2019 IEEE 35th International Conference on Data Engineering (ICDE)

SAC: A System for Big Data Lineage Tracking

Mingjie Tang*, Saisai Shao*, Weiqing Yang*,Yanbo Liang*,Yongyang Yu†,Bikas Saha*, Dongjoon Hyun*
*Hortonworks †Facebook
{tangrock, saisai, yangweiqing001}@gmail.com, {yliang, bikas}@hortonworks.com, yongyangyu@fb.com

***Abstract*—In the era of big data, a data processing flow contains various types of tasks. It is nontrivial to discover the data flow/movement from its source to destination, such that monitoring different transformations and hops on its way in an enterprise environment. Therefore, data lineage or provenance is useful to learn how the data gets transformed along the way, how the representation and parameters change, and how the data splits or converges after each hop. However, existing systems offer limited support for such use cases in a distributed computing setup. To address this issue, we build Spark-Atlas-Connector (short as SAC), a new system to track data lineage in a distributed computation platform, e.g., Spark. SAC tracks different processes involved in the data flow and their dependencies, supporting different data storage (e.g., HBase, HDFS, and Hive) and data processing paradigms (e.g., SQL, ETL, machine learning, and streaming). SAC provides a visual representation of data lineage to track data from its origin to downstreams, and is deployed in a distributed production environment for demonstrating its efficiency and scalability.**

***Index Terms*—data provenance and lineage, big data computation, machine learning**

I. INTRODUCTION

Data provenance (or data lineage) provides historical records of data and its origins. The provenance of data generated by complex transformations is of considerable value to different applications, e.g., *data lineage* with metadata management, *data debugging* to trace root reason for data errors, *data trust* to quickly respond to business opportunities and regulatory challenges, and *data security* to place access control for data query based on the data lineage [11].

The use of data provenance is proposed in distributed systems to trace records through a dataflow (i.e., data pipeline), replay the dataflow on a subset of its original inputs and debug dataflows. To accomplish such tasks, one needs to keep track of the set of inputs to each operator, which are used to derive each of its outputs. For example, one can ascertain the quality of the data based on its ancestral data and derivations, backtrack sources of errors, allow automated re-enactment of derivations to update data.

For big data applications, data provenance is more critical with big lineage. Take Facebook–Cambridge Analytica data scandal as an example. It is reported University of Cambridge distributed the data from the personality quiz app "myPersonality" to hundreds of researchers via a website with insufficient security provisions, which made user data vulnerable to access for four years. More than 3 millions of personal information of Facebook users are shared without sufficient monitoring. The investigations by Facebook and the Information Commissioner's Office are trying to determine who accessed these data and what it was used for. However, as it was shared with so many different people, tracking everyone that has a copy and what they did with it proves very difficult. Similarly, personal data like user's activities and website click logs are used to train machine learning model for advertisements and recommendations. However, users are unaware of how and where the personal data are used in the real-world applications.

Even though the use of data lineage approaches is a promising way for big data management, the process is rather complex. The challenges include scalability of the lineage store, fault tolerance of the lineage store, accurate capture of lineage for black box operators and many others. These challenges must be addressed carefully and trade-offs between them need to be evaluated to make a realistic design for data lineage capture. Current approaches supporting data lineage in big data systems, e.g., RAMP [16], Newt [13], Titan [14], Smoke [18] do not meet the requirement of real applications due to following limitations: (1) They fail to design a uniform data pipeline involving various tasks like machine learning, SQL, OLAP, and streaming. Their solutions cannot be applied to specific applications, and track data processing across Spark Jobs, e.g., [14]. (2) They do not support a unified data model and query language to manage the data lineage. More importantly, they do not have a common platform to manage large scale data lineage, suffering scalability issues. (3) Finally, when a data pipeline incurs multiple data storage sources, existing approaches fail, and cannot understand the lineage across data platforms. These limitations prevent efficient support for large scale interactive lineage management.

In this work, we introduce SAC, a system that enables interactive data provenance in Apache Spark [22]. SAC can track information about the existing data continuously. It has a visual representation of data lineage, effortlessly raising the maturity of data governance programs. The contributions of this work are listed as below.

