operators within a dataflow). For example, an analytics deployed using Nova can consist of several Pig programs, each of which in turn compile to several MapReduce jobs. Here, each Pig program is an operator. Each MapReduce job is a lower-level operator within a Pig program, and each MapReduce task is a lower-level operator within each job. This poses significant challenges to integrating end-to-end provenance across these analytics, since provenance must be captured across multiple operators, on multiple levels of operators, and accommodate different DISC architectures. Subsequently, tracing must also scale to long dataflows and be able to answer queries across multiple operator levels.

2.1.2 Processing constructs in DISC systems

Many DISC systems share common processing constructs. To capture provenance for users' analytics, a provenance collection system must accommodate these constructs. Two common processing constructs in DISC systems are the *group-wise*

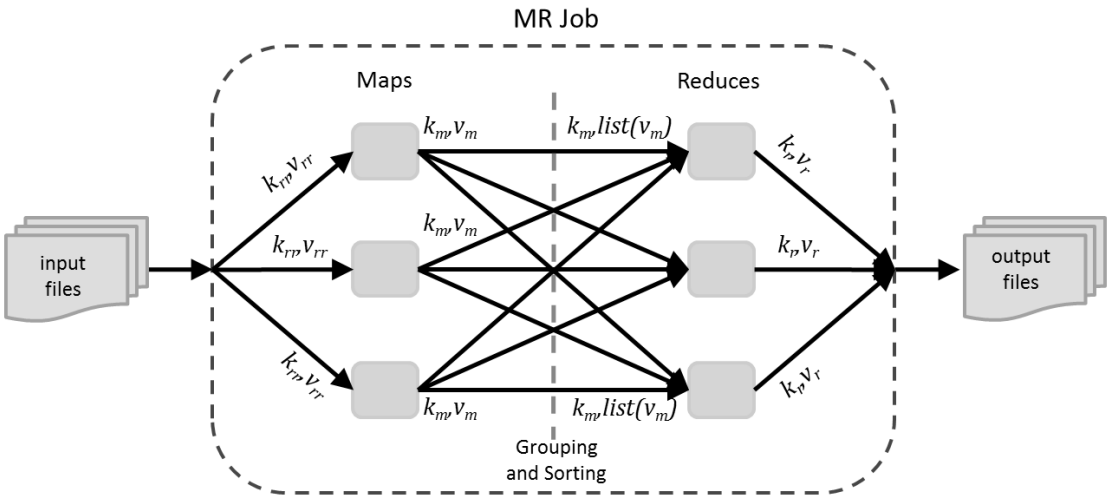


Figure 2.1: A MapReduce job consists of two distinct operators, map and reduce. This figure shows the dataflow across these operators and the schema of the input and output for each operator.

processing construct and the *map* construct [27, 41, 21, 1].

The map construct processes individual input elements, or an input *record*, independently from others, producing one or more output records. The group-wise construct processes values grouped by a key, and produces zero or more outputs. The map operator and the reduce operator in MapReduce [27, 1], are examples of the map and group-wise constructs respectively. Vertices in Dryad [41] and the HashGroup operator in Hyracks [21] are also examples of group-wise constructs. Group-wise processing also underlies common operators used in Pregel [43].

2.1.3 MapReduce architecture

Among the different DISC architectures, MapReduce is one of the most widely used, and is therefore, used in most examples in this thesis. This section gives a brief overview of this architecture.

A MapReduce job consists of a map and a reduce operator. Each operator also constitutes a task within the job[1]. The input to the job is set of files IF_i. The map

[1] Additionally, a map task also consists of a record reader. We view the record reader as an operator,

operator processes key-value pairs from these input files and for each input key-value pair, (k_{rr}, v_{rr}), it produces one or more output key-value pairs, (k_m, v_m). The output of the map operator is grouped (using a grouping key specified by the user, or the default grouping key used by the MapReduce implementation, which is k_m), sorted on the grouping key and then fed to the reduce. Thus, assuming the default grouping key is used, the reduce operator reads in $(k_m, [v_m^i])$, where $[v_m^i]$ is a list of v_m values. For each key-value-list pair the reduce reads in, it outputs one or more key-value pairs, (k_r, v_r) which are written to an output file of the job, OF_i. Figure 2.1 shows a MapReduce job and its tasks. At runtime, several physical instances of each operator exist within a job, due to a copy of each task executing on different machines in the cluster.

2.2 Data provenance

This thesis proposes the use of data provenance in DISC systems to trace records through a dataflow, replay the dataflow on a subset of its original inputs and debug dataflows. To do so, we need to track the set of inputs to each operator, which were used to derive each of its outputs. Although there are several forms of provenance, such as copy-provenance and how-provenance [23, 40], the information we need is a simple form of why-provenance, or lineage, as defined by Cui et al. [26].

Intuitively, for an operator T producing output o, lineage consists of triplets of form $\{I, T, o\}$, where I is the set of inputs to T used to derive o. Capturing lineage for each operator T in a dataflow enables users to ask questions such as *"Which outputs were produced by an input i on operator T?"* and *"Which inputs produced output o in operator T?"* A query that finds the inputs deriving an output is called a *backward tracing* query, while one that finds the outputs produced by an input is called a *forward tracing* query [38]. Backward tracing is useful for debugging, while forward tracing is useful for tracking error propagation [38]. Tracing queries also form the basis for

since it transforms data from files to records. Symmetrically, a reduce task consists of a record writer, which transforms a group of records into files.

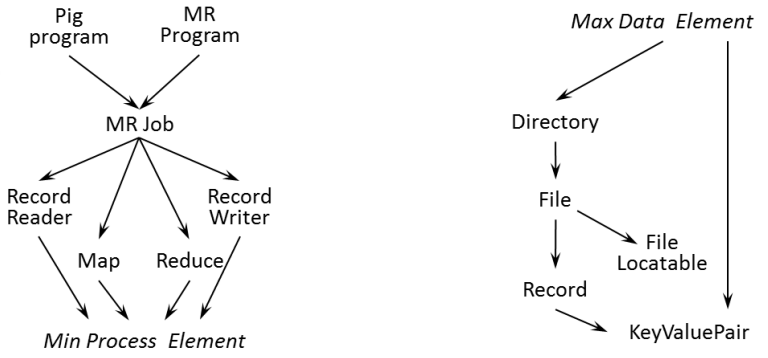
replaying an original dataflow [26, 31, 63, 65, 48, 38].

However, to efficiently use lineage in a DISC system, we need to be able to capture lineage at multiple levels (or granularities) of operators and data, capture accurate lineage for DISC processing constructs and be able to trace through multiple dataflow stages efficiently. The following sections describe the notions of lineage at different granularities, lineage accuracy, and different approaches to lineage capture and how they affect tracing performance.

2.2.1 Granularities in lineage

A DISC system consists of several levels of operators and data, and different use cases of lineage can dictate the level at which lineage needs to be captured. For example, consider the MapReduce job in Figure 2.1. While lineage can be captured at the level of the job, using files and giving lineage tuples of form $\{IF_i, MRJob, OF_i\}$, lineage can also be captured at the level of each task, using records and giving, for example, lineage tuples of form $\{(k_{rr}, v_{rr}), map, (k_m, v_m)\}$. The first form of lineage is called *coarse-grain* lineage, while the second form is called *fine-grain* lineage. Integrating lineage across different granularities enables users to ask questions such as "Which file read by a MapReduce job produced this particular output record?" and can be useful in debugging across different operator and data granularities within a dataflow.

To capture end-to-end lineage in a DISC system, we use the Ibis model [46], which introduces the notion of containment hierarchies for operators and data. Specifically, Ibis proposes that an operator can be *contained* within another and such a relationship between two operators is called *operator containment*. Operator containment implies that the contained (or child) operator performs a part of the logical operation of the containing (or parent) operator. For example, a MapReduce task is contained in a job. Similar containment relationships exist for data as well, called *data containment*. Data containment implies that the contained data is a subset of the containing data (superset). For example, a record is contained in a file. Figure 2.2 shows operator and data



(a) Operator containment hierarchy or gset. (b) Data containment hierarchy or gset.

Figure 2.2: Containment hierarchies or gsets in the MapReduce ecosystem.

containment hierarchies for the MapReduce ecosystem. While on one hand we have coarse-grain operator and data such as the Pig program and directories, on the other hand, we have fine-grain operator and data such as map and reduce tasks and records.

2.2.2 Lineage accuracy

One of the challenges of DISC systems is supporting the processing constructs found in these systems. We must be able to capture accurate lineage across these constructs to enable fine-grain debugging. A set of inputs I constitutes *accurate* lineage of an output o in an operator T, if executing T on I produces o and for each proper subset $I' \subset I$, executing T on I' does not produce o. On the other hand, a set of inputs I^c constitutes *complete* lineage of o if executing T on I^c produces output o. Thus, I is also a complete lineage of o. Notice that for a monotonic operator [38], any superset I^* of I, where I^* is a subset of the original input to T, may be used to derive o as well. Thus, I^* constitutes complete lineage of o, but not accurate.

For example, consider the reduce operator in Figure 2.1. It ingests input records $(k_m, [v_m^i])$ and for each such input record, it produces one or more output records, (k_r, v_r). Suppose the reduce operator produces one output for each value in the list of values $[v_m^i]$. Although, associating each output with the entire input record $(k_m, [v_m^i])$

would produce complete lineage, accurate lineage would associate only one value from the input list with each output. Accurate lineage is important for fine-grain debugging, as it enables the user to narrow down the subset of inputs that may be faulty. This is particularly important in long dataflows since it reduces the number of records that must be traced through multiple stages to identify the lineage of a bad output.

2.2.3 Active vs. lazy lineage collection

One of the requirements for debugging DISC dataflows is the ability to perform fine-grain tracing across multiple stages efficiently. Thus, it is important that we use a lineage capture approach that enables us to do so. Two common lineage collection approaches are active and lazy lineage collection.

Lazy lineage collection typically captures only coarse-grain lineage at runtime. For example, SNooPY [65] and Lineage Tracing in Warehouses [26] use lazy lineage collection. These systems typically incur low capture overheads due to the small amount of lineage they capture. However, to answer fine-grain tracing queries, they must replay the dataflow on all (or a large part) of its input and collect fine-grain lineage during the replay. This approach is suitable for forensic systems, where a user wants to debug an observed bad output.

However, to detect dataflow anomalies in the absence of known bad outputs, simulate what-if scenarios and perform fine-grain step-wise debugging, we require a different approach. Active collection systems capture entire lineage of the dataflow at runtime. The kind of lineage they capture may be coarse-grain or fine-grain, but they do not require any further computations on the dataflow after its execution. For example, PASSv2 [45] and Lipstick [18] use active lineage collection. Ikeda at al. use active collection in Provenance for Generalized MapReduce Workflows [38] as well. Active fine-grain lineage collection systems incur higher capture overheads than lazy collection systems. However, they enable sophisticated replay and debugging. For these reasons, we use active lineage collection.

2.3 Challenges

There are several challenges to capturing and using lineage in DISC systems. This section explores these challenges, and presents current approaches.

Scalability and fault tolerance. DISC systems are primarily batch processing systems designed for high throughput. They execute several jobs per analytics, with several tasks per job. The overall number of operators executing at any time in a cluster can range from hundreds to thousands depending on the cluster size. Lineage capture for these systems must be able scale to both large volumes of data and numerous operators to avoid being a bottleneck for the DISC analytics.

Lineage capture systems must also be fault tolerant to avoid rerunning dataflows to capture lineage. At the same time, they must also accommodate failures in the DISC system. To do so, they must be able to identify a failed DISC task and avoid storing duplicate copies of lineage between the partial lineage generated by the failed task and duplicate lineage produced by the restarted task.

Black-box operators. Lineage systems for DISC dataflows must be able to capture accurate lineage across black-box operators to enable fine-grain debugging. Current approaches to this include Prober, which seeks to find the minimal set of inputs that can produce a specified output for a black-box operator by replaying the dataflow several times to deduce the minimal set [52], and dynamic slicing [17], as used by Zhang et al. [64] to capture lineage for noSQL operators through binary rewriting to compute dynamic slices. Although producing highly accurate lineage, such techniques can incur significant time overheads for capture or tracing, and it may be preferable to instead trade some accuracy for better performance. Thus, there is a need for a lineage collection system for DISC dataflows that can capture lineage from arbitrary operators with reasonable accuracy, and without significant overheads in capture or tracing.

Efficient tracing. Tracing is essential for debugging, during which, a user can issue multiple tracing queries. Thus, it is important that tracing has fast turnaround times. Ikeda et al. [38] can perform efficient backward tracing queries for MapReduce dataflows, but are not generic to different DISC systems and do not perform efficient forward queries. Lipstick [18], a lineage system for Pig [49], while able to perform both backward and forward tracing, is specific to Pig and SQL operators and can only perform coarse-grain tracing for black-box operators. Thus, there is a need for a lineage system that enables efficient forward and backward tracing for generic DISC systems and dataflows with black-box operators.

Sophisticated replay. Replaying only specific inputs or portions of a dataflow is crucial for efficient debugging and simulating what-if scenarios. Ikeda et al. present a methodology for lineage-based refresh, which selectively replays updated inputs to recompute affected outputs [39]. This is useful during debugging for re-computing outputs when a bad input has been fixed. However, sometimes a user may want to remove the bad input and replay the lineage of outputs previously affected by the error to produce error-free outputs. We call this *exclusive* replay. Another use of replay in debugging involves replaying bad inputs for step-wise debugging (called *selective* replay). Current approaches to using lineage in DISC systems do not address these. Thus, there is a need for a lineage system that can perform both exclusive and selective replays to address different debugging needs.

Anomaly detection. One of the primary debugging concerns in DISC systems is identifying faulty operators. In long dataflows with several hundreds of operators or tasks, manual inspection can be tedious and prohibitive. Even if lineage is used to narrow the subset of operators to examine, the lineage of a single output can still span several operators. There is a need for an inexpensive automated debugging system, which can substantially narrow the set of potentially faulty operators, with reasonable accuracy, to minimize the amount of manual examination required.

Chapter 3

Design

The previous chapters propose the notion of capturing and using data lineage in DISC dataflows, and highlight the challenges involved in doing so. This thesis presents Newt, a lineage system for DISC, which provides a scalable architecture for capturing multi-granular lineage and performs efficient forward and backward tracing. Newt also performs fine-grain dataflow replays for step-wise debugging, and enables exclusive replay. Finally, Newt addresses debugging challenges in DISC systems, including determining bad input data responsible for program failure and detecting dataflow anomalies, with minimal assistance from the user.

Newt *actively* captures lineage by instrumenting a DISC system through a set of generic APIs. Specifically, to use Newt, a developer inserts instrumentation code into each *actor* in the system, where an actor is an entity in the DISC system that transforms data. For example, a MapReduce job and a Dryad vertex are both actors. Different executions of the same actor are called actor *instances*. The instrumentation APIs enable Newt to treat actors as black-boxes and tap their inputs and outputs to capture lineage in the form of *associations*, where an association is a triplet $\{i, T, o\}$ that relates an input i with an output o for an actor T. The instrumentation thus captures lineage in a dataflow one actor at a time, piecing it into a set of associations for each actor.

