Data Lineage for Big Data Systems

"**Data lineage** is defined as a data life cycle that includes the data's origins and where it moves over time." [1] It describes what happens to data as it goes through diverse processes. It helps provide 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. In fact, database systems have used such information, called data provenance, to address similar validation and debugging challenges already.[2]

Data provenance documents the inputs, entities, systems, and processes that influence data of interest, in effect providing a historical record of the data and its origins. The generated evidence supports essential forensic activities such as data-dependency analysis, error/compromise detection and recovery, and auditing and compliance analysis. "**Lineage** is a simple type of **why provenance**."[2]

1 Case for Data Lineage

The world of big data is changing dramatically right before our eyes. Statistics say that Ninety percent (90%) of the world's data has been created in the last two years alone. [3]. This explosion of data has resulted in the ever-growing number of systems and automation at all levels in all sizes of organizations.

Today, distributed systems like Google Map Reduce[4], Microsoft Dryad[5], Apache Hadoop [6](an open-source project) and Google Pregel[7] 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[8]. "The Large Synoptic Survey Telescope is expected to generate terabytes of data every night and eventually store more than 50 petabytes, while in the bioinformatics sector, the largest genome 12 sequencing houses in the world now store petabytes of data apiece."[9] Due to the humongous size of the big data, there could be features in the data that are not considered in the machine learning algorithm, possibly even outliers. It is very difficult for a data scientist to trace an unknown or an unanticipated result.

1.1 Big Data Debugging

Big data analytics is the process of examining large data sets to uncover hidden patterns, unknown correlations, market trends, customer preferences and other useful business information. They apply machine learning algorithms etc to the data which transform the data. Due to the humongous size of the data, there could be unknown features in the data, possibly even outliers. It is pretty difficult for a data scientist to actually debug an unexpected result.

The massive scale and unstructured nature of data, the complexity of these analytics 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 [10][11][12][13].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 Challenges in Big Data Debugging

1.2.1 Massive Scale

The past two decades have seen a nuclear explosion in the collection and storage of digital information. In 2012, 2.8 zettabytes—that's 1 sextillion bytes, or the equivalent of 24 quintillion tweets—were created or replicated, according to the research firm IDC. There are hundreds or thousands of petabyte-scale databases today, and we'd compare their size to what existed two decades ago, only every time the basis of comparison would be zero. Here's a look at some of the world's largest and most interesting data sets. Working with this scale of data has become very challenging [14].

1.2.2 Unstructured Data

The phrase unstructured data usually refers to information that doesn't reside in a traditional row-column

database. As you might expect, it's the opposite of structured data the data stored in fields in a database. Unstructured data files often include text and multimedia content. Examples include e-mail messages, word processing documents, videos, photos, audio files, presentations, webpages and many other kinds of business documents. Note that while these sorts of files may have an internal structure, they are still considered "unstructured" because the data they contain doesn't fit neatly in a database. Experts estimate that 80 to 90 percent of the data in any organization is unstructured. And the amount of unstructured data in enterprises is growing significantly often many times faster than structured databases are growing. "Big data can include both structured and unstructured data, but IDC estimates that 90 percent of big data is unstructured data."[15]

1.2.3 Long Runtime

In today's hyper competitive business environment, companies not only have to find and analyze the relevant data they need, they must find it quickly. The challenge is going through the sheer volumes of data and accessing the level of detail needed, all at a high speed. The challenge only grows as the degree of granularity increases. One possible solution is hardware. Some vendors are using increased memory and powerful parallel processing to crunch large volumes of data extremely quickly. Another method is putting data in-memory but using a grid computing approach, where many machines are used to solve a problem. Both approaches allow organizations to explore huge data volumes. Even this this level of sophisticated hardware and software, few of the image processing tasks in large scale take a few days to few weeks[16]. Debugging of the data processing is extremely hard due to long run times.

1.2.4 Complex Platform

Big Data platforms have a very complicated structure. Data is distributed among several machines. Typically the jobs are mapped into several machines and results are later combined by reduce operations. Debugging of a big data pipeline becomes very challenging because of the very nature of the system. It will not be an easy task for the data scientist to figure out which machine's data has the outliers and unknown features causing a particular algorithm to give unexpected results.

1.3 Proposed Solution

Data provenance or data lineage can be used make the debugging of big data pipeline easier. This necessitates the collection of data about data transformations. The below section will explain data provenance in more detail.

2 Data Provenance

Data Provenance provides a historical record of the data and its origins. The provenance of data which is generated by complex transformations such as workflows is of considerable value to scientists. From it, one can ascertain the quality of the data based on its ancestral data and derivations, track back sources of errors, allow automated re-enactment of derivations to update a data, and provide attribution of data sources. Provenance is also essential to the business domain where it can be used to drill down to the source of data in a data warehouse, track the creation of intellectual property, and provide an audit trail for regulatory purposes.