- We propose a new data entity model to represent data processing unit in a data pipeline, and provide a query language and an interactive interface to understand the captured data lineage clearly.
- We build data lineage for the entire data processing pipeline across different data sources and jobs. The system is able to capture data lineage for different computation jobs, e.g., Spark machine learning, Spark streaming and Spark SQL jobs, as well as different data storage, e.g., Apache HBase, Apache Hive, HDFS, Apache Kafka.
- We build the system to track Spark applications without

2375-026X/19/$31.00 ©2019 IEEE
DOI 10.1109/ICDE.2019.00215

IEEE computer society

additional users' inputs, which captures the lineage automatically. The easy-to-use and plug-in implementation is able to support Spark ecosystem without modifications of Spark internal.
- We demonstrate the system in a real production environment, since it natively supports existing system without any modification of Spark internal. The released version is already adopted by multiple companies and end-users.

The rest of this paper is organized as follows. Section II lists the related work. Section III provides the data model and programming interface of the system. Section IV gives the system architecture and design principles. Finally, experimental studies are given in Section V, and Section VI concludes the work.

II. Related Work

An industry data workflow or pipeline consists of multiple stages. Take a data pipeline for processing Internet of things (IoT) as an example. The IoT data is at first congested and preprocessed via Apache Kafka [6] to distribute input streaming data to different processing units. For example, a Spark streaming job is used to extract, transform, and load (ETL) the input data into some physical storage (e.g., HDFS [4], HBase [3], and Hive [5]) for further analysis. At the same time, OLAP jobs based on Spark SQL [9] analyze the cleaned data. Further more, a machine learning model based on Spark ML is trained from the historical data. Considering the ecosystem of Hadoop [2], this situation becomes more complex if multiple third-party packages (e.g., distributed matrix computation [21], and spatial data processing [19], [20], [15]) based on Spark are deployed in the same cluster.

Data provenance and lineage tracking is widely studied in database and big data platforms [11], [16], [8], [7], [12]. Data lineage is widely used in various applications from data security to data debugging. More recently, different systems are proposed to track data lineage along the big data ecosystems. For example, RAMP [16] is built to track jobs for Hadoop and Hyracks computation platform; Newt [13] is used to track data lineage for MapReduce job; Titan [14] debugs a Spark job based on the extension of Spark RDD; Inspector Gadget (IG) [17] monitors Apache Pig jobs; Arthur [10] debugs Spark job by re-executing the jobs. Although addressing several challenges of data lineage tracking for big data, there are still some limitations to overcome when tracking data lineage.

At first, most of exising work mainly consider one type of jobs or specific data storage. Newt [13] tracks lineage for MapReduce job, and Titan [14] builds lineage for Spark RDD. Such a design is easy to implement but ignore the data storage characteristics. For example, an end-user at first reads data from an HBase table, then writes the query results into a Hive table. We can imagine there would be a lineage to track such transformations from an HBase table to a Hive table. The metadata of HBase and Hive need to be recorded when the system computes the jobs. However, most approaches do not track data lineage for the entire data processing pipeline. In addition, most of existing systems fail to provide a user-friendly interface to visualize the data lineage, as well as an query interface to manage the data lineage graph. In this work, we build a system to track data lineage for different Spark jobs in a processing pipeline. The users are able to interact with the tracking lineage via a web interface or a command-line terminal. The built system is already deployed in real production environments, and is able to track data lineage for more than 100GB per day.

III. Data Model

In this section, we introduce the data model to manage data lineage.

A. Coarse vs. Fine-Grained Provenance

Ikeda et al. [13] modify the MapReduce and Hadoop file system for fine-grained granularity provenance tracking. However, fine-grained provenance needs to modify the Hadoop system internal greatly, and suffers from expensive overhead to record the lineage. This impedes these systems to be used in real-world applications for expensive system maintenance and performance consideration. In this work, we mainly focus on coarse-grained granularity provenance tracking for the following reasons. At first, the built data provenance system should not incur expensive overhead to track the lineage, and slow down the computation job. Secondly, the distributed computation clusters are already deployed into some production environment, we cannot replace these systems to enforce data governance. Therefore, we mainly focus on coarse-grained provenance for general data storage like Hive table, HDFS file, HBase table, streaming data, machine learning model, and machine learning data processing data pipelines.