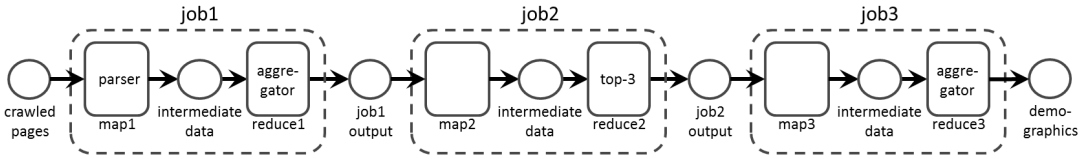
This chapter provides a design overview of Newt and presents specific examples

that motivate its API and interface design. Section 3.1 describes a running example and Section 3.2 describes debugging use-cases in the example dataflow and how lineage can be useful. Section 3.3 describes the different APIs Newt provides to capture lineage for different types of actors. Once lineage is captured, Newt must be able to use it for efficient tracing and replay. To do so, Newt stores the captured associations in a set of relational tables, called *association tables*, and maintains indexes on each table to optimize retrieval. Section 3.4 describes how Newt performs tracing across multi-stage dataflows. During the query, Newt arranges these association tables in a topological order that represents the original dataflow (Section 3.4.1) and connects the pieces back together to be able to trace each input or output through the dataflow (Section 3.4.2). Finally, Newt uses lineage tracing as a building block for replay by re-executing the dataflow based on the results of a tracing query (Section 3.5).

3.1 Running example

Before we proceed, let us consider a MapReduce application that calculates demographics for movie preferences, as shown in Figure 3.1, which we will use as a running example. The application parses a set of webpages, and processes the movie ratings information obtained, to produce demographics for each genre by age and gender.

The application consists of three MapReduce jobs (Figure 3.1a). The input webpages consist of several ratings for different movies by different users. job1 parses each rating entry, extracts the profile of the user and the genre, and for each unique user profile, it computes the average rating the user assigns to different movie genres (Figure 3.1b). job2 groups together the average ratings for different genres for each unique user and extracts the top-3 genres the user prefers (Figure 3.1c). job3 groups all input records by genre and computes demographics for each genre by age and gender (Figure 3.1d).



(a) This figure shows the detailed MapReduce dataflow for the movie preferences. The figures below show the details of each job in the dataflow, with data formats and field descriptions.

map1

Input: *rating entry*
Output:
Key: *user, gender, birthdate, genre*
Value: *rating*

Input	Output
Rating1	UserA, M, 01-01-1991, action, 4
Rating2	UserA, M, 01-01-1991, suspense, 5
Rating3	UserB, F, 02-02-1992, comedy, 4

map2

Input: *user, gender, birthdate, genre, avg_rating*
Output:
Key: *user, gender, birthdate*
Value: *genre:avg_rating*

Input	Output [(key), value]
UserA, M, 01-01-1991, action, 4.7	UserA, M, 01-01-1991, action:4.7
UserA, M, 01-01-1991, suspense, 4	UserA, M, 01-01-1991, suspense:4

map3

Input: *user, gender, birthdate, <top-3-genres>*
Output:
Key: *genre*
Value: *gender:age*

Input	Output [key, value]
UserA, M, 01-01-1991, action, comedy, suspence	action, M:21 comedy, M:21 suspense, M:21
UserB, F, 02-02-1992, comedy	comedy, F:20

reduce1

Input:
Grouping key: *user, gender, birthdate, genre,*
Value: *list(rating)*
Output: *user, gender, birthdate, genre, avg_rating*

Input [(key), {value list}]	Output
(UserA, M, 01-01-1991, action), {4, 5, 5}	UserA, M, 01-01-1991, action, 4.7
(UserA, M, 01-01-1991, suspense), {3,4,5}	UserA, M, 01-01-1991, suspense, 4

reduce2

Input:
Grouping key: *user, gender, birthdate*
Value: *list(genre:avg_rating)*
Output: *user, gender, birthdate, <top-3-genres>*

Input [(key), {value list}]	Output
(UserA, M, 01-01-1991), {action:4.7, suspense:4, comedy:4.5, horror:1}	UserA, M, 01-01-1991, action, comedy, suspence
(UserB, F, 02-02-1992), {comedy:4}	UserB, F, 02-02-1992, comedy

reduce3

Input:
Grouping key: *genre*
Value: *list(gender:age)*
Output: *genre, gender, avg_age, count*

Input [key, {value list}]	Output
action, {F:25, M:21, M:30}	action, F, 25, 1 action, M, 25.5, 2 action, all, 25.3, 3
comedy, {F:20, M:21}	comedy, F, 20, 1 comedy, M, 21, 1 comedy, all, 20.5, 2

(b) Details of job1. (c) Details of job2. (d) Details of job3.

Figure 3.1: An example application for calculating movie preferences demographics.

3.2 Use-cases for lineage

We consider some common debugging use-cases, which can occur due to bad data or actors in the dataflow. Note that we do not consider other causes of error, such as hardware errors or errors in the DISC system code. Errors due to bad or unexpected inputs can cause an actor to crash, or manifest as bad output. Errors due to unexpected or faulty actor behavior on certain inputs usually result in the actor producing bad outputs, which can then have effects similar to bad inputs in the dataflow. Apart from debugging, another useful application of identifying errors is *data cleansing*, which deals with removing errors and inconsistencies from data to improve its quality [51].

Newt provides five techniques that can be useful in debugging. Backward tracing enables an analyst (debugging user) to track errors to their sources, while forward tracing enables the analyst to track error propagation from a corrupt source. *Selective* replay enables the analyst to replay specific inputs for step-wise debugging. *Exclusive* replay enables the analyst to replay specific inputs in the absence of certain known bad inputs to update outputs previously affected by error. Crash culprit determination identifies inputs of an actor responsible for its failure. Finally, suspicious actor detection identifies (or narrows down) the subset of actors in the dataflow potentially responsible for an observed anomaly in the output. The following examples highlight how these techniques can be used to assist the analyst for different types of errors in the context of the movie preferences application.

3.2.1 Bad outputs

After the application finishes, suppose an analyst is surprised to note that the average age of users who prefer $horror$ is 5 years. The analyst suspects that there is an error, however, simply observing the output of job3 does not help her. What the analyst needs is to identify the source of the error.

Backward tracing. Using lineage, the analyst can backward trace the output record for $horror$, O_{hor}, to its sources. Suppose, she traces O_{hor} to its lineage in the inputs of job3, $IL(\text{job3}, O_{hor})$, and realizes that the values for the `birthdate` field in several input records is quite recent, obviously incorrect.

To identify the cause, the analyst traces the records in $IL(\text{job3}, O_{hor})$ to the webpage they originated from, say `BadOutputPage.com`, and realizes that the values assigned to `birthdate` during parsing in job1 actually denote the user's registration date with the website. Consequently, the analyst removes `BadOutputPage.com` from the application's inputs.

Forward tracing. However, the analyst must now remove all derived records, which originated from `BadOutputPage.com`, from job3's inputs. To identify such records, she forward traces `BadOutputPage.com` to the outputs of job2, which gives OL(job2, `BadOutputPage.com`), i.e. the set of inputs to job3 that originated from `BadOutputPage.com`.

Exclusive replay. Finally, the analyst must remove the records in OL(job2, `BadOutputPage.com`) from job3's input and replay job3 to produce error-free output. To do so, she uses exclusive replay to remove these records and successfully replay the application devoid of error.

3.2.2 Program crash

Suppose, instead, that the application crashed while running map3. The analyst checks relevant logs to identify the error and finds an unhelpful log message, "Could not parse date", which does not give her any insight into the cause of the error. What she needs is to find the inputs responsible for the crash and identify their sources.

Crash culprit determination. Using crash culprit determination, the analyst identifies the bad input record that caused the crash, say I_{crash}, and realizes that the value in the `birthdate` field is concatenated with a genre value, most likely due to incorrect webpage parsing during map1. To determine the cause, the analyst next backward traces I_{crash} to identify its originating webpage, say `CrashPage.com`.

Selective replay. Suppose the analyst cannot find anything wrong with `CrashPage.com`, but it is important that she identify the cause of the error to avoid similar crashes in the future due to other webpages. To do so, the analyst can use selective replay to replay only the lineage of I_{crash} through each actor. This enables her to perform step-wise debugging on a small set of inputs.

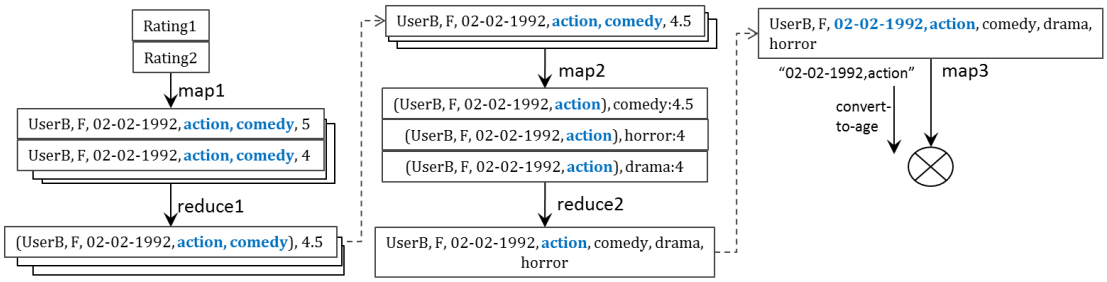


Figure 3.2: This figure shows the propagation of error due to multiple genres in the input to the application, eventually causing it to crash.

Thus, the analyst uses selective replay, and observes that `CrashPage.com` associates multiple genres with each movie, which causes map1 to produce records of form $\{UserB, female, 02{-}02{-}1992, action, comedy, 5\}$. These additional genres propagate through reduce1, map2 and reduce2 as part of the input key, and finally, due to naive string parsing at map3, which expects at most three genres per input record, the additional genres end up as part of the `birthdate` field, causing map3 to crash. This error propagation, which selective replay reproduces, is shown in Figure 3.2. The analyst, having identified the cause of the error, rewrites map1 with additional code that handles input webpages with multiple associated genres.

3.2.3 Suspicious actors

Suppose, the analyst is surprised to observe that the most preferred genre for females is $action$. She is not sure if this is actually true or an error. She backward traces the output record for $action$, O_{act}, to identify the webpages contributing to it, finds nothing suspicious, and concludes that if there is an error, it must be due to a faulty actor. However, the application's dataflow consists of several hundred actor instances, and it is difficult to manually inspect for a faulty actor.

Suspicious actor debugging. What the analyst needs is to examine the dataflow in general and spot anomalies. We call this a *general health query*. With the

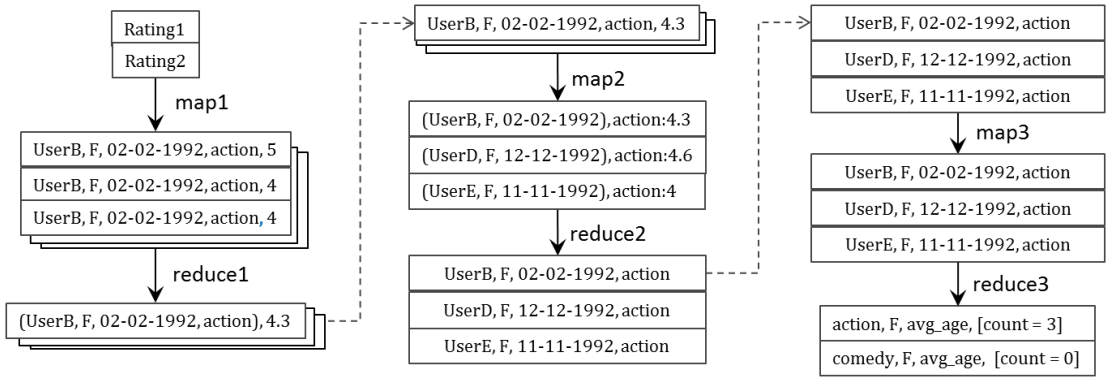


Figure 3.3: This figure shows the propagation of spurious records, inserted by (map1) actor, through the dataflow, eventually manifesting as bad output records.

help of certain observable dataflow characteristics in lineage, e.g. selectivity, the general state of each actor in the dataflow can be classified as apparently normal or suspicious, thus narrowing the subset of the actors the analyst would need to manually inspect. The analyst issues a general health query and identifies a set of actors, $A_{suspect}$, which are flagged as suspicious.

The analyst finds that $A_{suspect}$ consists of a single map1 instance and some reduce1 instances. The analyst then selectively replays the application with the inputs that map1 behaved suspiciously on, and observes that map1 produces too many output records for some ratings from `CrashPage.com`, all with the *action* genre. The analyst examines map1's code and realizes that when an input rating entry consists of multiple genres, map1 produces duplicate records for the first genre, while ignoring other genres in the entry. `CrashPage.com` lists genres in alphabetical order, thus placing *action* in the first place. The missing records for the other genres caused reduce2 to select only *action* as the top genre for users from `CrashPage.com`, thus leading to higher counts for *action* in final output. This error propagation is shown in Figure 3.3.

The three use-cases presented in this section illustrate how basic lineage functions, such as tracing and replay, can be used to efficiently debug different kinds

Table 3.1: Newt capture APIs

Function	Management
register(name,g,α,handle) $\rightarrow$ id	Register actor of type g with the 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$_{src}$.
fail(id,H_{in},H_{out})	Records culprit input when actor encounters exception—H_{out} is optional.
Standard Capture API	
unpaired_capture(id, H_{in}, H_{out}) $\rightarrow$ filter	Create lineage association $(H_{in}, \alpha, H_{out})$. If supporting replay, if `filter` is true, drop output.
Timed Capture API	
addInput(id, H_{in}, T, $reset$) $\rightarrow$ filter	Add H_{in} to current association set A^*, where T is an optional tag. If `reset` is true, empty A^* before adding H_{in}. If supporting replay, if `filter` is true, drop input.
addOutput(id, H_{out}, T, $reset$)	Add lineage association (A^*, p_{id}, H_{out}), where T is an optional tag. If `reset` is true, empty A^* after adding the association.

of errors in a dataflow. The following sections present Newt's capture, trace and replay APIs and interfaces, and describe how Newt uses lineage to enable different kinds of tracing and replay, which serve as the building blocks for debugging with Newt.

3.3 Capture APIs

Newt faces several challenges in capturing lineage for DISC systems. It must capture lineage at multiple granularities of actors and data, capture lineage from black-box actors with reasonable accuracy and accommodate common DISC system processing constructs. This section describes how Newt addresses these challenges and presents the instrumentation APIs that a developer can use to capture lineage in DISC systems.

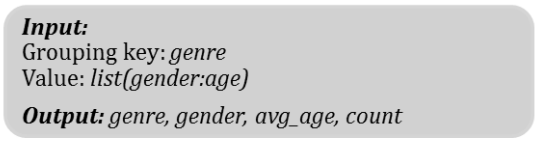
Recall that a DISC system consists of multiple levels of actors and data (Section 2.2.1). For Newt to be able to integrate captured lineage across different granularities, the instrumented DISC system must inform Newt about the different actor granularities within it. Thus, prior to capture, it specifies its actor hierarchies to Newt, to establish containment relationships. This containment specification is

called a $gset$ [46]. A gset contains an actor type for each logical actor in the DISC system. For example, each actor in Figure 2.2a would have a type in the gset. For each actor type, the gset consists of tuples of form $\{g, id, \theta\}$, where g is the actor type, id is its globally unique identifier, and θ is the set of identifiers of its ancestor types in the gset. Thus, θ specifies the containment relationships amongst the actor types and is the template for containment relationships between actor instances. Each actor type can also be associated with an input data type and an output data type. For example, the input and output data type for a MapReduce job is $file$.

