2018 IEEE 14th International Conference on e-Science

Big Provenance Stream Processing for Data Intensive Computations

Isuru Suriarachchi, Sachith Withana and Beth A. Plale
School of Informatics, Computing and Engineering, Indiana University
Bloomington, IN, USA
Email: {isuriara,swithana,plale}@indiana.edu

***Abstract*—In the business and research landscape of today, data analysis consumes public and proprietary data from numerous sources, and utilizes any one or more of popular data-parallel frameworks such as Hadoop, Spark and Flink. In the Data Lake setting these frameworks co-exist. Our earlier work has shown that data provenance in Data Lakes can aid with both traceability and management. The sheer volume of fine-grained provenance generated in a multi-framework application motivates the need for on-the-fly provenance processing. We introduce a new parallel stream processing algorithm that reduces fine-grained provenance while preserving backward and forward provenance. The algorithm is resilient to provenance events arriving out-of-order. It is evaluated using several strategies for partitioning a provenance stream. The evaluation shows that the parallel algorithm performs well in processing out-of-order provenance streams, with good scalability and accuracy.**

***Index Terms*—Big Data, Big Provenance, Stream Processing**

I. Introduction

With the pervasive availability of Internet infrastructure and data services, a commercial or research organization can easily collect and continuously integrate large volumes of data from both external and internal sources (*e.g.*, social media, click streams, sensors, IoT devices, server logs). It has become routine for Internet scale organizations to run continuous analysis of public and proprietary data for insight, decision making, and predictive forecasts; the cornerstone of Big Data. These analysis computations are called *large scale, data-intensive computations* or data-intensive computations (DIC) for short [43]. DICs share in common their utilization of data-parallel processing frameworks such as Hadoop [6], Spark [42] and Flink [8].

Data Lakes [19] [39] is a paradigm for Big Data management and analysis where data flows into a data lake as streams flow into a physical lake. The strength of the Data Lake is that it is schema-on-read where data transformations to a particular schema are deferred to time of use [36]. That is, data are ingested in a raw form then converted to a particular schema immediately prior to use. This applies to structured, semi-structured, and unstructured data; continuously arriving or batch. The environment of the Data Lake, with multiple simultaneously and long-running DICs, motivates our work.

Our objective is to track the lifecycle of data products - existence and movement - in a Data Lake. That is, to be able to produce backward and forward provenance for the data products flowing through DICs in a Data Lake. It has been noted elsewhere that data residing in a Data Lake, has what can be, a confusing lifecycle [5] [30] that motivates traceability of the data products. Suppose sensitive information (*e.g.*, consumer credit card data) has seeped into a dataset that had been subjected to multiple transformations. Cleansing requires tracing forward in time and cleaning data products that are derived by the sensitive data.

Our earlier research [36] [37] demonstrates the feasibility of data provenance traceability for a DIC in a Data Lake by a reference architecture to collect, manage and analyze provenance data captured from analysis tools integrated with the Data Lake. The solution is evaluated using the Komadu provenance management system [38]. However storing and querying large volumes of provenance generated by DICs is expensive. This paper extends the earlier work in recognition of the need for on-the-fly processing of provenance.

There is substantial evidence accumulated through years of provenance research that fine-grained provenance from large scale computations can be extremely voluminous [36] [21] [41]. Managing large volumes of provenance data for use is identified as a challenge in Big Data provenance and named as the *Big Provenance problem* [41][17]. RAMP [21] and HadoopProv [4] are two efforts from the literature to capture and analyze provenance from DICs. While both systems are limited to Hadoop MapReduce, they store full provenance into HDFS and face the Big Provenance problem.

Our approach is to use stateful, one-pass parallel stream processing to summarize a full provenance stream on-the-fly by preserving backward provenance and forward provenance. It does this through maintaining state within each parallel stream consumer. Intuitively, *backward provenance* begins with an output and traces backwards in time to the subset of input data on which the output depends. *Forward provenance* begins with an input and traces forward in time to the subset of output data elements derived by it.

Backward and forward provenance on static, stored provenance graphs have been calculated using recursive traversal through a graph [20], and maintenance of transitive closure tables [13]. Neither technique is suitable for our needs because of their high compute and storage overhead. So we develop *parallel-prov-stream*, a parallel stream algorithm for reducing full provenance on-the-fly by preserving backward and forward provenance that is scalable, order independent and one-pass only.