B. Data Model

At first, we formalize a type system to define the data source and related data processing tasks. This type system is used to build specific structures for storing different types of metadata, where the metadata is used to define the data storage and the relationships between them. In this work, Apache Atlas [1] is developed to define a model for the metadata objects which users want to manage. The model is composed of definitions called "type". Instances of "type" is called "entitie". Entities represent the actual metadata objects being managed.

Consider a table named T, and users wants to model this table through the built type system and record its metadata. The schema of Table T contains the following information: database name, table name, column name, partition-by or cluster-by specific columns, storage properties. Therefore, we at first define an entity to record the metadata of table columns T_c and data storage properties T_s. Then the metadata of Table T is defined via T_c and T_s. The generated data model is called *Spark_Table* type, while this table is created from Spark SQL. The instance of type *Spark_Table* is used to record the created tables in Spark SQL.

Formally, an attribute a is used to define the concept related to the type system. That is, one attribute is used to

formalize a type. For example, Attribute a is defined with the following information: {*name*: string, *typeName*: string, *isOptional*: boolean, *isIndexable*: boolean, *isUnique*: boolean, *cardinality*: enum}. More specifically, *name* represents the name of the attribute; *dataTypeName* is the metatype name of the attribute (native, collection or composite); *isIndexable* indicates if this property should be indexed on, such that look-ups can be performed using the attribute value as a predicate and can be performed efficiently. *isUnique* stands for any attribute with a true value. This flag is treated as a primary key to distinguish this entity from other entities. *Cardinality* indicates whether this attribute is required, optional, or could be multi-valued.

Next, a type (says t) in Atlas is a definition on how a particular type of metadata object is stored and accessed. A type t represents one or a collection of attributes that define the properties for the metadata object. Users with a development background will recognize the similarity of a type to a "class" definition of object-oriented programming languages, or a "table schema" of relational databases. Naturally, an instance of type t is called entity e. An entity is used to define the created table T. We demonstrate the introduced concepts (e.g., attribute a, type t, and entity e) based on follow running example.

C. Data Entity

The first kind of entity is called data entity. The data entity could be (A) SQL related tables, e.g., Spark table and database, Hive table and database, (B) HBase data storage and HDFS Data storage, (C) streaming data source (e.g., Kafka streaming), (D) machine learning model and data pipeline. For example, a Spark database entity includes following attributes: {*name*, *description*, *locationUri*, *properties*, *owner*}. Therefore, the metadata is recorded into the system once a Spark table is created. More details of entity design for data storage will be introduced in a white paper of SAC.

D. Data Processing Entity

Besides the data entity, data operation entity is used to demonstrate the data processing job itself. Thus, data processing entity is an association for a combination of 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. Each unique actor is represented by its own association table. An association itself looks like $\{i, A, o\}$, where i is the set of inputs to the actor A, and o is set of outputs produced by the actor. Associations or data processing entities are the basic units of data lineage. Individual associations are later clubbed together to construct the entire history of transformations that are applied to the data. In this work, we consider a Spark application as one process entity. In Spark, the highest-level unit of computation is an application. A Spark application can be used for a single batch job, an interactive session with multiple jobs, or a long-lived server continually satisfying requests. Thus, one Spark application could be related to different kinds of jobs, e.g., ETL job, machine learning job, streaming processing job, and SQL job. Then, job information about one Spark application (e.g., job types, job running time, and job related query plan) is stored for the processing entity.