Newt provides different capture APIs to leverage common programming constructs used in most DISC systems (Section 2.1.2). Through these APIs, Newt captures associations between inputs and outputs of different kinds of actors, without requiring any knowledge of the actors being instrumented. To minimize the size of the lineage, Newt only captures *hashes* of inputs and outputs. These are cryptographic hashes constructed from the contents of the data being hashed and uniquely identify the hashed data.

3.3.1 Unpaired capture

The most basic capture API, unpaired_capture can be used where both the output and the exact set of inputs used to derive all of it are known and available, i.e. when the accurate lineage of the output is available (Section 2.2.2). This is the simplest instrumentation case and the developer needs to insert a single instrumentation call to unpaired_capture, which associates an input hash H_{in} of an actor to an output hash H_{out} of the same actor, as shown in Table 3.1.



Input [key, {value list}]	Output
action, {F:25, M:21, M:30}	action, F, 25, 1 action, M, 25.5, 2 action, all, 25.3, 3
comedy, {F:20, M:21}	comedy, F, 20, 1 comedy, M, 21, 1 comedy, all, 20.5, 2

Figure 3.4: reduce3 actor from the movie preferences application.

3.3.2 Paired capture

In many cases, it may be difficult to instrument an actor with unpaired_capture, or using it would not capture accurate lineage. For example, in a group-wise actor, such as reduce3 in our running example, the input record consists of a group of input elements, which are iteratively processed to produce outputs. Using unpaired_capture in this case would associate all outputs with all input elements in the grouped input record, when that is clearly not accurate.

Moreover, accuracy also reduces the size of the lineage, since it creates fewer associations. For example, using unpaired_capture in reduce3, shown in detail in Figure 3.4, associates the output record for female users who prefer $action$ with three inputs $\{action, F:25\}$, $\{action, M:21\}$ and $\{action, M:30\}$, when the first association is the only one that is accurate.

Instead, Newt provides the *paired capture* APIs, which enables the developer to introduce separate instrumentation calls for collecting inputs and outputs of an actor. These APIs collect inputs and outputs separately, and Newt builds the actual associations lazily only after the actor terminates[1]. Using unpaired capture for group-wise constructs not only increases the accuracy of the lineage, but also reduces its size. The effect of using paired capture APIs in reduce3 is shown in Figure 3.5. Newt provides two methodologies for associating separately captured inputs and outputs to their correct counterparts, as described next.

[1]Note that while the associations are built lazily, the fine-grain lineage is captured actively.

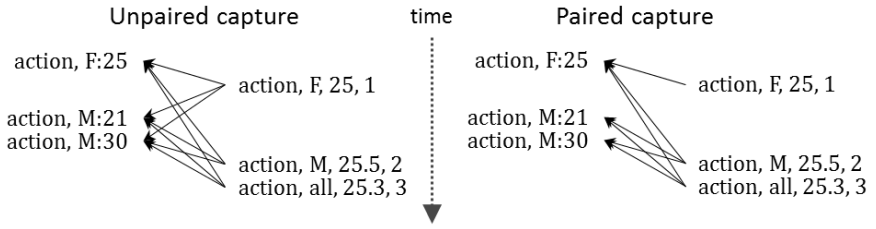


Figure 3.5: This figure shows the effect of paired capture on lineage associations. Capturing lineage from reduce3 using paired capture produces fewer associations while being more accurate than unpaired capture.

Timed capture. Most actors execute on a single machine and many are pipelined [46]. Thus, the temporal order of outputs and inputs, when timestamped locally, can be used to infer lineage, since an output can only appear if its inputs have been processed. The timed capture APIs are a set of paired capture APIs, which leverage this.

These APIs consist of the addInput API, which captures an input record, and the addOutput API, which captures an output record. During capture, Newt locally timestamps each record. Timed capture uses these timestamps to determine the order in which inputs and outputs were generated and associates each output with only the inputs that were generated before it.

However, this could cause outputs to be associated with all inputs generated before them, even those that play no part in their derivation. To avoid this, the developer can optionally configure addInput or addOutput to insert a *reset* into the logical lineage stream. A reset indicates that input or output records that precede it are unrelated to those that follow it. Thus, logically, each call to addInput builds an *association set* A^*. A subsequent call to addOutput with an output record o associates o with each input $i \in A^*$, and introducing a reset empties A^*, preparing it for the next set of inputs. Figure 3.6 shows the effect of reset when using timed capture APIs in reduce3 of our example.

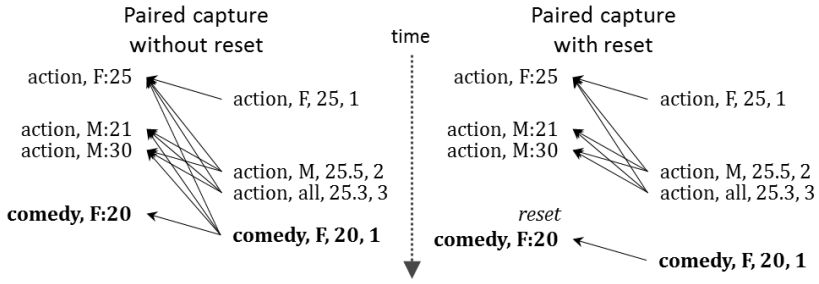


Figure 3.6: $reset$ enables the actor to dissociate inputs generated prior to the reset from outputs generated after it.

Tagged capture. Sometimes, the timed capture APIs may not capture accurate lineage, such as when the actor being instrumented does not pipeline inputs to outputs, but buffers inputs locally before processing them. For example, the Hyracks [21] HashGroup operator reads in all its inputs beforehand, hashes them into a hash table and then processes each hash key from the table to produce an output. As such, the actual order in which inputs are read in has no correlation to the order in which they are processed.

To address such actors, Newt provides tagged capture APIs, an extension of the timed capture APIs, which use *tags* to associate inputs and outputs. The addInput API can take an optional tag T along with the data, which builds an association set A^T. The addOutput API can also take a tag T associated with an output record o. Subsequently, Newt associates each output carrying tag t, with all inputs carrying tag t generated before it, i.e. it associates o with all input records in A^T. Figure 3.7 shows how tagged capture APIs associate inputs and outputs. Timed capture APIs also support reset. However, reset here applies to each tag, i.e. it only empties the association set A^T, where T is tag in the capture call (addInput or addOutput) which inserts the reset into the logical lineage stream, while not affecting association sets for other tags.

Note that although Newt uses tags to create associations during tagged capture, these tags are not introduced into the actual dataflow, but are identifiers, which

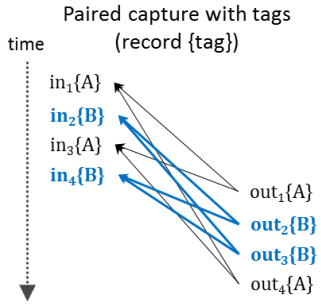


Figure 3.7: Tagged capture associates each tagged output only with inputs carrying the same tag generated before it.

the actor uses while processing buffered inputs.

3.4 Lineage tracing

Newt provides efficient and scalable tracing that is generic to different DISC systems. Once Newt finishes collecting lineage for a dataflow, it builds an association table for each actor. This table contains the actor's lineage. At this point, Newt can perform tracing on this lineage.

A tracing query is a function $F = \mathsf{trace}(d[], T_{root}, dir)$, where $d[]$ is the list of data elements to be traced, T_{root} is the actor instance that produced $d[]$, and dir specifies the trace direction, which can be either backward or forward. trace returns the topologically arranged set of actors (dataflow) that processed the traced data or its containing data instances, their derivations (forward trace), or precursors (backward trace). This result dataflow is called a *tracing dataflow* and is a subset of the original dataflow.

To answer a tracing query, Newt must do two things. First, Newt arranges the actors into a topological order that represents the original dataflow (Section 3.4.1). Next, once the dataflow is successfully reconstructed, Newt issues the trace on the association tables of these actors (Section 3.4.2).

3.4.1 Dataflow reconstruction

Before Newt can serve tracing queries that involve multiple actors in a dataflow, Newt must first build a graph of these actors to represent the dataflow. To do so, Newt links each actor T to its *upstream* and *downstream* actors in the dataflow. An upstream actor of T is one that produced the input of T, while a downstream actor is one that consumes the output of T. Newt reconstructs the dataflow with the help of three types of links.

Explicitly specified links. The simplest link is an explicitly specified link between two actors. When an actor is aware of its *exact* upstream or downstream actor, it can communicate this information to Newt, using the flow_link API, shown in Table 3.1. Newt then uses this information to link these actors during the tracing query. For example, in the MapReduce architecture, each map instance knows the exact record reader instance whose output it consumes.

Logically inferred links. Newt allows developers to attach dataflow archetypes to each logical actor type in the gset (Section 3.3). A dataflow archetype explains how the children types of an actor type arrange themselves in a dataflow. With the help of this information, Newt can infer a link between each actor of a source type and a destination type. For example, in the MapReduce architecture, the map actor type is the source for reduce, and vice-versa. Newt infers this from the dataflow archetypes and duly links map instances with reduce instances.

However, there may be several MapReduce jobs in the dataflow, and linking all map instances with all reduce instances can create false links. To prevent this, such links are restricted to actor instances contained within a common actor instance of a containing (or parent) actor type. Thus, map and reduce instances are only linked to each other if they belong to the same job.

Implicit links through dataset sharing. In DISC systems, sometimes there are implicit links, which are not specified during execution or through gsets. For example, an implicit link exists between an actor that wrote to a file and another actor that read from it. Newt creates a link between two such actors that share a dataset (Section 5.4).

3.4.2 Tracing query

Once Newt completes dataflow reconstruction, it can issue trace on the actors involved. Recall that Newt captures lineage in a dataflow one actor at a time. However, since DISC dataflows consist of multiple actors, a tracing query potentially spans several actors.

To trace lineage across multiple actors, tracing performs *output-input matching* between the actors in the dataflow, i.e. it matches outputs of upstream actors with inputs of downstream actors. For example, consider an actor T, which uses input i to produce output o. To forward trace i, Newt looks up its lineage, which is o, and queries the association table of T's downstream actor for an input matching o. The backward tracing process for o is symmetrical, and Newt queries the association table of upstream actors for an output matching i. The *query results*, i.e. the matching data elements in upstream or downstream actors, are stored in a table called the *tracing table*, which is a subset of the actor's association table. Newt performs this output-input matching recursively, in a breadth-first manner, until there are no more upstream (or downstream) actors remaining.

However, output-input matching can only occur between two actors if Newt can compare their data elements. In our example, it can occur between records of reduce1 and map2, between files of job1 and job2, but not between a file and a record of job1 and map2 respectively. To overcome this and answer multi-granular queries, trace uses containment specifications in the gset (Section 3.3) to identify containment relationships between the query results and data elements of a parent

data type.

3.5 Replaying with lineage

Fine-grain replay is another important application of lineage. Current approaches to fine-grain replay include selective refresh [38], which replays updated inputs to compute output updates. However, other important uses of replay include replay for step-wise debugging and removing bad inputs to reproduce error-free outputs. Newt enables these two types of replay, called selective and exclusive replay, respectively. This section describes how Newt performs replay.

Newt uses tracing as a building block for replay. Users request a replay of a dataflow by submitting a trace along with it. Newt uses the results of the trace, stored in tracing tables, to coordinate the replay. It is important to note that although Newt can accurately replay most black-box actors, non-deterministic actors are beyond its current scope.

3.5.1 Replay initialization

Recall that Newt only records hashes of inputs and outputs to minimize the size of the capture lineage (Section 3.3). Thus, the actual input dataset is necessary to replay an actor. Moreover, not all actors in a dataflow can be restarted in isolation, and Newt cannot use such actors to initiate a replay. For example, individual tasks in a MapReduce job cannot be restarted unless the job is restarted as well, and thus cannot be replayed unless the job is replayed as well.

Thus, to replay a dataflow, Newt needs a set of actors that are *restartable* and have *materialized* inputs. Whether an actor is restartable and whether its input is materialized are characteristics of its actor type and the data type of its input. For example, a MapReduce job is restartable while individual tasks are not. Similarly, files are materialized while records are not. Newt identifies such restartable actors

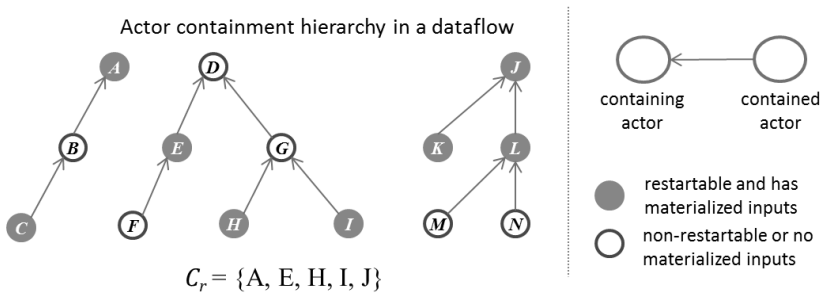


Figure 3.8: This figure shows an actor containment hierarchy with the highest-level restartable actors with materialized inputs selected as candidates to restart. In a subtree rooted at a candidate actor, Newt ignores any other potential candidates and only selects the highest-level candidate, which is the root itself.

with materialized inputs, or *replay candidates*, in the following manner.

Newt uses a top-down search on the traced dataflow to identify replay candidates. The search begins with the most top-level actors and recursively searches through their child actors. It maintains a set of replay candidates, C_r, initially empty, into which it adds any actor that is restartable and has materialized inputs. If an actor qualifies, Newt does not look any further into its child actors. However, if an actor does not qualify, it is discarded and its child actors are searched. The search stops when Newt reaches the lowest-level actors and there are no more actors to search.

For example, Figure 3.8 shows a containment hierarchy of actors, along with the set C_r of candidates that Newt must restart at the end of the search. If C_r remains empty after the search, the replay cannot proceed. Otherwise, Newt restarts each actor $c \in C_r$ on a subset of its materialized inputs as dictated by the results of the trace and the type of replay requested (selective or exclusive). Each restarted actor, in turn, restarts its child actors recursively.

3.5.2 Replay filters

Newt performs two types of replay - selective replay and exclusive replay. During selective replay, Newt must prune the inputs that each actor in the dataflow

is replayed on. To do so, Newt installs a *filter* on each actor being replayed. The filter consists of the elements in the tracing table of the actor's counterpart in the original dataflow. Replay at each actor is facilitated by the capture instrumentation in the actor. During replay, the unpaired_capture, addInput and addOutput APIs return a boolean value, `filter`, for each input (or output) of the actor. `filter` indicates whether the input (or output) can be allowed to continue into the actor (or out of it and to downstream actors), or discarded. Inputs are considered when the trace used for the replay is a backward trace, while outputs are considered when it is a forward trace. If `filter` is true, the data element is allowed to continue, otherwise, it is discarded.

For selective replay, `filter` is true when the hash of the input (or output) matches an entry in the installed filter. Filtering ensures that Newt only allows inputs, which existed in the original execution of the dataflow, to continue, thus ensuring accurate replay. With filtering, Newt can perform selective replay for non-monotonic actors as well. As shown by Ikeda et al. [38], filtering is necessary for accurate replay when there are non-monotonic actors in the dataflow.