978-1-5386-9156-4/18/$31.00 ©2018 IEEE
DOI 10.1109/eScience.2018.00039

IEEE computer society

The stream processing approach is framed in the context of a DIC workflow where multiple DICs run concurrent with each other, are long lived, and are themselves distributed. Provenance is streamed to a single, standalone provenance stream processing system that is itself distributed. There is an assumption that each DIC executes in *batch mode*. Our algorithm processes the stream of provenance on-the-fly, but the batch assumption is used to correctly determine termination of the stream. The algorithm and processing approach is resilient to *event ordering*, by being able to accommodate events out-of-order within some time delta.

The final framing of the approach is its implicit treatment of *provenance identity*. The provenance system is built based on a log store abstraction that supports "topics" to which provenance events are published. Topics, and their uniqueness, is used to guarantee that provenance events are associated with the correct DIC from which they were generated. We assume that when a new DIC begins, it is configured to publish provenance to a unique topic ID in the distributed log store.

Related to our research is [9] where a dependency matrix is computed across input parameters and variable values from a stream of data provenance from an Agent Based Model. Our work both extends and complements [9] through application of provenance stream processing to large scale DICs.

The primary contribution of the paper is a parallel stream processing algorithm that reduces a stream of provenance from a DIC on-the-fly while preserving backward and forward provenance. The algorithm is resilient to provenance events out-of-order. It is evaluated using a streaming system built using the Kafka distributed log store [23] and the Flink streaming framework [8]. We evaluate three different partition strategies: horizontal, vertical, and random, to split a graph stream of provenance from a DIC into partitions to be processed in parallel.

The remainder of the paper is organized as follows: Section II defines backward and forward provenance in DICs and gives a model of the provenance graph stream. The model is illustrated through examples in Section III. The parallel streaming algorithm is detailed in Section IV, and implementation and system architecture are presented in Section V. The experimental evaluation is Section VI. Related work, Section VII, and future work, Section VIII, round out the paper.

II. Provenance model

A Data-Intensive Computation (DIC) consists of some number of functions and executes on a framework such as Hadoop or Spark.

Definition: *A data-intensive computation (DIC) is a sequence of n ordered functions F_1, F_2, ..., F_n which satisfy the following two conditions. D_k^i is input data and D_k^o is output data for function F_k. D_1^i is input data and D_n^o is output data for the DIC as a whole.*

$$F_k(D_k^i) = D_k^o; 1 \leq k \leq n$$

$$D_k^i = D_{k-1}^o; 1 < k \leq n$$

Each function F_k is executed in parallel on partitions of its input data satisfying the following condition. We call such an execution $f_{kj}(d_{kj}^i)$ a *function execution*.

$$F_k(D_k^i) = f_{k1}(d_{k1}^i) \cup f_{k2}(d_{k2}^i) ... \cup f_{kp}(d_{kp}^i)$$

In a given DIC, data flows through a sequence of functions where each function performs some action on its input data and produces some output which is fed into the next function. For example MapReduce uses functions: map, combine, and reduce while Spark and Flink has a broader set: map, filter, reduce, sortByKey, join, and etc. These frameworks assign multiple worker nodes to execute a function in parallel on different partitions of input data.

A DIC workflow may consist of one or more DICs depending on the number of steps in the workflow.

Definition: *A DIC workflow with input D^i is a set of m DICs C_1, C_2, .., C_m such that $C_k(D_k^i) = D_k^o; 1 \leq k \leq m$ where $D_k^i \subset \{D^i \cup D^*\}$, D^* is the union of outputs from already completed DICs in the workflow and the final output, $D^o \subset \cup_{k=1}^m D_k^o$.*

Some DICs within a workflow may execute in parallel and they may consume data from the inputs to the workflow and outputs from the other DICs. The final output of the workflow may consist of the outputs from one or more DICs. The input data to a DIC workflow often consists of differently typed data from different sources. Figure 1 shows a DIC workflow which consists of six DICs operating on four partitions of input data. Each DIC is a sequence of functions.

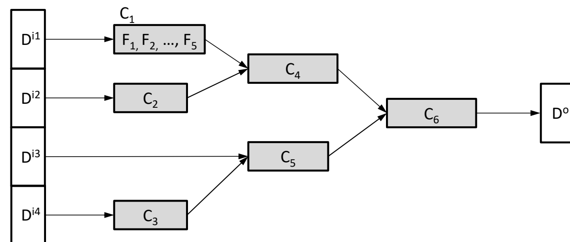


Fig. 1. DIC workflow made up of six DICs

A. Backward and Forward Provenance

Backward provenance and forward provenance define a minimal but sufficient type of provenance needed for several useful provenance analysis tasks [10], [9]. Whereas RAMP [29] [21] recursively defines provenance for any data element in MapReduce, we generalize the definition for any data element in a DIC.