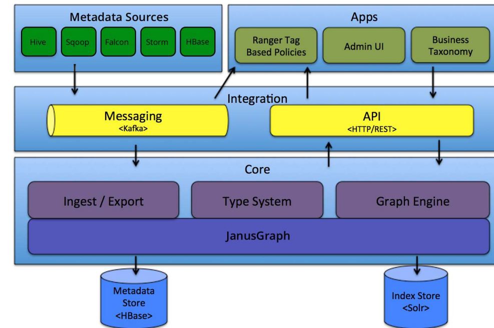

Fig. 1: Architecture of Atlas

IV. System Architecture

Lazy lineage collection typically captures only coarse-grained lineage at runtime. These systems introduce low overhead due to the small amount of lineage they capture. On the other hand, active collection systems capture the entire lineage of the dataflow at runtime. The kind of lineage they capture may be coarse-grained or fine-grained, but they do not require any further computations on the dataflow after its execution. Active fine-grained lineage collection systems incur higher capture overhead than lazy collection systems. In this work, we adopt lazy and coarse-grained lineage collection mechanism, since one big data or machine learning job cannot guarantee one hundred percentage success ratio in most cases. For example, the OOM (out of memory exception) usually breaks a Spark job in the middle.

In order to catch up coarse-grained lineage, SAC builds the connection between Apache Spark and Apache Atlas based on the Spark event tracker and interface provided by Atlas, respectively.

A. Apach Atlas

Figure 1* presents the system framework of Apache Atlas. Atlas adopts a graph engine to store the ingested metadata. This approach provides great flexibility and enables efficient handling of rich relationships between the metadata objects. The graph engine is powerful for graph query over the tracked data lineage. The underlying data storage is HBase for data entities. Solor is used to index the entities for fast entity query processing. Atlas provides two external APIs: the Rest API and Apache Kafka streaming. In terms of messaging system Kafka, it is useful for communicating metadata objects to Atlas, and for consuming metadata change events from Atlas.Atlas uses Apache Kafka as a notification server for communication between hooks and downstream consumers of metadata notification events. Events are written by the hooks and Atlas to different Kafka topics.

*https://atlas.apache.org/Architecture.html

B. Spark Event Tracking

Based on the hook mechanism of Atlas, we monitor the Spark job processing via Spark event tracker. Spark event tracker is one kind of callback function, which is used to send Spark job related information back to Atlas. Specifically, *SparkListener* is a mechanism in Spark to intercept events from the Spark scheduler, that are emitted over the course of execution of a Spark application. SAC adopts multiple event trackers to catch up the status of Spark Jobs. Consider a Spark SQL related job, we adopt the Spark SQL listener to monitor the status of Spark SQL DDL or Spark DML. In terms of a Spark ML job, we develop a new Spark ML job listener to register the status of an ML job. It could record the Spark ML pipeline and related data model. More design details about trackers like fault tolerance, security token management, and efficient event buffering will be presented in the white paper later.

V. Demonstration plan

SAC is open source and the technical preview version is already released[†]. In this demonstration, we will showcase how SAC tracks lineage with an end-to-end solution.

A. Demonstration Scenarios

We would demonstrate SAC to track lineage for different types of data processing jobs and data storage. The data processing jobs consist of Spark SQL, Spark ML, Spark streaming job, and the data storage contains local disk (with different types of data format like Parquet, Arvo, and Json), HDFS, Hive table, HBase and Apache Kafka. When a Spark application starts, SAC will transparently track the execution plan of submitted SQL/DF transformations, parsing the plan and creating related entities in Atlas.

In addition, we demonstrate how SAC performs data lineage tracking in secure and non-secure cluster setup. For a non-secure environment, we at first include the built jar of SAC to be accessible from the Spark Driver, then configure *spark.extraListeners* and *spark.sql.queryExecutionListeners* accordingly. Next, the related Spark job status can be captured silently.

For a secure environment, we have to enable the secure connection via secure Kafka client API. Thus, we at first shift to use Kafka client API by configuring *atlas.client.type=kafka* in *atlas-application.properties*. Meanwhile, this feature depends on Kakfa delegation token mechanism for performance consideration. Thus, we make sure keytab (a.keytab) is accessible from the Spark Driver. When running in the cluster mode, we also need to distribute this keytab to each executor as well.

B. Query Interface

SAC provides the RESTful API to manage the lineage via command line. In addition, a web-based interface is built to visualize and query the data lineage interactively. For data security consideration, Apache Ranger is plugged in and enforced to protect data security based on users' privilege.