Newt can also perform exclusive replay, which removes specific (typically bad) inputs during replay. Exclusive replay only filters inputs at the actor from which the bad inputs must be removed. The rest of the downstream dataflow is executed without filtering. For example, suppose the actor at which inputs must be removed is α, the set of inputs to be replayed is I_{replay}, and the set of inputs to be removed is I_{remove}. If α is not restartable or does not have materialized inputs, Newt must find the topologically closest replay candidate containing α, say β, and selectively replay the dataflow from β to α on the lineage of I_{replay}. Once replay reaches α, Newt removes I_{remove} from its inputs and allows the remaining dataflow to execute without filtering. Note that Newt must still restart the replay candidates in the remaining dataflow to execute it.

Exclusive replay does not ensure that the outputs of the replay will be a

subset of the original output of the dataflow, especially if non-monotonic actors are present in the part of the dataflow downstream of α. However, the goal of exclusive replay is to compute error-free outputs or simulate what-if scenarios. As such, the outputs are usually expected to be different.

The use-cases in this chapter show how five different lineage functions, i.e. tracing, selective replay, exclusive replay, crash culprit determination and suspicious actor debugging can be used to debug DISC dataflows. While this chapter describes the set of APIs and interfaces Newt provides to facilitate efficient capture, tracing and replay, a detailed description of the other lineage functions (crash culprit determination and suspicious actor debugging) is presented in Chapter 4.

Chapter 4

Debugging

The previous chapter illustrates how lineage can be used for debugging DISC dataflows and describes Newt's capture, tracing and replay methodologies. Newt can use tracing and replay to assist with debugging several different types of errors (Section 3.2). While lineage tracing and replay can successfully debug bad outputs produced due to bad inputs, they are not sufficient to identify faulty actors and crash culprits in a dataflow. Discovering bad outputs in a large dataset and pinpointing the responsible faulty actors can be difficult for a complex analytics with thousands of actors.

Current approaches to debugging faulty actors include recursively performing coarse-grain replay on actors in the dataflow [65], which can be expensive in resources for long dataflows. Another approach is to manually inspect lineage logs to find anomalies [30, 54], which can be tedious and time-consuming across several stages of a dataflow. Furthermore, these approaches work only when the user can discover bad outputs. To debug analytics without known bad outputs, users need to analyze the dataflow for suspicious behavior in general. Inspector Gadget [48] uses predicate-based monitoring to flag violations in a dataflow. However, often, a user may not know the expected normal behavior and cannot specify predicates.

Thus, there is a need for an inexpensive automated debugging technique,

which incurs low time overheads and minimizes the amount of work a user must do to identify faulty actors in the dataflow. To address these debugging needs, Newt provides a set of debugging interfaces, which assist a user in inspecting a dataflow for anomalies, as well as identifying faulty actors given a bad output.

Newt can also assist in debugging crashes in a DISC dataflow. While one approach involves repeatedly replaying the failed actor on increasingly smaller subsets of its input to reproduce the crash and identify the inputs responsible, it can be expensive. Instead, Newt isolates the crash culprits at runtime, thus avoiding replay overheads. This chapter describes how Newt performs crash culprit determination and suspicious actor debugging.

4.1 Crash culprit determination

Newt performs crash culprit determination by flagging the last inputs seen by the actor as culprits of the crash [48]. To do so, Newt provides the fail API, as shown in Table 3.1. To enable crash culprit determination, a developer integrates this API into each actor's code to capture last seen inputs. For example, in a MapReduce job, fail can be inserted into the map actor's exception handler. During a crash, fail records the crash culprit inputs and Newt stores them as a separate lineage table. Subsequently, users can query Newt for these inputs and trace them through the dataflow to identify their sources.

4.2 Suspicious actor debugging

This section describes a debugging methodology for retrospectively analyzing lineage to identify faulty actors in a multi-stage dataflow. We believe that sudden changes in an actor's behavior, such as its average selectivity, processing rate or output size, is characteristic of an anomaly. Lineage can reflect such changes in

actor behavior over time and across different actor instances. Thus, mining lineage to identify such changes can be useful in debugging faulty actors in a dataflow.

To do so, Newt provides an anomaly detection interface that returns a list of actors in the dataflow, ranked by the likelihood of them being faulty, thus narrowing the set of actors a user must manually examine. Newt uses selectivity values to characterize faulty behavior, and applies outlier detection techniques to selectivity values in the captured lineage to identify suspicious actors. This section presents the anomaly detection interface and describes how Newt uses selectivity to rank actors.

4.2.1 Anomaly detection interface

Often, in complex analytics, a user may not know if there is a bad output, but wish to assess the general health of the dataflow. At other times, the user may know a particular bad output and wish to identify the actor responsible for it. To address these different debugging scenarios, Newt provides a flexible anomaly detection interface. An anomaly detection query is expressed as the function $F =$debug$(T_{root}, O_{bad}, O_{good}[])$, where T_{root} is an actor instance in the dataflow to be examined, O_{bad} is an optional bad output produced by T_{root}, and $O_{good}[]$ is an optional list of a sample of good outputs produced by it.

The most basic debug query, enables the user to test the general health of the dataflow. In this query, the user only specifies T_{root}. To process this query, Newt first identifies a subset of the dataflow graph consisting of T_{root}, its parent and child actors, and all upstream and downstream actors, which derived or process their input precursors and output derivations respectively. Next, it examines this graph, and returns its state as a ranked list of all actors within it. Note that Newt only identifies the dataflow graph, but does not trace through it.

The user can also provide a known bad output to improve debugging accuracy. When the user to specifies T_{root} and a bad output O_{bad}. Newt performs a backward trace on O_{bad} and identifies a subset of actors to rank in the dataflow

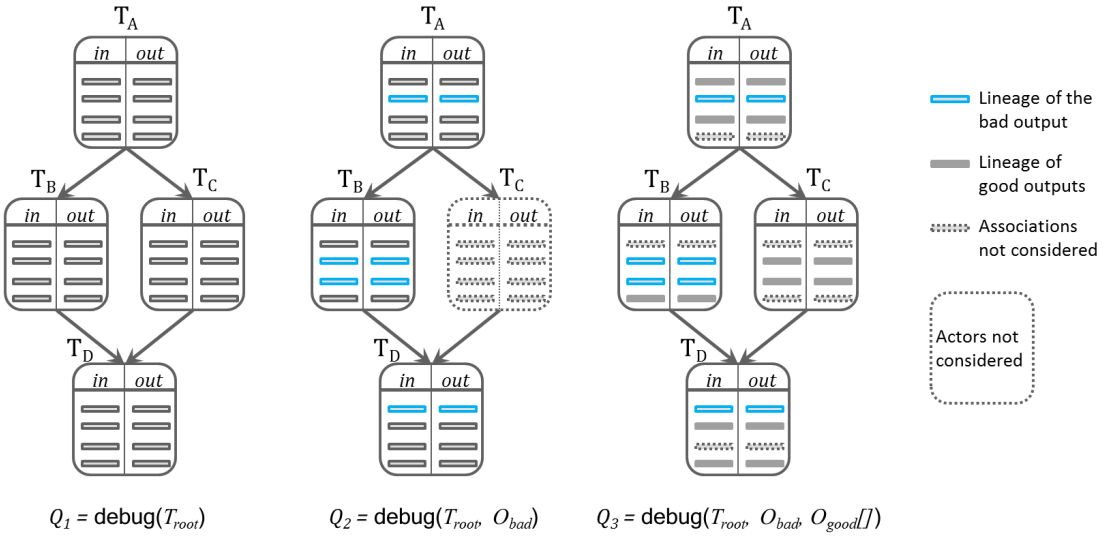


Figure 4.1: This figure shows the subset of actors and associations considered by the anomaly detection query when the user supplies different information. Q_1 considers all associations of an actor and ranks all actors in the dataflow. Q_2 considers all associations of an actor and ranks only the actors, which process the lineage of the bad output. Q_3 considers only the lineage of the good and bad outputs and ranks all actors, which process their lineage.

graph, consisting of only the actors that processed O_{bad} or its lineage.

Finally, the user can also supply a sample of good outputs of the dataflow, $O_{good}[]$, along with T_{root} and O_{bad}. Newt then considers only a subset of associations (and consequently, a subset of selectivity values) for each actor, only consisting of the lineage of O_{bad} and $O_{good}[]$.

As the user specifies more information, in the form of a bad and a sample of good outputs, the response latency of the query improves due to the smaller subset of actors and associations within actors that Newt considers. This also helps in eliminating noise, thus improving debugging accuracy. Figure 4.1 shows the subsets of the dataflow and actor associations each debug query considers.

4.2.2 Characterizing faulty behavior

To identify and rank faulty actors, Newt needs a methodology for characterizing faulty behavior. Our hypothesis is that within an execution, the range of selectivity for given actor and input data is homogeneous. Thus, Newt uses differences in observed lineage selectivity to identify anomalous actors, and characterizes actor behavior in a two-step process.

Selectivity across time for an actor instance. To identify an anomalous actor instance, Newt first examines association tables of each actor in the dataflow to check for anomalous selectivity values, and characterizes an actor displaying significantly different selectivity values for a small subset of its inputs as faulty.

Selectivity across actor instances. Sometimes, a particular actor instance may exhibit anomalous selectivity for all or most of its inputs, in which case the previous check will not flag it as faulty. Thus, Newt next compares the average selectivity of each actor instance to other instances of the same actor type and characterizes instances with markedly different average selectivity as faulty.

4.2.3 Ranking faulty actors within a dataflow

To narrow the subset of actors that a user must manually inspect, Newt ranks the actors in order of the likelihood of them being faulty. Newt ranks actors based on the number of faulty characteristics they exhibit. Newt maintains a score (initially zero) for each actor in the dataflow, and increments this score when an actor appears anomalous. For example, Newt increments the score for an actor instance α, which exhibits significantly different selectivity values for some of its inputs, by 1. Subsequently, if α also displays different average selectivity than other actor instances of its actor type, Newt increments its score by 1. Thus, a higher score indicates more anomalous behavior.

```
1.  S = {s1,s2,…,sn}                                  //set of selectivity values
2.  O = {}                                            //empty set of outliers

3.  begin detect_outliers(S)
    1.   μ = mean(S),
    2.   δ = standard_deviation(S)
    3.   dmin = (min(S) – μ)/δ                        //lowest distance value
    4.   dmax = (max(S) – μ)/δ                        //highest distance value
    5.   pnum = ceiling(dmax – dmin)                  //number of partitions
    6.   p1,p2,…,ppnum = {}                           //begin with empty partitions

    7.   foreach si in s; do
           di = (si – μ)/ δ                           //distance from μ in units of δ
           k = ceiling(di – dmin)                     //determine partition number
           add si to pk                               //partition selectivity values
    8.   done

    9.   α = mean(count(pk),…,count(ppnum))           //average partition size
    10.  foreach pk in p1,p2,…,ppnum; do
           if count(pk) < α/2, then add all si in pk to O
    11.  done
4.  end detect_outliers
```

Figure 4.2: Outlier detection algorithm for detecting multiple anomalous selectivity values.

To identify anomalous selectivity values, Newt uses a modified version of Grubbs' outlier detection technique [35]. While Grubbs' technique finds a single outlier in a given set of data points, we modify it to enable Newt to identify multiple outliers. For each actor in the dataflow, Newt first computes the mean μ and standard deviation δ of selectivity for each unique input. Next, it calculates the distance of each selectivity value from μ in units of δ. Newt partitions these "distance" values into unit ranges, and computes the average number of data points per partition. Finally, it flags any partitions with less than half the average number of data points as an outlier partition, and determines all corresponding selectivity values as anomalous. Figure 4.2 shows the outlier detection algorithm that Newt uses. Newt increments an actor's score if it contains anomalous selectivity values.

When more than two instances of a particular actor type exist in the dataflow, Newt computes average selectivity value for each instance and applies the same

outlier detection technique to these values. Subsequently, it increments the score of any actor instance that falls into an outlier partition.

Chapter 5

Architecture

One of the primary challenges in capturing lineage for DISC systems is scaling to their high throughputs and multi-stage dataflows. Newt addresses this through a scalable peer-based architecture for lineage capture and storage, distributed query, replay and debug support. The Newt system consists of a set of peers and a logically centralized controller, each of which contains a SQL database. The controller manages the distribution of incoming lineage load across the cluster, tracks the location of association tables in the cluster and coordinates distributed tracing queries across these tables. Newt handles failures in its cluster by replicating lineage logs and providing fail-over capability in peers.

Figure 5.1 shows the basic Newt architecture. Newt provides a client-side library, which a developer uses to instrument a DISC system. The client library provides APIs to capture lineage and manage the lifecycle of an actor. When an instrumented actor executes, the client issues its lineage to a Newt peer, which logs incoming lineage to local disk and replicates it to another peer in the cluster. Once the actor terminates, the peer imports the actor's lineage log to a SQL association table. The controller tracks the distribution of logs and tables on different peers, and balances CPU load and disk utilization across the cluster. Once tables are built and indexed, users can trace the captured lineage and perform replay and debug.

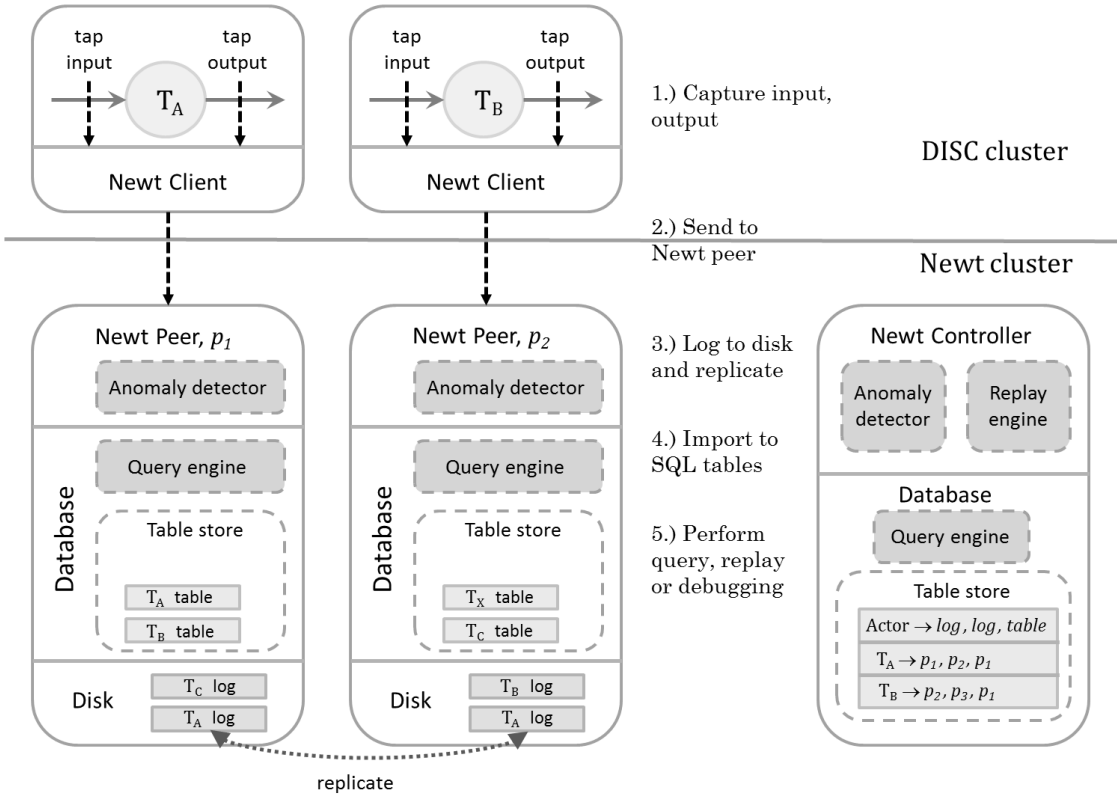


Figure 5.1: The Newt system consists of a set of peers and a logically centralized controller. Clients send lineage to a Newt peer, which logs them to disk and replicates them to another peer. Once capture is complete, logs are imported to SQL tables. The controller tracks the distribution of load and tables across the cluster.