Definition: *One-function backward provenance of output element o from function f_i of a DIC is the set E of intermediate elements contributing as input to f_i. Backward provenance of o is then the recursive union of the backward provenance for each $e \in E$. Recursivity terminates when all inputs e are elements in input data for the DIC.*

Definition: *One-function forward provenance of input element i to function f_j of a DIC is the set E of intermediate*

elements produced as output by f_j. Forward provenance of i is then the recursive union of the forward provenance for each $e \in E$. Recursivity terminates when all outputs e are elements in output data from the DIC.

Backward provenance then for an output or intermediate data element in a DIC is the subset of input data elements (to the DIC) on which it depends. Forward provenance for an input or intermediate data element in a DIC is the subset of output data elements (from the DIC) derived by it. The intermediate data is between functions in the context of a DIC.

Having established the data-intensive computation (DIC) as the basic building block of a DIC workflow, we can extend the above definitions to reason about backward and forward provenance for the DIC workflow as a whole. In a workflow, a dependency path between an input data element and an output data element may go through one or more DICs as illustrated in Figure 1. Backward provenance then for an output or intermediate data element in a DIC workflow is the subset of input data elements (to the workflow) on which it depends. Forward provenance for an input or intermediate data element in a DIC workflow is the subset of output data elements (from the workflow) derived by it. The intermediate data is between DICs in the context of a DIC workflow.

B. Provenance Streams

The two widely used provenance representation languages, OPM [28] and W3C PROV [27] which we use, both represent provenance as a directed acyclic graph. Provenance generated by each DIC in a DIC workflow corresponds to a single provenance graph.

Definition: *A Provenance Graph G = (V, E, A) is a directed, acyclic graph where a node (v ∈ V) is an activity, entity, or agent defined in W3C PROV, an edge (e = $\langle v_i, v_j \rangle$ where e ∈ E and v_i, v_j ∈ V) represents a relationship defined in W3C PROV directed from v_i to v_j and a set of attributes A(p) = $\{a_1, a_2, ...\}$ belongs to node or edge p.*

A provenance stream can be thought of as a serialization of a static provenance graph. A provenance stream for a DIC is created on-the-fly during execution of a DIC. Elements that grow a provenance graph on-the-fly correspond to provenance relationships (edges) being established (e.g., use of a particular data product). Frequently a node's existence is asserted upon its first use.

Definition: *A Provenance Stream S = $\{s_1, s_2, ..., s_n\}$ representing a Provenance Graph G = (V, E, A) is an append-only sequence of elements where an element s represents one of the following.*

1) $s \in E$
2) *s = $\langle P_m \rangle$ where m ∈ V or m ∈ E, m is already found in the stream before s and P_m ⊆ A(m)*

An element in a provenance stream is a provenance relationship asserted between two vertices or a set of attributes for either a vertex or an edge that has already appeared in the stream before the current element. Attributes are allowed for the edge elements when they are initially created. However in some situations attributes need to be added later. For example, when a function execution starts, the start time can be recorded as an attribute in the very first edge which uses the function. However the end time can only be added after the function execution has completed.

The provenance stream model defined in [9] allows only derivation relationships that are temporally ordered. Here we extend their definition to allow other relationships and accommodate events out-of-order.

Each DIC in a DIC workflow generates a separate provenance graph stream. We call a stream of raw provenance (before processing) a *full provenance graph stream*. We apply our parallel provenance stream processing algorithm on a full provenance graph stream generated by a single DIC to reduce the amount of provenance on-the-fly while preserving backward and forward provenance. Each reduced provenance stream is stored in a provenance repository for archiving and querying.

III. Big Provenance in DICs

Fine-grained provenance captured from DICs is useful for debugging and monitoring computations, for tracing the origins of derived data, and for tracing the derivation paths for input data. Figure 2 shows an example of a MapReduce DIC that counts hashtags in a set of tweets from an input file. When the job is executed, each line in the input file is fed into a map function which outputs $\langle hashtag, 1\rangle$ for each hashtag found in that line. Once completed, the reduce function calculates the total number of occurrences for each hashtag and produces the output file shown.

In order to analyze how each input was processed and how each output was generated, details on each function execution should be recorded including their input and output data products. Fine-grained provenance collected from both map and reduce functions can fulfill that requirement. Figure 3 shows the full provenance graph for the above example including inputs and outputs for all function executions. Backward provenance for output $c : 2$ and forward provenance for first tweet (input to M_1) are also illustrated in the same figure. As we defined above, backward and forward provenance for a DIC shows dependencies between the inputs to the first function and the outputs from the last function. Provenance of intermediate functions are used only to derive paths between inputs and outputs of the DIC.