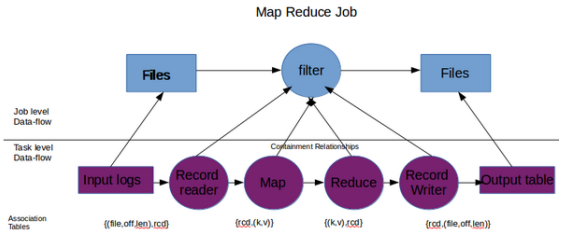
The use of data provenance is proposed in distributed systems to trace records through a dataflow, replay the dataflow on a subset of its original inputs and debug data flows. To do so, one needs to keep track of the set of inputs to each operator, which were used to derive each of its outputs. Although there are several forms of provenance, such as copy-provenance and how-provenance [13], [17], the information we need is a simple form of **why-provenance, or lineage**, as defined by Cui et al[18].

3 Lineage Capture

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 ?"[2] 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[19]. Backward tracing is useful for debugging, while forward tracing is useful for tracking error propagation.[19] Tracing queries also form the basis for replaying an original dataflow [18][11][19]. 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.

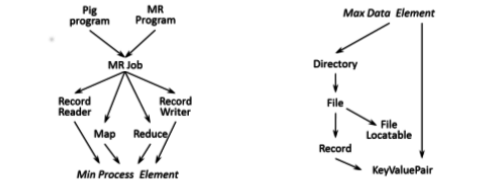
DISC system consists of several levels of operators and data, and different use cases of lineage can dictate the level at which lineage needs to be captured. Lineage can be captured at the level of the job, using files and giving lineage tuples of form {IF i , M RJob, OF i }, lineage can also be captured 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.[2]



Map Reduce Job showing containment relashionshpis

To capture end-to-end lineage in a DISC system, we use the Ibis model [20], 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."[2] 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).



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

Containment Hierarchy

4 Active vs Lazy Lineage

Lazy lineage collection typically captures only coarse-grain lineage at run time. These systems incur low capture overheads due to the small amount of lineage they capture. However, to answer fine-grain tracing queries, they must replay the data flow 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.

Active collection systems capture entire lineage of the data flow at run time. The kind of lineage they capture may be coarse-grain or fine-grain, but they do not require any further computations on the data flow after its execution. Active fine-grain lineage collection systems incur higher capture overheads than lazy collection systems. However, they enable sophisticated replay and debugging.[2]

5 Actors

An actor is an entity that transforms data; it may be a Dryad vertex, individual map and reduce operators, a MapReduce job, or an entire dataflow pipeline. Actors act as black-boxes and the inputs and outputs of an actor are tapped 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. The system developer needs to capture the data an actor reads (from other actors) and the data an actor writes (to other actors). For example, a developer can treat the Hadoop Job Tracker as an actor by recording the set of files read and written by each job. [21]

6 Associations

Association is a combination of the inputs, outputs and the operation itself. The operation is represented in terms of a black box also known as the actor. The associations describe the transformations that are applied on the data. The associations are stored in the association tables. Each unique actor is represented by its own association table. An association itself looks like {i, T, o} where i is the set of inputs to the actor T and o is set of outputs given produced by the actor. Associations are the basic units of Data Lineage. Individual associations are later clubbed together to construct the entire history of transformations that were applied to the data.[2]

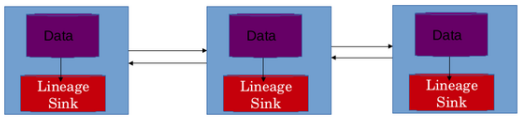
7 Architecture

Big data systems scale horizontally i.e increase capacity by by adding new hardware or software entities into the distributed system. The distributed system acts as a single entity in the logical level even though it comprises of multiple hardware and software entities. The system should continue to maintain this property after horizontal scaling. An important advantage of horizontal scalability is that it can provide the ability to increase capacity on the fly. The biggest plus point is that horizontal scaling can be done using commodity hardware.