The Newt query engine federates the tracing query across the cluster, and the replay engine on the controller uses the results of the query to coordinate dataflow replay. Finally, each peer also contains an anomaly detector for identifying faults in local actor instances. The controller gathers anomaly results from each peer and creates a ranked list of faulty actors. This chapter describes different components of the Newt architecture.

5.1 Client-side library

Newt provides a client-side library which has two primary functions. It exposes an instrumentation API to the actor for managing the actor lifecycle and capturing lineage. As the instrumented actor generates inputs and outputs, the client internally hashes and timestamps them, and buffers them in local memory before periodically issuing them to a Newt peer. This section describes these two functions of the client.

5.1.1 Actor lifecycle management

Prior to capture, the DISC system specifies its actor types and containment hierarchies to the Newt controller. When an instrumented actor executes, it first registers itself with the Newt controller using the register API in the client library (Table 3.1). To do so, the actor instance specifies its actor type g, provides the *actor IDs* of its containing actor instances and specifies a text $handle$, which uniquely identifies this instance among all instances of the same actor type. Newt uses this $handle$ to identify this actor instance during replay (Section 6.4).

An actor ID is a globally unique identifier for each actor instance, which the Newt controller assigns to the instance once it registers. These IDs also serve to establish instance containment relationships, since child instances provide their parent instances' IDs during registration. In response to an actor's register request, Newt assigns it an actor ID and a Newt peer. This assigned peer logs all lineage associations from the actor to its local disk. Once the actor finishes processing, it notifies this to Newt through the commit API in the client library (Table 3.1). commit tells Newt that the actor's lineage log is complete, and Newt can then import it to an association table. It is important that an actor, which successfully finishes processing, calls commit, since it helps Newt distinguish successful actors from failed actors (Section 5.3).

5.1.2 Lineage data types

Recall that Newt only stores hashes of inputs and outputs of an actor. This not only minimizes the size of the lineage captured, it also enables Newt to handle lineage associations in a uniform manner during tracing, without needing custom handlers for different DISC data types, such as integers, text and key-value pairs. The client-side library internally hashes all DISC data types into one of two *lineage data types* that Newt stores and uses.

The first lineage data type that Newt uses is a 128-bit byte array. Most common DISC system data types, such as key-value pairs, integers and text, can be hashed to a 128-bit byte array. For example, the client cryptographically hashes input and output key-value pairs from map instances to this data type.

Newt also uses a *locatable* type, which incorporate notions of data containment. A data instance of a locatable type contains information that can be used to identify its parent data instance, as well as sub-parts of it. For example, a file location with byte offset and length is a locatable type instance. The file path identifies its parent directory, and the byte range within the file identifies smaller byte ranges within it. A locatable type for files is specified by the triplet $L = \{filepath, offset, length\}$. In a MapReduce job, the client hashes record reader inputs and record writer outputs to locatable type.

5.2 Storage subsystem

Each machine in the Newt cluster contains a SQL database. When an actor finishes processing, the Newt peer assigned to it imports its lineage log to an association table in its database. When the actor instance uses unpaired_capture to collect lineage, the peer imports its lineage into a single association table, in which each row contains a timestamped lineage association. However, when the actor uses paired capture APIs to collect inputs and outputs separately, the association table

at the peer is split into two - one containing timestamped (and optionally tagged) inputs, and the other containing timestamped outputs (also optionally tagged). All tables are indexed on all columns and indexes are stored along with the tables themselves. Newt peers also store tracing tables for each association table, when they are created during a tracing query.

The Newt controller maintains tables in its database to manage peers and actors, and to track the distribution of lineage tables across the cluster. To do so, it maintains three important tables. It stores the gset specification for an instrumented DISC system in the actorGset table. actorGset contains all actor types, their containing actor types, their input and output data types, whether they are restartable, and the links between the actor types. The controller also maintains a table actorInstances, which has one entry per registered actor instance. This table maintains the instance's actor ID, its parent's ID, the IDs of its upstream and downstream actors (if specified by the actor using flow_link), the peer assigned to the actor and the actor's state. The state of the actor tracks the lifecycle of its lineage. An actor's state can be `Incomplete`, `Committed`, `Populating`, `Complete` or `Failed`, respectively indicating that the actor is still generating lineage, has finished and called commit, its lineage is being imported into an association table, its lineage tables are complete, or the actor indicated runtime failure using fail (Section 4.1). Finally, the controller maintains a table, dataInstances, which records each locatable data instance along with all actor instances that read from it or wrote to it. Newt uses this information to infer implicit links between actor instances (Section 3.4.1).

5.3 Load balancing and fault tolerance

To scale to large volumes of data and large numbers of actors, Newt distributes CPU and storage load across the cluster. Newt assigns a peer to an actor

using weighted moving averages of CPU utilizations on all peers to find the peer with the lowest load. Once lineage is complete and can be imported into a table, Newt finds the peer with lowest disk utilization and creates the table on it.

It is also important for Newt to be fault tolerant, to avoid the need to re-execute dataflows for capturing complete lineage. To address this, the Newt controller also assigns a secondary peer to each actor. Each client sends lineage synchronously to its primary peer. The primary peer, in turn, logs this lineage locally and replicates it to the secondary peer before acknowledging the client. The client waits for the acknowledgment before discarding its local copy of the lineage. If the primary peer fails during the actor's lifetime, the secondary peer takes over as primary and the controller assigns a new secondary to the new primary peer. The new secondary then receives a copy of the lineage captured thus far from the new primary.

Newt also gracefully handles failures in the DISC system. To do so, the controller tracks the lifecycle of each actor (through register and commit calls), and uses instance containment relationships to identify failed actors. When an actor registers with the Newt controller, its state is marked `Incomplete` in the actorInstances table. When the actor finishes processing, it calls commit, and subsequently, the controller marks the actor's state as `Committed`. However, if the actor fails before it can call commit, its state on the controller remains `Incomplete`.

Newt expects all child actor instances to finish processing before their parent actor instances, which is justified by the hierarchical relationships between parent and child actors. Thus, when a parent actor, α, commits, the controller initiates a *cleanup phase*. During the cleanup phase, the controller sweeps through all of α's child actors, and discards all lineage from any child still in `Incomplete` state (and all its children, recursively). This successfully prevents Newt from storing duplicate copies of lineage across task restarts in DISC systems. Note that if a

failed actor instance calls fail to explicitly indicate failure (Section 4.1), its state is set to `Failed`, which preserves its lineage during cleanup phase.

5.4 Distributed query engine

Newt consists of a distributed query engine, which serves tracing queries. Recall that before Newt can perform a trace, it reconstructs the dataflow, creating links between two actors writing to and reading from a common dataset. To do so, Newt uses locatable data type to identify the dataset, and builds an association table called a *ghost table*, which associates outputs of the writer actor with overlapping inputs of the reader actor using locatable information. A ghost table is treated as any other association table, and Newt also creates a *ghost actor* instance for this table in actorInstances.

Once the dataflow is reconstructed, the query engine federates the tracing query across the cluster. The query performs output-input matching between association tables using a sequence of relational joins on these tables. For actors that collect lineage through paired capture, the query also creates associations from their split association tables (Section 5.2), using timestamps or tags to decide the associations.

To optimize the query processing time, Newt supports three different policies for distributing tables across the peers - random, vertical co-location and horizontal co-location. The random placement policy randomly allocates tables across the cluster to any peer. The other two placement policies co-locate actor instance tables based on the dataflow. In vertical co-location, Newt assigns the same peer to two tables whose actor instances are linked to each other (directly, logically or implicitly). This leverages parallel processing capacity of the cluster, since parallel portions of the dataflow can be processed simultaneously during the query. In horizontal co-location, Newt assigns all tables of the same actor type to the same

peer. This minimizes network traffic between peers containing actor instances from different stages of the dataflow, but the query can take longer to process than during vertical co-location.

5.5 Replay engine

The Newt controller also consists of a replay engine, which coordinates dataflow replay. When a user requests a replay, the replay engine internally issues a tracing query for the outputs (or inputs) to be replayed and replays the actors in the dataflow on a subset of their original inputs. To replay the dataflow, users can either manually re-execute the dataflow or have Newt restart actors. For the latter, restartable actors implement a restart$(name, conf)$ RPC method. The $name$ is the same $name$ that the original instance provides when calling register, and $conf$ is actor configuration data, which Newt specifies when restarting it.

5.6 Anomaly detector

Each machine in the Newt cluster also contains an anomaly detector. When a user submits a debug request, Newt first identifies the subset of the dataflow and associations it needs to consider for ranking (Section 4.2.1). If necessary, Newt issues a backward trace to create tracing tables containing the relevant subset of associations. Note that while a tracing query is required whenever the user specifies any outputs, tracing tables are only used when the user supplies a sample of good outputs; the other forms of debug consider all associations of an actor instance.

Recall that Newt uses a two-step process to identify faulty actor instances in a dataflow (Section 4.2.2). In the first step, the anomaly detector on each Newt peer queries relevant local association tables (or tracing tables), for their selectivity values, performs outlier detection on these values and increments the score of faulty

actor instances. The Newt peer sends these scores and average selectivities for each of its relevant local actor instances to the controller.

In the next step, the Newt controller collects scores and average selectivities from its peers. The anomaly detector on the controller then performs outlier detection on average selectivities across instances of the same actor type, and increments the score of faulty actor instances. Finally, it creates a ranked list of actor instances and returns this list to the user.

Chapter 6

Implementation

This chapter describes key aspects of the implementation of Newt, including optimizations for better performance and lessons learned. Our Newt prototype is written in Java, and consists of approximately 3.9k lines of code for the controller and 600 lines for the client. Newt uses MySQL 5.5 as the relational database. Communication within the Newt cluster and with the clients uses RPC. Newt uses Hessian 2.0 RPC package, which provides a dynamically-typed binary RPC protocol. Section 6.1 describes the hash function that the Newt client uses. Section 6.2 describes pitfalls with importing binary data into MySQL tables, and how Newt avoids them. Section 6.3 describes optimizations for faster index creation and query performance. Section 6.4 describes how the Newt replay engine coordinates replay. Finally, Section 6.5 describes optimizations for improving the time to send lineage to Newt.

6.1 Hash algorithm

Newt hashes non-locatable type DISC data to 128-bit byte arrays to minimize lineage size and avoid custom handlers for data types during queries. The hash function is implemented using hash algorithms provided by standard Java libraries.

We found that MD5 is faster than all SHA variants and produces a smaller hash than most SHA variants. Consequently, we use MD5 for hashing non-locatable data.

6.2 Storing lineage with MySQL

Newt stores lineage in MySQL [7] tables and submits queries to it using the JDBC driver for MySQL. When batch importing binary data from logs into a MySQL table with a binary-data schema, we observed that MySQL uses character streams to read these logs and strips trailing spaces from the data. While this is not a problem for text, cryptographic hashes can contain trailing whitespaces and the backslash character '\' as bytes, which MySQL would discard, or treat as escape characters in a character stream, respectively. To circumvent this, the client escapes all backslashes in the byte array and replaces the last trailing whitespace with an underscore.

6.3 Index optimizations

Queries on tables with a large number of rows are faster when using indexes instead of sequential scans of the entire table. Thus, Newt builds indexes on each column of each association table. The time to build indexes and the size of the completed indexes depend on the type of data being indexed and its length [7]. Specifically, integers and byte arrays (which are treated as integer arrays by MySQL), take significantly less time to index than text of same length. The size of the index increases proportionally with the length of the data being indexed. Since Newt uses 128-bit byte arrays and locatables, care is necessary that these data types do not slow down index creation.

While 128-bit byte arrays do not pose a problem, locatables can be arbitrarily long and are classified as text, which can slow down index creation and yield a large

index size. Therefore, Newt configures MySQL to only create *prefix indexes* on the locatable data types, i.e. indexes on a fixed-length prefix of the locatable values. Prefix indexing increases the chance of collision since prefixes of different entries may be identical. Therefore, care is necessary that locatable data types are formatted to place frequently varying fields near the beginning of their text representations. For example, the text representation of a file locatable begins with the file offset and length, followed by the file path, which may be identical for several records of an actor.

6.4 Replay coordination

Recall that Newt uses filtering to ensure accurate replay (Section 3.5.2). Newt does so by installing a tracing table for each actor instance in the original dataflow on the corresponding instance in the replayed dataflow.

However, for this to work, Newt must be able to match each actor instance in the replayed dataflow with an instance from the original dataflow. This can be challenging when there are multiple actor instances of the same actor type executing in parallel, for example, map and reduce instances. To address this, Newt uses the $handle$ provided by the original actor instance during registration (Section 5.1.1), which remains the same across different invocations of the instance and can be used to match replayed and original actor instances. For example, a reduce instance in Hadoop MapReduce job can be identified by its partition number. Note that this is an optimization; if Newt is unable to uniquely match a replayed instance with an original instance, it logically unions the tracing tables of all instances of the actor type to create a filter for the replayed instance.

6.5 Communication

An essential component of Newt is its communication module, which is used for both communicating with the client and within the Newt cluster. We use Hessian 2.0 RPC package [5] for all communication. Hessian is a binary web service protocol with its own serialization library. During evaluations, we observed a warm-up phenomenon during the first few RPC requests, which incur larger overheads than the stable-state time to send lineage (average time to send lineage observed over a large number of requests). Actors that produce little lineage, and thus, require only a few requests to send it all to Newt, incurred higher overheads due to this.

To avoid this pitfall, the Newt client caches Hessian connection objects for each peer in the Newt cluster, and reuses them across all actors on the client machine. This prevents each actor from needing to warm-up its connection object and significantly reduced time overheads for an experimental dataflow, which consisted of a large number of actors, each of which produced little lineage.

Chapter 7

Evaluation

This chapter describes the evaluation of our Newt prototype. We evaluate Newt in several contexts. We first describe how Newt can be used to instrument two DISC systems, Apache Hadoop [1] and Hyracks [21], for fine-grain lineage capture and replay. Next, we evaluate Newt's ability to scale with increasing lineage generation rates. Next, we measure time and space overheads incurred when capturing lineage with Newt using several different workloads, including a Contrail [53] genome assembly workload. Next, we evaluate the tracing query performance with different lineage table placement policies, and study accuracy of the lineage captured by Newt. We also evaluate Newt's replay performance and accuracy. Finally, we evaluate Newt's debugging accuracy when used to identify faulty actors in a dataflow.