Backward and forward provenance is useful in many different scenarios. For an example, if an increase in interest for a certain product is seen as a result of a twitter data analysis workflow, backward provenance can locate the subset of input tweets for further analysis like users' geographic distributions, age distributions etc. Forward provenance is useful in cases like tracing all output records which were derived by some corrupted records in the input.

Earlier efforts to capture and analyze provenance from DICs include RAMP [21], which uses a wrapper-based approach to extend Hadoop to capture provenance and HadoopProv [4], which modifies Hadoop instead of extending it, and does so to reduce the run-time overhead of capturing provenance. Both

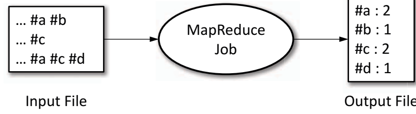


Fig. 2. MapReduce DIC for hashtag counting

solutions persist full provenance information into a storage system. When fine-grained provenance is collected from a DIC workflow, the amount of provenance generated can grow to amounts that challenge most storage solutions. Going beyond relational databases, few recent efforts tackle the volume in provenance stores through techniques utilizing distributed file systems [43], NoSQL stores [1], and graph databases [16].

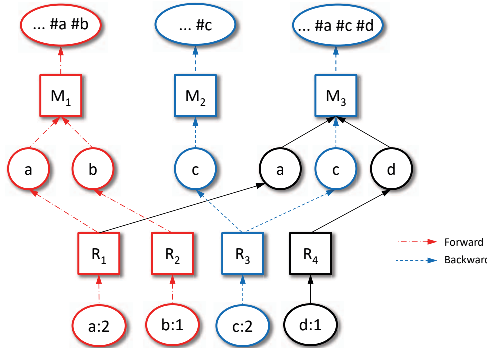


Fig. 3. Full provenance in MapReduce illustrating forward provenance for first tweet and backward provenance for output c : 2. Direction of arrows follow W3C Prov convention for derivation and generation.

Irrespective of the techniques used, having to expand the size of the storage layer by multiple times just to store provenance is not realistic and efficient in most applications. Provenance queries frequently require extensive graph traversal: calculating backward/forward provenance, finding all paths through a given function, and checking whether a given output is dependent on a given input. Graph traversal queries are frequently supported using recursion, however, recursion is considered extremely slow and compute intensive [20] [24] for large graphs. Storing transitive closure tables for each node in a graph is another technique for faster graph traversal. But transitive closure tables consume significant space [24] and are computationally expensive.

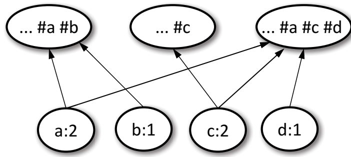


Fig. 4. Reduced backward and forward provenance

In this paper, we propose a stream processing approach that runs in near real-time to the application to reduce the volume of provenance from a DIC, saving unnecessary writes to storage. Our stream processing techniques are applied to a stream of full provenance from a DIC to derive a reduced provenance graph which only contains backward and forward dependency relationships between the inputs and outputs. Intuitively, provenance related to intermediate data products and function executions are removed in real-time and they are only used to maintain dependency paths between inputs and outputs. Figure 4 shows the reduced provenance graph which only contains backward and forward provenance for the example in Figure 2. For large scale DICs with multiple functions, this reduction in graph size helps with both storage and query efficiency.

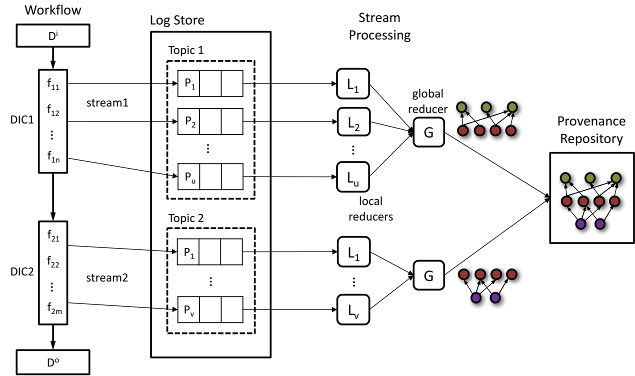


Fig. 5. Stream processing solution for a DIC workflow

Figure 5 shows the provenance stream processing solution for a DIC workflow which consists of multiple DICs. Provenance from each DIC in the workflow is considered as a separate stream. Each provenance stream is split into multiple partitions to achieve scalability through parallel stream processing. Each partition is processed by a local stream reducer and then the reduced result is periodically forwarded into a global reducer which provides the final provenance graph with backward and forward provenance. Reduced provenance graphs from all DICs in a workflow are stored and merged in a central provenance repository to build backward and forward provenance for the entire workflow.