The horizontal scaling feature of Big Data systems should be taken into account while creating the architecture of lineage store. This is essential because the lineage store itself should also be able scale in parallel with the Big

data system. The number of associations and amount of storage required to store lineage will increase with the increase in size and capacity of the system. The architecture of Big data systems makes the use of a single lineage store not appropriate and impossible to scale. The immediate solution to this problem is to distribute the lineage store itself.[2]

The best case scenario is to use a local lineage store for every machine in the distributed system network. This allows the lineage store also to scale horizontally. In this design, the lineage of data transformations applied to the data on a particular machine is stored on the local lineage store of that specific machine. The lineage store typically stores association tables. Each actor is represented by its own association table. The rows are the associations themselves and columns represent inputs and outputs. This design solves 2 problems. It allows horizontal scaling of the lineage store. If a single centralized lineage store was used, then this information had to be carried over the network, which would cause additional network latency. The network latency is also avoided by the use of a distributed lineage store.[21]



Architecture of Lineage Systems

8 Data flow Reconstruction

The information stored in terms of associations needs to be combined by some means to get the data flow of a particular job. In a distributed system a job is broken down into multiple tasks. One or more instances run a particular task. The results produced on these individual machines are later combined together to finish the job. Tasks running on different machines perform multiple transformations on the data in the machine. All the transformations applied to the data on a machines is stored in the local lineage store of that machines. This information needs to combined together to get the lineage of the entire job. The lineage of the entire job should help the data scientist understand the data flow of the job and he/she can use the data flow to debug the big data pipeline. The data flow is reconstructed in 3 stages.

8.1 Association tables

The first stage of the data flow reconstruction is the computation of the association tables. The association tables exists for each actor in each local lineage store. The entire association table for an actor can be computed by combining these individual association tables. This is generally done using a series of equality joins based on the actors themselves. In few scenarios the tables might also be joined using inputs as the key. Indexes can also be used to improve the efficiency of a join.The joined tables need to be stored on a single instance or a machine to further continue processing. There are multiple schemes that are used to pick a machine where a join would be computed. The easiest one being the one with minimum CPU load. Space constraints should also be kept in mind while picking the instance where join would happen.

8.2 Association Graph

The second step in data flow reconstruction is computing an association graph from the lineage information. The graph represents the steps in the data flow. The actors act as vertices and the associations act as edges. Each actor T is linked to its upstream and downstream actors in the data flow. 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 . Containment relationships are always considered while creating the links. The graph consists of three types of links or edges.

8.2.1 Explicitly specified links

The simplest link is an explicitly specified link between two actors. These links are explicitly specified in the code of a machine learning algorithm. When an actor is aware of its exact upstream or downstream actor, it can communicate this information to lineage API. This information is later used 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.[2]

8.2.2 Logically inferred links

Developers can attach data flow archetypes to each logical actor. A data flow archetype explains how the children types of an actor type arrange themselves in a data flow. With the help of this information, one 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. The system infers this from the data flow archetypes and duly links map instances with reduce instances. However, there may be several MapReduce jobs in the data flow, 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.[2]

8.2.3 Implicit links through data set sharing

In distributed systems, sometimes there are implicit links, which are not specified during execution. For example, an implicit link exists between an actor that wrote to a file and another actor that read from it. Such links connect actors which use a common data set for execution. The dataset is the output of the first actor and is the input of the actor following it.[2]

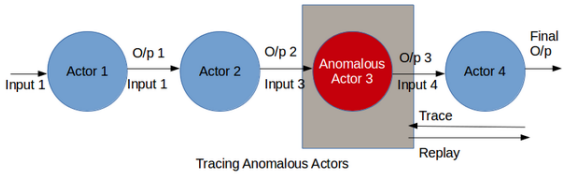
8.3 Topological Sorting

The final step in the data flow reconstruction is the Topological sorting of the association graph. The directed graph created in the previous step is topologically sorted to obtain the order in which the actors have modified the data. This inherit order of the actors defines the data flow of the big data pipeline or task.

9 Tracing & Replay

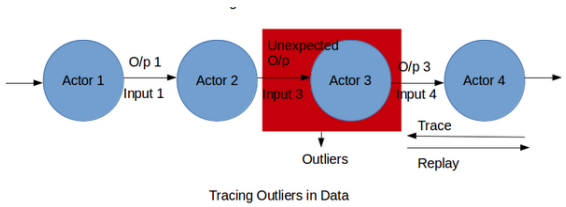
This is the most crucial step in Big Data debugging. The captured lineage is combined and processed to obtain the data flow of the pipeline. The data flow helps the data scientist or a developer to look deeply into the actors and their transformations. This step allows the data scientist to figure out the part of the algorithm that is generating the unexpected output. A big data pipeline can go wrong in 2 broad ways. The first is a presence of a suspicious actor in the data-flow. The second being the existence of outliers in the data.