Unless noted otherwise, all experiments use a 15-node cluster of Dual Intel Xeon 2.4GHz machines with 4GB of RAM, a single SCSI disk, and connected by gigabit Ethernet. Newt employs seven nodes: six Newt peers and the Newt controller. The other eight nodes run our instrumented Hadoop environment. All experiments use vertical co-location table placement (Section 5.4) as it outperforms other placement policies for all tested tracing queries.

7.1 Instrumenting DISC systems

This section describes how we instrument Hadoop and Hyracks using the Newt client library. The primary goal of instrumentation is to capture accurate fine-grain lineage from dataflows, while being transparent to user-defined functions, such as, map and reduce. Unlike other approaches, which must propagate lineage through the dataflow [38], Newt's instrumentation requires developers to only insert capture calls at the input and output boundaries of actors. Thus, Newt requires less intrusive modifications in the DISC system.

We use Newt to instrument version 0.21.0 of Hadoop. Instrumentation adds 53 lines of code to the job controller, 9 lines to map, and 11 lines to reduce. For Hyracks, we instrument version 0.1.8, adding 60 lines to the job controller, 10 lines for the FileWrite operator, 15 lines for FileScan, and 20 lines for the HashGroup operator.

7.1.1 Hadoop instrumentation

Hadoop is a framework for running applications on a large cluster of commodity hardware, which implements the MapReduce paradigm [27], described in Section 2.1.3. We instrument the job controller, the record reader, map, reduce and record writer operators to capture lineage and enable replay.

Job controller. We instrument the job controller to first specify the MapReduce gsets (shown in Figure 2.2) to Newt. Next, the job controller uses the register API to register each MapReduce job submitted to it as an instance of *MRJob* type, and captures input and output files of the job using unpaired_capture. The job controller also registers the job's record reader, map, reduce and record writer instances with Newt. When the job successfully completes, the job controller calls commit on all these instances. Finally, the job controller implements the restartable

API to enable Newt to restart MapReduce jobs during replay.

Record Reader. record reader reads inputs from a materialized source, and produces output records as key-value pairs. We instrument the record reader with unpaired_capture to collect its inputs and outputs, which hash to locatables and 128-bit bytes arrays, respectively.

Map. map instances consume key-value pairs generated by record reader instances, and produce key-value pairs. We instrument map with unpaired_capture to collect its inputs and outputs, which hash to 128-bit byte arrays. Each map instance in Hadoop is directly linked to exactly one record reader instance and vice-versa. The map instance uses flow_link to specify its upstream record reader instance to Newt.

Reduce. reduce instances read inputs as a key and a list of values (key-value-list), and produce key-value pair outputs. reduce uses an iterator to read and process values in the list one at a time. Thus, each output record may depend on any number of input values already seen for the current input key being processed. We instrument reduce with the timed capture APIs to timestamp and collect inputs and outputs separately, and build the lineage using timestamps after the actor terminates. We also configure the addInput API to issue a reset each time a new input key is seen, to prepare for the next set of input values (Section 3.3.2). reduce instances can receive inputs from any of the upstream map instances, hence, they do not specify any instance as their direct upstream actor. Newt uses dataflow archetypes to link all reduce instances of a job to all its map instances.

Record Writer. The record writer is symmetrical to the record reader, except that it reads key-value pairs and writes to a file. We instrument the record writer using unpaired_capture, hashing its inputs and outputs to 128-bit byte arrays and

locatables, respectively. Each record writer instance is directly linked to a reduce instance, and uses flow_link to specify its upstream reduce instance to Newt.

Replay filtering. The capture instrumentation also enables replay. record reader, map, and record writer instances check the value of `filter` returned by the capture APIs before allowing outputs to propagate to downstream actors. However, reduce instances check `filter` during both input and output capture, before accepting inputs or allowing outputs to propagate.

7.1.2 Hyracks

Hyracks provides a flexible platform for executing large-scale analytics. Unlike Hadoop, in which programs are represented as sequences of maps and reduces, Hyracks programs are represented as arbitrary DAGs of operators. We instrument three operators in Hyracks - FileScan, FileWrite and HashGroup, along with the Hyracks job controller.

Job controller. When a user submits a job to Hyracks, the job controller builds a DAG of operators and executes them. However, the operators (or actors) in a Hyracks DAG can be connected in an arbitrary way. As such, unlike in Hadoop, it is difficult to specify fixed dataflow archetypes in Hyracks. Instead, the Hyracks job controller uses flow_link to directly specify the links between different actors to Newt.

Instrumenting operators. Operators in Hyracks read input tuples, and apply arbitrary functions to generate output tuples. We have instrumented three Hyracks operators found in several common dataflows - the FileScan operator that reads input files and emits words, the FileWrite operator that writes tuples to a file, and the HashGroup operator that implements hash-based grouping and can apply any

user-defined aggregation function per group. FileScan and FileWrite are similar to record reader and record writer, respectively. Here, we describe instrumentation for the HashGroup operator.

Before the HashGroup operator begins processing, it reads all input tuples and hashes them into a hashtable based on a grouping attribute in the tuples. For each input tuple read, it updates the aggregate (e.g. a count) in the corresponding entry in the hashtable. After reading all input tuples, it emits each entry in the table.

While similar to the reduce operator in Hadoop, HashGroup receives unordered inputs, and must read all input tuples before emitting any output. Using the timed capture APIs, as in the Hadoop reduce, would result in associating each output tuple to every input tuple, generating inaccurate lineage.

Instead, we instrument HashGroup with tagged capture APIs, using the hash of the grouping key as the tag for each input and output tuple. Thus, for each input tuple read, HashGroup calls addInput($tuple\_in, hashkey$), and for each emitted output, HashGroup calls addOutput($tuple\_out, hashkey$). We also configure addOutput to reset the association set for the output tag after each output is emitted. This enables Newt to capture accurate lineage for the HashGroup operator.

7.2 Capture scaling

By default, the Newt clients are configured to batch five hundred associations in each lineage request to a Newt peer, and employ double-buffering to enable continuous lineage collection and logging. The default replication factor in the Newt cluster is 2.

Figure 7.1 shows the aggregate performance of the Newt cluster. Here we measure throughput in terms of total volume of lineage recorded as we increase the number of actors by starting dummy Hadoop jobs. The Newt controller is assigning

dummy actors to Newt peers in a round-robin manner. We observe the average rate of logging lineage at a Newt peer to be 67.33 MB/s without replication and 60.56 MB/s with replication. The cluster tops out at a combined 403.98 MB/s without replication and 363.36 MB/s with replication. Thus a single actor can send approximately 1.62 million associations a second to a single peer (without replication), until the local disk throughput becomes the bottleneck.

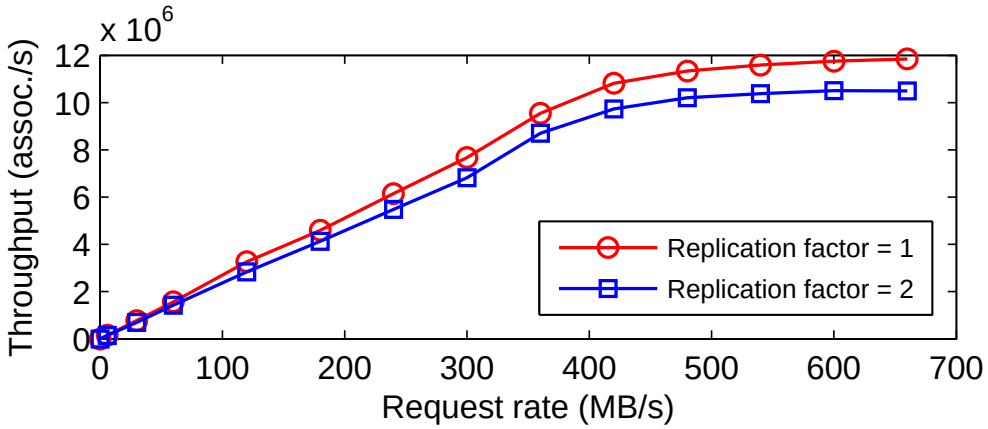


Figure 7.1: Combined throughput of our 7-node (6 peer) Newt cluster in terms of rate of lineage logged to disk. The aggregate throughput reaches 9.7 million associations per second.

Next, we evaluate our load balancing scheme for Newt and DISC cluster layouts. The Newt cluster can be physically separate from the DISC cluster, or can run co-located in the same physical cluster. Balancing the load from logging captured lineage and creating association tables across all peers is crucial for high-throughput dataflows, to avoid overloading peers when there are input or processing skews in DISC actors. To do so, Newt uses weighted moving averages of CPU to find the peer with the least load, which is then assigned to the next new actor. Newt also finds the peer with the least disk utilization for creating new lineage association tables.

Figure 7.2 shows the Hadoop job throughput of the Newt cluster as we increase the number of concurrently executing jobs. We compare our CPU and

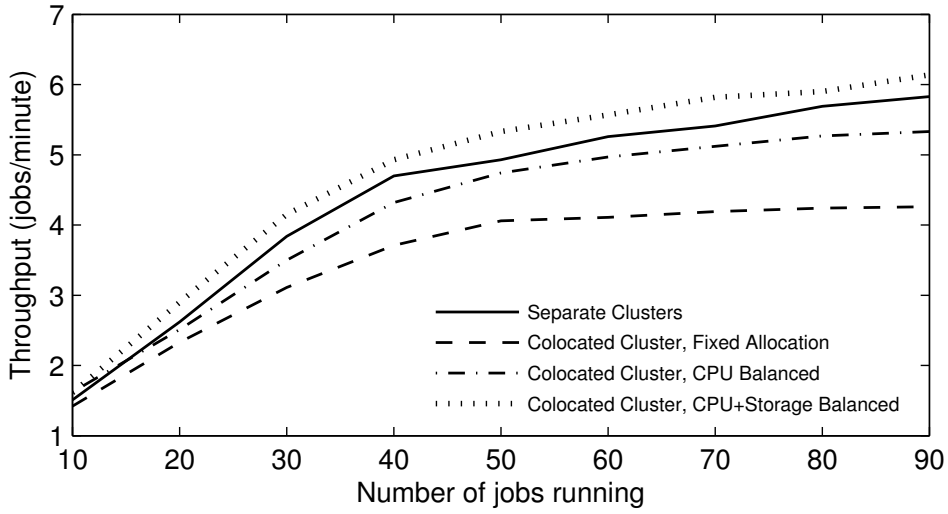


Figure 7.2: Average number of jobs completed by the DISC system per minute (while capturing lineage) for different co-location and load balancing policies in Newt. Dynamic CPU and disk load balancing improves throughput by 50% over fixed peer assignments.

disk load balancing scheme to a fixed assignment, as well as to a layout, in which the Newt cluster is separated from the DISC cluster. We observe that Newt's load balancing scheme improves job throughput by 50% over static, fixed peer assignments.

7.3 Capture overheads

Next we measure time and space overheads of capturing fine-grain lineage in DISC dataflows. Typically, the size of lineage captured depends on whether capture uses paired or unpaired APIs, and the selectivity of the actors in the dataflow, where the selectivity of an actor is a function of its input data and processing logic. We evaluate lineage capture on both Hadoop and Hyracks dataflows. In Hadoop, we ran PigMix [8] benchmarks and an identity MapReduce job. The PigMix benchmarks consist of a variety of operators, including `join`, `groupby`,

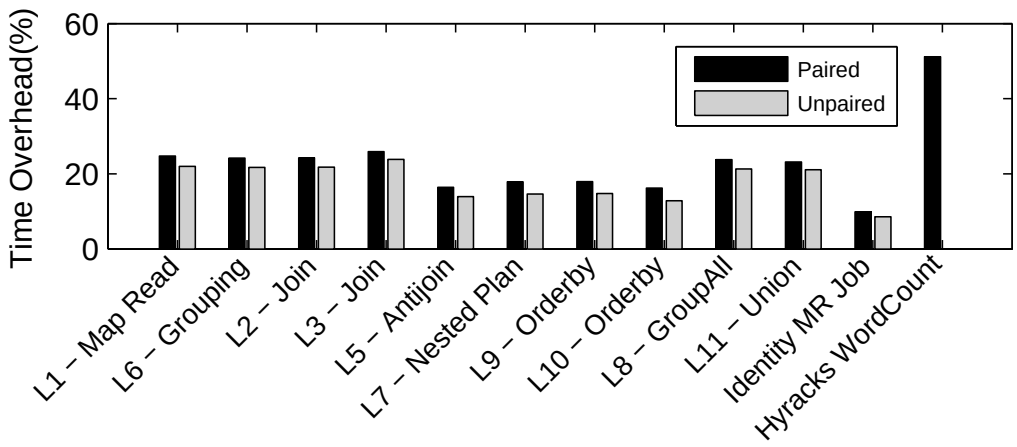


Figure 7.3: The time overheads of PigMix benchmarks, identity MapReduce job and Hyracks WordCount job with and without paired capture.

`orderby` and `union`. In Hyracks, we evaluate Newt on a WordCount program on a 34GB Wikipedia dataset.

Figure 7.3 shows that we incur low-to-moderate overheads when capturing lineage with Newt. The PigMix benchmarks and the identity job incur overheads ranging from 10-26%, while WordCount incurs 51%. The overhead for WordCount is larger because it is computationally inexpensive and generates too many lineage associations. We also observe that paired capture incurs higher time overheads, which is expected, since it uses twice as many API calls as unpaired capture. Unpaired capture incurs 9-24% time overheads for PigMix identity jobs, while overheads for these jobs are in the range 10-26% for paired capture. The identity job has the lowest overhead since it creates approximately 1 million associations per job versus an average 5 million for the PigMix jobs. Note that we do not evaluate Hyracks WordCount using unpaired capture because it is difficult to instrument without using separate capture calls for inputs and outputs.

However, paired capture also improves lineage accuracy and reduces the size of the captured lineage. Figure 7.4 shows the space overheads incurred as

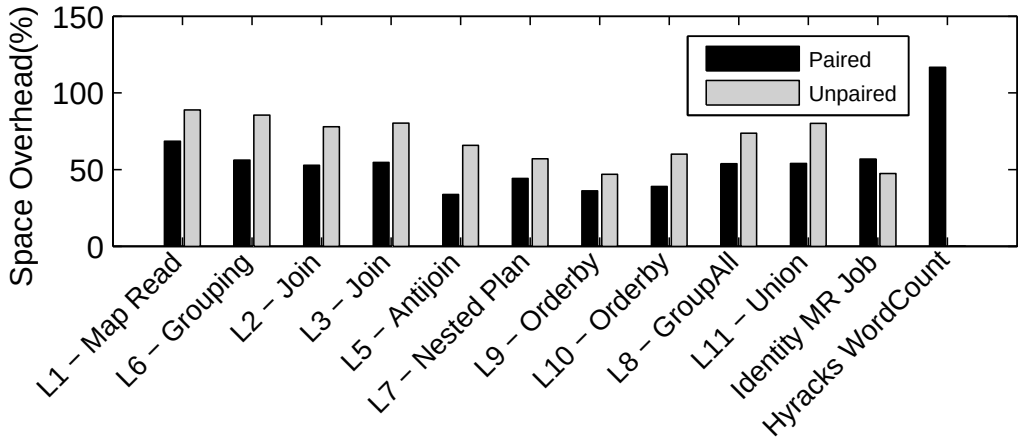


Figure 7.4: The space overheads (relative to total output data) of storing lineage for PigMix benchmarks, identity MapReduce jobs and Hyracks WordCount job with and without paired capture.

a percentage of the size of total output of these jobs. We calculate the size of stored lineage as the number of bytes on disk for all association tables, including all indexes on each table. Note that on an average, indexes add 10% overhead on table size. We observe that paired capture has smaller overheads than unpaired capture, requiring 31% less space on average. Again, the large space overheads for WordCount reflect the higher time overheads incurred due to large number of associations generated by the job.