[†]https://github.com/hortonworks-spark/spark-atlas-connector

VI. Conclusions

We have presented SAC, a data lineage tracking system. It extends Apache Altas and supports different types of data processing job lineage tracking. SAC is able to plug into the existing Spark computation cluster seamlessly, and provides efficient query interface to manage the captured data lineage. The initial version of SAC is already deployed into real-world production environments, and the feedback demonstrate the built system achieve efficient data lineage tracking for more than 100GB per day. Part of this work is supported by the National Natural Science Foundation of China (Grant No. 61802364), cooperated with CNIC, Chinese Academy of Science.

References

[1] "Atlas," https://altas.apache.org/.
[2] "Hadoop," http://hadoop.apache.org/.
[3] "Hbase," https://hbase.apache.org/.
[4] "Hdfs," https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.
[5] "Hive," https://hive.apache.org/.
[6] "Kafka," https://kafka.apache.org/.
[7] Y. Amsterdamer, S. B. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen, "Putting lipstick on pig: Enabling database-style workflow provenance," *Proc. VLDB Endow.*
[8] M. K. Anand, S. Bowers, and B. Ludäscher, "Techniques for efficiently querying scientific workflow provenance graphs," in *EDBT '10*. New York, NY, USA: ACM, 2010.
[9] M. Armbrust, R. S. Xin, C. Lian, Y. Huai, D. Liu, J. K. Bradley, X. Meng, T. Kaftan, M. J. Franklin, A. Ghodsi, and M. Zaharia, "Spark SQL: relational data processing in spark," in *SIGMOD 2015*, 2015, pp. 1383–1394.
[10] A. Dave, M. Zaharia, S. Shenker, and I. Stoica, "Arthur: Rich post-facto debugging for production analytics applications."
[11] B. Glavic, "Big data provenance: Challenges and implications for benchmarking," in *Revised Selected Papers of the First Workshop on Specifying Big Data Benchmarks - Volume 8163*. New York, NY, USA: Springer-Verlag New York, Inc., 2014, pp. 72–80.
[12] T. J. Green and V. Tannen, "The semiring framework for database provenance," in *PODS '17*, 2017.
[13] R. Ikeda, H. Park, and J. Widom, "Provenance for generalized map and reduce workflows," in *In CIDR*, 2011, pp. 273–283.
[14] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie, "Titian: Data provenance support in spark," *Proc. VLDB Endow.*, vol. 9, no. 3, pp. 216–227, Nov. 2015.
[15] Y. Liang, Y. Yu, M. Tang, C. Li, W. Yang, W. Xu, and R. Zheng, "Optimizing generalized linear models with billions of variables," in *CIKM '18*, 2018.
[16] D. Logothetis, S. De, and K. Yocum, "Scalable lineage capture for debugging disc analytics," in *Proceedings of the 4th Annual Symposium on Cloud Computing*, ser. SOCC '13. New York, NY, USA: ACM, 2013, pp. 17:1–17:15.
[17] C. Olston and B. Reed, "Inspector gadget: A framework for custom monitoring and debugging of distributed dataflows," in *SIGMOD '11*. ACM, 2011.
[18] F. Psallidas and E. Wu, "Smoke: Fine-grained lineage at interactive speed," *Proc. VLDB Endow.*
[19] M. Tang, Y. Yu, Q. M. Malluhi, M. Ouzzani, and W. G. Aref, "Locationspark: A distributed in-memory data management system for big spatial data," *PVLDB*, 2016.
[20] W. Yang, M. Tang, Y. Yu, Y. Liang, and B. Saha, "Shc: Distributed query processing for non-relational data store," in *ICDE 2018*, April 2018, pp. 1465–1476.
[21] Y. Yu, M. Tang, W. G. Aref, Q. M. Malluhi, M. M. Abbas, and M. Ouzzani, "In-memory distributed matrix computation processing and optimization," in *ICDE'17*, 2017, pp. 1047–1058.
[22] M. Zaharia, M. Chowdhury, T. Das, A. Dave, J. Ma, M. McCauly, M. J. Franklin, S. Shenker, and I. Stoica, "Resilient distributed datasets: A fault-tolerant abstraction for in-memory cluster computing," in *NSDI'12*. San Jose, CA: USENIX, 2012, pp. 15–28.