The first case can be debugged by tracing the data-flow. By using lineage and data-flow information together a data scientist can figure out how the inputs are converted into outputs. During the process actors that behave unexpectedly can be caught. Either these actors can be removed from the data flow or they can be augmented by new actors to change the data-flow. The improved data-flow can be replayed to test the validity of it. Debugging faulty actors include recursively performing coarse-grain replay on actors in the data-flow [22], which can be expensive in resources for long dataflows. Another approach is to manually inspect lineage logs to find anomalies [12][23], which can be tedious and time-consuming across several stages of a data-flow. Furthermore, these approaches work only when the data scientist can discover bad outputs. To debug analytics without known bad outputs, the data scientist need to analyze the data-flow for suspicious behavior in general. However, often, a user may not know the expected normal behavior and cannot specify predicates. This section describes a debugging methodology for retrospectively analyzing lineage to identify faulty actors in a multi-stage data-flow. 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 data-flow.



Tracing Anomalous Actors

The second problem i.e the existence of outliers can also be identified by running the data-flow step wise and looking at the transformed outputs. The data scientist finds a subset of outputs that are not in accordance to the rest of outputs. The inputs which are causing these bad outputs are the outliers in the data. This problem can be solved by removing the set of outliers from the data and replaying the entire data-flow. It can also be solved by modifying the machine learning algorithm by adding, removing or moving actors in the data-flow. The changes in the data-flow are successful if the replayed data-flow does not produce bad outputs.



Tracing Outliers in the data

10 Challenges

Even though use data lineage is a novel way of debugging of big data pipelines, the process is not simple. The challenges are scalability of lineage store, fault tolerance of the the lineage store, accurate capture of lineage for black box operators and many others. These challenges must be considered carefully and trade offs between them need to be evaluated to make a realistic design for data lineage capture.

10.1 Scalability

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.

10.2 Fault tolerance

Lineage capture systems must also be fault tolerant to avoid rerunning data flows 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. A lineage system should also be able to gracefully handle multiple instances of local lineage systems going down. This can achieved by storing replicas of lineage associations in multiple machines. The replica can act like a backup in the event of the real copy being lost.

10.3 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 data-flow several times to deduce the minimal set[24], and dynamic slicing, as used by Zhang et al.[25] 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.

10.4 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. [19]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[26], a lineage system for Pig[27], 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.

10.5 Sophisticated replay

Replaying only specific inputs or portions of a data-flow 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[28]. 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.

10.6 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.[2]

11 See also

- Provenance
- Big Data
- Topological Sorting
- Debugging
- NoSQL
- Scalability
- Directed acyclic graph

12 References

[1] http://www.techopedia.com/definition/28040/data-lineage

[2] De, Soumyarupa. (2012). Newt : an architecture for lineage based replay and debugging in DISC systems. UC San Diego: b7355202. Retrieved from: https://escholarship.org/uc/item/3170p7zn

[3] http://newstex.com/2014/07/12/thedataexplosionin2014minutebyminuteinfographic/

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

[5] 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.

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

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

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

[9] The data deluge in genomics. https://www-304.ibm.com/connections/blogs/ibmhealthcare/entry/data overload in genomics3?lang=de, 2010.

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

[11] 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.

[12] 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.

[13] 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

[14] The Wired. http://www.wired.com/2013/04/bigdata/

[15] Webopedia http://www.webopedia.com/TERM/U/unstructured_data.html

[16] SAS. http://www.sas.com/resources/asset/five-big-data-challenges-article.pdf

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

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

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

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

[21] Dionysios Logothetis, Soumyarupa De, and Kenneth Yocum. 2013. Scalable lineage capture for debugging DISC analytics. In Proceedings of the 4th annual Symposium on Cloud Computing (SOCC '13). ACM, New York, NY, USA, , Article 17 , 15 pages.

[22] 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.

[23] 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.

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

[25] 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.

[26] 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.

[27] 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.

[28] 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.

13 Text and image sources, contributors, and licenses

13.1 Text

- **Data Lineage for Big Data Systems** *Source:* http://en.wikipedia.org/wiki/Data%20Lineage%20for%20Big%20Data%20Systems?oldid=638867285 *Contributors:* -revi and Skamisetty

13.2 Images

- **File:Containment_Heirarchy.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/1/1c/Containment_Heirarchy.png *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Skamisetty
- **File:Map_Reduce_Job_-1.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/a/aa/Map_Reduce_Job_-1.png *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Skamisetty
- **File:Selection_065.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/c/c4/Selection_065.png *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Skamisetty
- **File:Tracing_Anomalous_Actors.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/d/db/Tracing_Anomalous_Actors.png *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Skamisetty
- **File:Tracing_Outliers_in_the_data.png** *Source:* http://upload.wikimedia.org/wikipedia/commons/0/07/Tracing_Outliers_in_the_data.png *License:* CC BY-SA 4.0 *Contributors:* Own work *Original artist:* Skamisetty

13.3 Content license

- Creative Commons Attribution-Share Alike 3.0