We believe these overheads are reasonable in many scenarios when Newt is used for debugging dataflows. While space overheads approach 120%, storage is cheap compared to CPU costs. However, these workloads consist of short dataflows (2-3 jobs) and mostly use simple or relational operators, unlike several real-world analytics, which use complex non-relational operators and involve long multi-stage dataflows.

To evaluate Newt on such an analytics, we ran Contrail [53], a de novo genomic assembler for Hadoop. Contrail takes as input a set of short reads (short, ≤ 50, DNA sequences) and assembles the original genome sequence by building

and repeatedly refining De Bruijn graphs from these reads.

Genome assembly is both extremely CPU and data intensive. A genome assembly on 2MB of short reads took approximately 3 hours on a 4-node (3 slaves) Hadoop cluster (without lineage capture), and produced 49MB of total output. We evaluate Newt on an assembly of Bacillus subtilis subsp. natto BEST195 from 2.29GB of short read data [3], on a 65-node (64 slaves) EC2 cluster, running large instances with 7.5GB RAM, 4 compute units and "high" I/O performance. For this experiment, we ran the Newt cluster co-located with the Hadoop cluster (with dynamic load balancing).

Contrail stages produce over 20 times as much lineage per second as the most intensive PigMix program. Bacillus assembly required 145 MapReduce jobs and Newt captured lineage from 34927 actor instances. However, lineage capture only incurred 14% time overhead without requiring any additional hardware. The assembly produced 306GB of intermediate and final data, and Newt incurred space overheads of 86%.

7.4 Tracing query and selectivity

Here we explore how tracing query performance is affected by different table placement policies (Section 5.4). This experiment uses lineage captured from a MapReduce WordCount job on an input of 85k randomly generated text lines. The Hadoop job created 70 association tables from 49 mappers and 21 reducers.

Figure 7.5 shows the total query processing time for random, vertical co-location and horizontal co-location table placement policies. Each data point is the average of three runs, where each run consisted of both a forward and backward tracing query that randomly selects records to trace. The tracing time typically increases linearly as more records are traced. We observe that horizontal co-location yields poor query performance for this workload, while vertical co-location performs

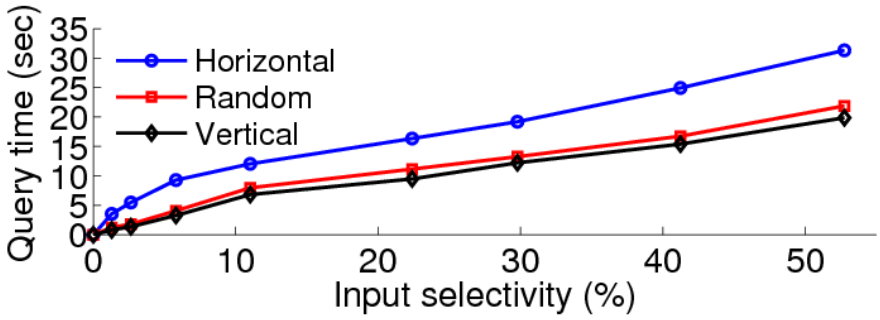


Figure 7.5: Tracing query latencies for different table placement policies.

best. We use vertical placement for all of our experiments.

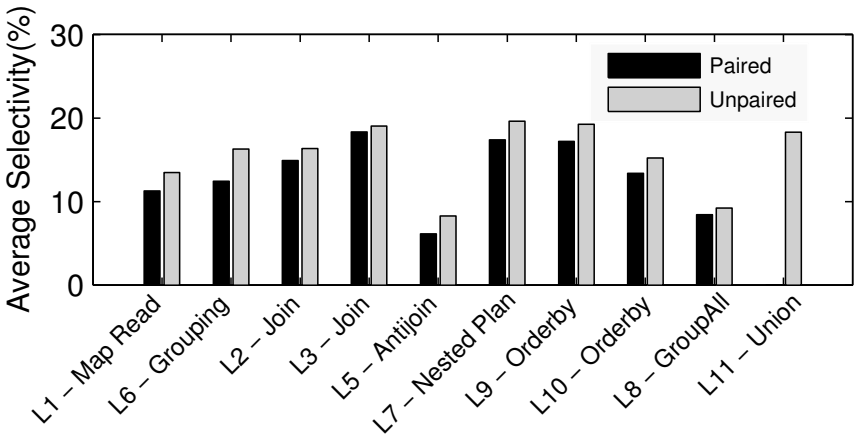


Figure 7.6: Tracing selectivity for PigMix jobs.

We also study *trace selectivity*, the per-actor ratio of the tracing table size to the association table size for the PigMix benchmarks. We can use trace selectivity to determine accuracy of the captured lineage and obtain a lower bound on the amount of work required to replay a dataflow on the tracing dataflow. Figure 7.6 shows the average percentage of input items that a randomly selected output depends upon for each job. From the figure, we observe that paired capture improves accuracy by 12% over unpaired capture.

7.5 Replay

This section evaluates Newt's selective replay accuracy and performance. These experiments used a MapReduce WordCount job with 635,000 lines of input generating 1,173,443 output records. We randomly select output records to replay and verify that the output reproduced was identical. A replayed dataflow can produce more records than those originally requested, due to the presence of one-many, many-many or non-monotonic transforms. These extra records represent potentially unnecessary work, and are filtered out at the downstream stage in the dataflow. We measure replay accuracy by comparing the number of these unnecessary records at each stage of the replay.

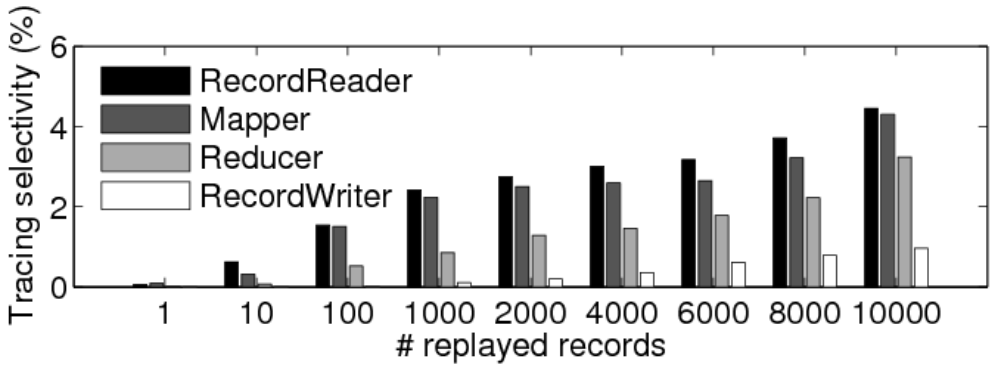


Figure 7.7: The percentage of data required to replay WordCount for each logical actor in the dataflow.

Figure 7.7 summarizes tracing selectivity for each logical actor. Since WordCount is a many-one operation, 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).

Next, we evaluate Newt's replay accuracy. Accurate replay depends on the ability to filter input data not present in the actor's tracing table (Section 3.5.2). Thus Newt installs tracing tables as filters in the clients. Figure 7.8 shows the *replay accuracy* for each logical actor. This is the ratio of the required output records

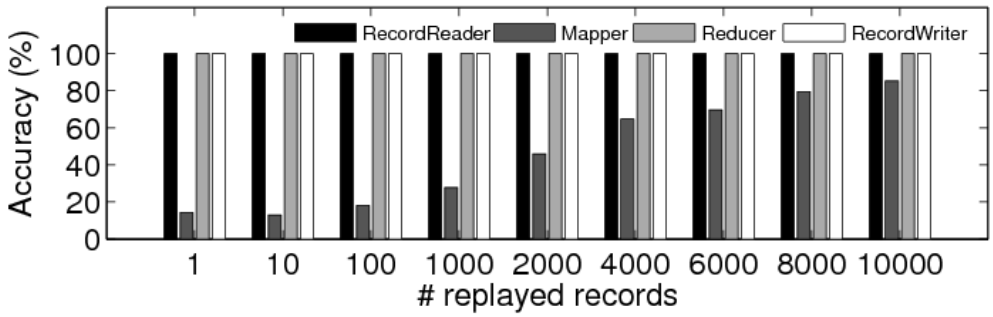


Figure 7.8: A replay may produce extra outputs besides the requested records. . other output besides the requested records. This graph shows the percentage of requested records in the entire output.

to the output records actually produced after replay. We observe that despite exact input filtering, the map actor generates additional records (since it is a one-many transform) by splitting input text lines into multiple words, some of whose counts were not selected to be regenerated. However, the replay for the rest of the actors is 100% accurate and the final replayed output is exact.

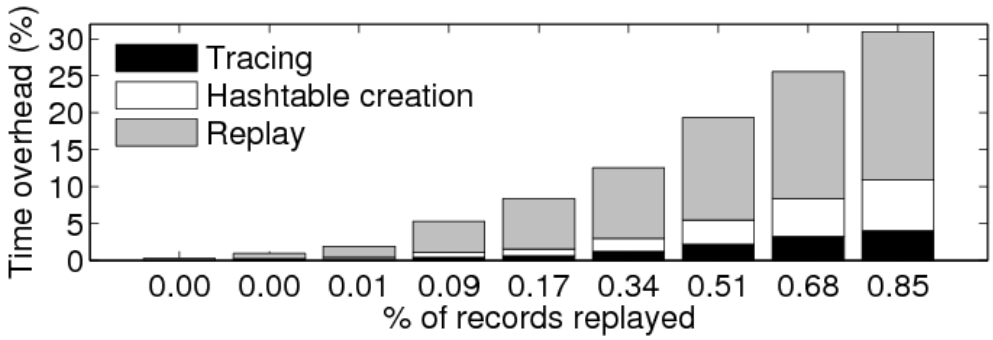


Figure 7.9: The percentage of normal running time required to reproduce a fraction of the output data set. For reference, the absolute count of reproduced records is identical to Figure 7.7.

Finally, we measure time to replay different subsets of original output, increasing in size. Figure 7.9 shows replay time, as the percentage of normal running time, as a function of the fraction of the output data set to regenerate. While the replay takes longer than a scaled version of the original execution, for small numbers of records (less than a 100), this holds true for WordCount. In fact, a single

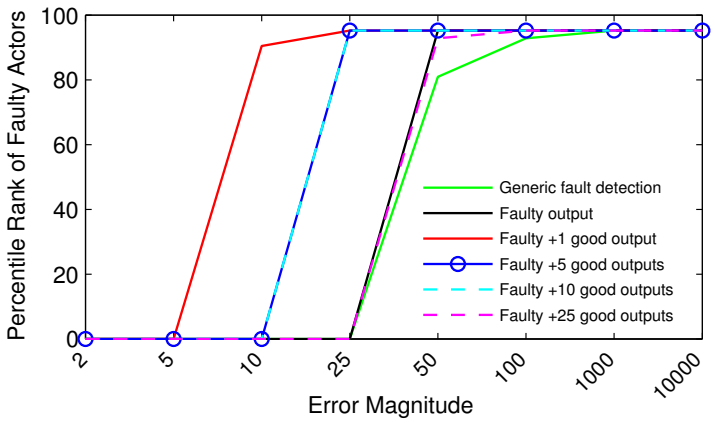


Figure 7.10: This figure illustrates the accuracy of the ranking scheme as a function of the magnitude of the error for different debugging queries and sample sizes of good outputs provided. As the error increases in size (left to right), the scheme ranks the offending actor higher.

record replay can execute in 0.3% of the original execution time. This is useful for step-wise debugging, where the user may want to replay the lineage of a single bad output and needs fast replay times, or for regenerating small sets of output (from a disk latent sector error), where it can be vastly less time intensive to regenerate the outputs with Newt.

7.6 Using fine-grain lineage for dataflow debugging

Finally, we evaluate the accuracy of our debugging methodology for identifying faulty actors in a dataflow using selectivity. We experiment with a dataflow that uses a sequence of three MapReduce jobs to join four different datasets, described below. The job has a total of 42 actor instances. Two faulty mapper instances are instrumented to produce spurious outputs when they encounter a specific key. We denote the number of spurious outputs produced as the *error magnitude* of the mapper. Because these datasets can be joined in several different join orders, we can move the faulty mappers to any of the three MapReduce stages in the pipeline.

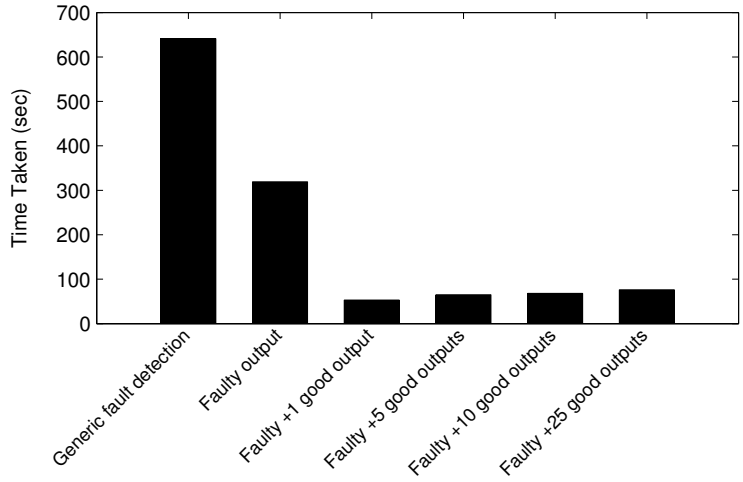


Figure 7.11: This figure shows the response time of each query. Providing more information (bad and good outputs) selects only a subset of the entire dataflow lineage for mining, and improves runtime by as much as 85%.

The job of our anomaly detector is to identify the faulty map actors.

The first dataset contains tuples of form $\{name, SSN, license\_number\}$. The second dataset contains tuples of form $\{SSN, outstanding\_balance\}$. The third dataset contains tuples of form $\{license\_number, offense\}$, and the final dataset contains tuples of form $\{name, age, gender\}$. The first faulty mapper in this application produces spurious $offense$ records for a specified $license\_number$. The second faulty mapper produces spurious $outstanding\_balance$ records for a specified SSN. We examine the accuracy of our ranking algorithm as a function of error magnitude. We test the effectiveness of Newt under three different conditions: user queries Newt for the general health of the dataflow without supplying any faulty outputs to be investigated, user supplies Newt with a known faulty output, and user supplies a known faulty output along with a sample of good outputs.

Figure 7.10 shows the percentile rank assigned to the faulty actor instances as a function of their error magnitude. As we expected, increasing error magnitude makes it easier to identify faulty actors, thus elevating their ranks. While providing a

single faulty output only marginally improves sensitivity, providing a small sample (5 or 10) of good inputs improves accuracy by enabling Newt to identify faulty actors that produce five or ten spurious outputs. However, adding too many good inputs masks the faulty behavior by adding too much noise, again reducing sensitivity. This behavior is at least partially because the Grubbs' method identifies a single outlier. Using more sophisticated algorithms to find multiple outliers may prove useful here.

Finally, we measure the query response time for each different query and different sample sizes of good outputs. We expect the time taken to be a function of the size of the lineage considered for each query. Figure 7.11 confirms this. While a general query to assess the health of the entire dataflow can be expensive, providing more information (bad and good outputs) limits the input data sizes and decreases the response time by 85%.

Chapter 8

Conclusions

DISC system analytics enable mining of large volumes of data for useful information. However, analyzing and debugging these analytics is one of the major challenges to utilizing the full potential of DISC systems and big data. To address these challenges, this thesis presents Newt, an architecture for capturing and using fine-grain lineage for analyzing and debugging in DISC systems. Newt actively collects fine-grain lineage from a DISC dataflow and enables efficient tracing, replay and debugging to provide transparency into the dataflow and trace individual data elements and errors through the analytics.

Experiments with different systems and analytics, including Apache Hadoop [1] and Hyracks [21], and a large-scale genomic assembly [53, 3], show that the Newt capture model is generic to different DISC systems and scales efficiently to multi-stage non-relational analytics. Capturing lineage with Newt incurs 10-51% runtime overheads for a variety of operators and workloads, while enabling a multitude of debugging options, such as step-wise debugging, removing inputs from a dataflow and retrospective lineage analysis to pinpoint errors in the dataflow. Newt accomplishes this through a unique instrumentation API, which supports common DISC system operators and works well with black-boxes, combined with a scalable capture, query and replay architecture, and outlier detection techniques for mining

captured lineage.

However, while Newt provides a powerful methodology for debugging DISC analytics, it also has limitations. Below, we discuss these limitations and interesting directions for future work.

- **Incomplete lineage.** Newt captures lineage through an instrumentation API that collects inputs and outputs at actor boundaries, instead of propagating lineage through the dataflow. This enables Newt to capture lineage across arbitrary UDFs. However, it also requires the developer to instrument all actors in the dataflow to capture complete lineage. If the developer overlooks a crucial actor, Newt is unable to successfully reconstruct the dataflow across these missing actors, and runs tracing queries only on the part of the dataflow that it could reconstruct.

- **Cyclic dataflows.** Newt performs output-input matching during tracing assuming acyclic physical dataflows. If a tracing query encounters cyclic dependencies between physical actor instances (including an actor instance reading its own output), it may return incomplete results or possibly loop forever through the cycle until killed. However, since most DISC system dataflows are physically structured as DAGs [27, 1, 21, 41], this is largely not an issue for such systems. Note that Newt can handle logical cycles as long as the cycles are unrolled in the physical dataflow.

- **Non-deterministic actors.** Non-deterministic actors can produce different outputs on the same input dataset during different executions. As such, Newt is unable to accurately replay non-deterministic actors, since any lineage captured for such actors is uncertain at best.

These limitations open up several opportunities for future research, including using copy-provenance [40], tracing functions [26] or probabilistic techniques [54] to trace across missing actors. Use of additional metadata, such as versioning, to

handle cyclic dependencies is another interesting idea. Other research directions include sampling lineage to reduce overheads, while still enabling retrospective lineage analysis to identify faulty actors, and using lineage for data cleaning [51]. This research can also benefit from further evaluations of our replay and debugging methodologies on real-world applications, more sophisticated outlier detection techniques involving multiple dataflow properties and a gradational scoring scheme for faulty actors.

Bibliography

[1] Apache Hadoop. http://hadoop.apache.org.

[2] Apache Oozie. http://incubator.apache.org/oozie/.

[3] Bacillus subtilis subsp. natto best195 genome sequencing project. http://www.ncbi.nlm.nih.gov/bioproject/38027.

[4] Climate data analytics using mapreduce. https://portal.futuregrid.org/projects/192.

[5] Hessian Binary Web Service Protocol. http://hessian.caucho.com.

[6] Large synoptic survey telescope. http://www.lsst.org/lsst/public/tour_software.

[7] MySQL 5.5 Reference Manual. http://dev.mysql.com/doc/refman/5.5/en.

[8] Pigmix benchmark. https://cwiki.apache.org/PIG/pigmix.html.

[9] Google processing 20 petabytes a day. http://www.datacenterknowledge.com/archives/2008/01/09/google-processing-20-petabytes-a-day, 2008.

[10] http://www.smartercomputingblog.com/2011/05/11/new-approaches-for-big-data-part-1-introduction, 2010.

[11] The data deluge in genomics. https://www-304.ibm.com/connections/blogs/ibm_healthcare/entry/data_overload_in_genomics3?lang=de, 2010.

[12] Libraries and archives for future generations. http://netpreserve.org/events/2010GApresentations/01_iipc_Singapore_Anderson_2010.pdf, 2010.

[13] Scaling facebook to 500 million users and beyond. http://www.facebook.com/note.php?note_id=409881258919, 2010.

[14] Google search terms used to track disease outbreaks. http://www.pcworld.com/article/228527/google_search_terms_used_to_track_disease_outbreaks.html, 2011.

[15] Walmartlabs launches shopycat for social gift suggestions. WalmartLabs Launches Shopycat For Social Gift Suggestions, 2011.

[16] Forecasting with internet search data. http://libertystreeteconomics.newyorkfed.org/2012/01/forecasting-with-internet-search-data.html, 2012.

[17] Hiralal Agrawal and Joseph R. Horgan. Dynamic program slicing. In *Proc. of ACM SIGPLAN*, June 1990.

[18] Yael Amsterdamer, Susan B. Davidson, Daniel Deutch, Tova Milo, and Julia Stoyanovich. Putting lipstick on a pig: Enabling database-style workflow provenance. In *Proc. of VLDB*, August 2011.

[19] Przemyslaw Kazienko andKatarzyna Musial and TomaszKajdanowicz. Multidimensional social network and itsapplication to the social recommender system. *IEEE Transactions on Systems, Man and Cybernetics - Part A: Systems and Humans*, 41(4):746–759, 2011.

[20] L. Bairavasundaram, G. Goodson, S. Pasupathy, and J. Schindler. An analysis of latent sector errors in disk drives. In *Proceedings of SIGMETRICS'07*, 2007.

[21] Vinayak Borkar, Michael Carey, Raman Grover, Nicola Onose, and Rares Vernica. Hyracks: A flexible and extensible foundation for data-intensive computing. In *Proc. of ICDE*, 2011.

[22] Peter Buneman, James Cheney, and Stijn Vansummeren. On the expressiveness of implicit provenance in query and update languages. *ACM Transactions on Database Systems*, 33(4):28:1–28:47, December 2008.

[23] Peter Buneman, Sanjeev Khanna, and Wang Chiew Tan. Data provenance: Some basic issues. In *Proceedings of the 20th Conference on Foundations of SoftwareTechnology and Theoretical Computer Science*, FST TCS 2000, pages 87–93, London, UK, UK, 2000. Springer-Verlag.

[24] Ronnie Chaiken, Bob Jenkins, Per-Ake Larson, Bill Ramsey, Darren Shakib, Simon Weaver, and Jingren Zhou. SCOPE: Easy and efficient parallel processing of massive data sets. In *Proceedings of the 34th International Conference on Very large Data Bases (VLDB'08)*, August 2008.

[25] Shimin Chen and Steven W. Schlosser. Map-reduce meets wider varieties of applications. Technical report, Intel Research, 2008.

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

[27] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[28] Jaliya Ekanayake, Shrideep Pallickara, and Geoffrey Fox. Mapreduce for data intensive scientific analyses. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 277–284, Washington, DC, USA, 2008. IEEE Computer Society.

[29] Hao Fan and Ra Poulovassilis. Using schema transformation pathways for data lineage tracing. In *In BNCOD*, pages 133–144. Springer, 2005.

[30] Rodrigo Fonseca, George Porter, Randy H. Katz, Scott Shenker, and Ion Stoica. X-trace: A pervasive network tracing framework. In *In Proceedings of NSDI'07*, 2007.

[31] Ian Foster, Jens Vockler, Michael Wilde, and Yong Zhao. Chimera: A Virtual Data System for Representing, Querying, and Automating Data Derivation. In *14th International Conference on Scientific and Statistical Database Management*, July 2002.

[32] Jerome Francois, Shaonan Wang, Walter Bronzi, Radu State, and Thomas Engel. Botcloud: Detecting botnets using mapreduce. *Information Forensics and Security, IEEE International Workshop on*, 0:1–6, 2011.

[33] Helena Galhardas, Daniela Florescu, Dennis Shasha, Eric Simon, and Cristian-Augustin Saita. Improving data cleaning quality using a data lineage facility. In *DMDW*, volume 39 of *CEUR Workshop Proceedings*, page 3. CEUR-WS.org, 2001.

[34] Boris Glavic and Gustavo Alonso. Perm: Processing provenance and data on the same data model through query rewriting. *2009 IEEE 25th International Conference on Data Engineering*, pages 174–185, 2009.

[35] Frank E Grubbs. Procedures for detecting outlying observations in samples. *Technometrics*, 11(1):1–21, 1969.

[36] Pradeep Kumar Gunda, Lenin Ravindranath, Chandramohan A. Thekkath, Yuan Yu, and Li Zhuang. Nectar: automatic management of data and computation in datacenters. In *Proceedings of OSDI'10*, 2010.

[37] Dustin Hillard, Stefan Schroedl, Eren Manavoglu, Hema Raghavan, and Chirs Leggetter. Improving ad relevance in sponsored search. In *Proceedings of the third ACM international conference on Websearch and data mining*, WSDM '10, pages 361–370, New York, NY, USA, 2010. ACM.

[38] Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. In *Proc. of CIDR*, January 2011.

[39] Robert Ikeda, Semih Salihoglu, and Jennifer Widom. Provenance-based refresh in data-oriented workflows. In *Proceedings of the 20th ACM international conference on Information and knowledge management*, CIKM '11, pages 1659–1668, New York, NY, USA, 2011. ACM.

[40] Robert Ikeda and Jennifer Widom. Data lineage: A survey. Technical report, Stanford University, 2009.

[41] Michael Isard, Mihai Budiu, Yuan Yu, Andrew Birrell, and Dennis Fetterly. Dryad: distributed data-parallel programs from sequential building blocks. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference onComputer Systems 2007*, EuroSys '07, pages 59–72, New York, NY, USA, 2007. ACM.

[42] Xiang Li, Beth Plale, Nithya N. Vijayakumar, Rahul Ramachandran, Sara J. Graves, and Helen Conover. Real-time storm detection and weather forecast activation through data mining and events processing. *Earth Science Informatics*, 1(2):49–57, 2008.

[43] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 international conference on Managementof data*, SIGMOD '10, pages 135–146, New York, NY, USA, 2010. ACM.

[44] Andréa Matsunaga, Maurício Tsugawa, and José Fortes. Cloudblast: Combining mapreduce and virtualization on distributedresources for bioinformatics applications. In *Proceedings of the 2008 Fourth IEEE International Conference on eScience*, ESCIENCE '08, pages 222–229, Washington, DC, USA, 2008. IEEE Computer Society.

[45] Kiran-Kumar Muniswamy-Reddy, Joseph Barillariy, Uri Braun, David A. Holland, Diana Maclean, and Margo Seltzer. Layering in provenance systems. In *Proceedings of USENIX Technical Conference*, June 2009.

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

[47] Christopher Olston, Greg Chiou, Laukik Chitnis, Francis Liu, Yiping Han, Mattias Larsson, Andreas Neumann, Vellanki B.N. Rao, Vijayanand Sankarasubramanian, Siddharth Seth, Chao Tian, Topher ZiCornell, and Xiaodan Wang. Nova: continuous pig/hadoop workflows. In *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, pages 1081–1090, New York, NY, USA, 2011. ACM.

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

[49] Christopher Olston, Benjamin Reed, Utkarsh Srivastava, Ravi Kumar, and Andrew Tomkins. Pig latin: A not-so-foreign language for data processing. In *Proc. of ACM SIGMOD*, Vancouver, Canada, June 2008.

[50] Luca Pireddu, Simone Leo, and Gianluigi Zanetti. Mapreducing a genomic sequencing workflow. In *Proceedings of the second international workshop on MapReduce and its applications*, MapReduce '11, pages 67–74, New York, NY, USA, 2011. ACM.

[51] Erhard Rahm and Hong H. Do. Data Cleaning: Problems and Current Approaches. *IEEE Data Engineering Bulletin*, 23(4):3–13, 2000.

[52] Anish Das Sarma, Alpa Jain, and Philip Bohannon. PROBER: Ad-Hoc Debugging of Extraction and Integration Pipelines. Technical report, Yahoo, April 2010.

[53] Michael Schatz, Avijit Gupta, Rushil Gupta, David Kelley, Jeremy Lewi, Deepak Nettem, Dan Sommer, and Miahi Pop. Contrail: Assembly of Large Genomes using Cloud Computing. http://sourceforge.net/apps/mediawiki/contrail-bio.

[54] Benjamin H. Sigelman, Luiz Andr Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google Inc, 2010.

[55] Yogesh L. Simmhan, Beth Plale, and Dennis Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, September 2005.

[56] Salvatore J. Stolfo, David W. Fan, Wenke Lee, Andreas L. Prodromidis, and Philip K. Chan. Credit card fraud detection using meta-learning: Issues and initial results. In *of AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 83–90, 1997.

[57] Vinay Sudhakaran and Neil P. Chue Hong. Evaluating the suitability of mapreduce for surface temperature analysis codes. In *Proceedings of the second international workshop on Dataintensive computing in the clouds*, DataCloud-SC '11, pages 3–12, New York, NY, USA, 2011. ACM.

[58] Ashish Thusoo, Joydeep Sen Sarma, Namit Jain, Zheng Shao, Prasad Chakka, Suresh Anthony, Hao Liu, Pete Wyckoff, and Raghotham Murthy. Hive -

a warehousing solution over a map-reduce framework. In *Proceedings of the VLDB endowment (vldb '09)*, pages 1626–1629, 2009.

[59] Shashank Tiwari. *Professional NoSQL*. John Wiley & Sons, 2011.

[60] Stijn Vansummeren and James Cheney. Recording provenance for sql queries and updates. *IEEE Data Eng. Bull.*, 30(4):29–37, 2007.

[61] Fei Wang, Vuk Ercegovac, Tanveer Syeda-Mahmood, Akintayo Holder, Eugene Shekita, David Beymer, and Lin Hao Xu. Large-scale multimodal mining for healthcare with mapreduce. In *Proceedings of the 1st ACM International Health Informatics Symposium*, IHI '10, pages 479–483, New York, NY, USA, 2010. ACM.

[62] Yuan Yu, Michael Isard, Dennis Fetterly, Mihai Budiu, Ulfar Erlingsson, Pradeep Kumar Gunda, , and Jon Currey. DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing Using a High-Level Language. In *OSDI'08*, San Diego, CA, December 2008.

[63] Matei Zaharia. Resilient distributed datasets : A fault-tolerant abstraction for in-memory cluster computing. *Engineering*, 66:1025–1043, 2011.

[64] Mingwu Zhang, Xiangyu Zhang, Xiang Zhang, and Sunil Prabhakar. Tracing lineage beyond relational operators. In *Proc. Conference on Very Large Data Bases (VLDB)*, September 2007.

[65] Wenchao Zhou, Qiong Fei, Arjun Narayan, Andreas Haeberlen, Boon Thau Loo, and Micah Sherr. Secure network provenance. In *Proceedings of 23rd ACM Symposium on Operating System Principles (SOSP)*, December 2011.