Provenance queries are executed on reduced provenance graph for the DIC workflow which is finally stored in the provenance repository. As the depth and size of the graph is reduced by multiple times compared to full provenance, both storage cost and the query complexity is reduced by several factors. Downside of reducing provenance on-the-fly is that once reduced full provenance can not be recovered. As we present in Section VI, advantages of reduction out-weights the disadvantages.

IV. Parallel Provenance Stream Processing

The provenance stream definition allows all types of W3C PROV edges as elements in the stream. Depending on the type of analysis, certain provenance stream elements should be filtered out. When designing a streaming algorithm for backward and forward provenance, we must first identify

the set of edge types which can exist in a derivation path between two entities. Direct derivation between two entities is represented by a *wasDerivedFrom* edge. Data derivation is also indicated by a *used* edge and a *wasGeneratedBy* edge connected through an activity node; and a *hadMember* edge and a *wasDerivedFrom* edge connected through an entity node.

There are other W3C PROV edge types like *alternateOf*, *specializationOf* etc. too which may participate in derivation paths. However, for the purposes of this work in the context of DICs, we consider *wasDerivedFrom*, *used*, *wasGeneratedBy* and *hadMember* as the set of edge types that occur in a path between two entities. Other edge types we filter out. In addition, we further filter stream elements that add attributes to a node or an edge (second type in the definition) as those are not important for backward and forward provenance.

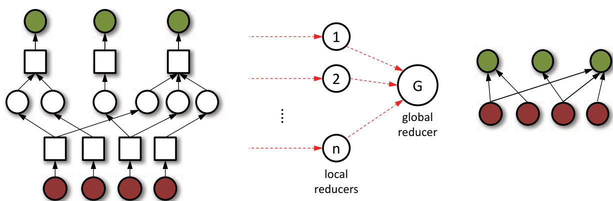


Fig. 6. Application of parallel stream processing on a partitioned stream of full provenance to produce a reduced provenance graph preserving backward and forward provenance

A. Parallel-prov-stream Algorithm

Our objective is an algorithm that can one-pass process provenance in parallel while adjusting for out-of-order events, and resulting in retention of backward and forward provenance. We illustrate this in action through Figure 6 which utilizes the full provenance graph given earlier in Figure 3. On the left is the full provenance graph for the computation which is streamed edge by edge as and when they are generated. Stream of full provenance is split into multiple partitions (using techniques in Section IV-B) and each partition is fed into a local reducer. On the right is the final reduced output from the global reducer which preserves backward and forward provenance.

Intermediate data items and edges are removed real-time by local and global stream processors. Both local and global processors maintain a state which contains the current set of reduced provenance edges. Each local stream processor filters out unnecessary edges first. New incoming edges are matched with the current local state to derive new dependencies by connecting them through common vertices. For example, if a local processor receives elements $\langle v_1, v_2 \rangle$, $\langle v_2, v_3 \rangle$ and $\langle v_5, v_6 \rangle$, it reduces the local state to $(\langle v_1, v_3 \rangle$, $\langle v_5, v_6 \rangle)$ by transitivity. Each local processor periodically flushes its reduced state into downstream global processor upon processing a fixed number (called *local batch size*) of stream elements. The global processor further reduces the state by merging compatible edges from different local processors and it also periodically flushes the reduced state upon processing a fixed number (called *global batch size*) of stream elements.

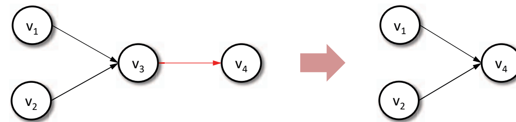


Fig. 7. Reduction through the source vertex of a new edge

Algorithm 1 gives our one-pass *parallel-prov-stream* algorithm which is used by both local and global processors. This algorithm maintains an internal state which contains the current most reduced set of edges. In other words, for any edge $\langle v_i, v_j \rangle$ in the state, there is no other edge whose destination vertex is v_i or source vertex is v_j. The algorithm maintains two HashMaps $sMap$ and $dMap$ for efficient access to edges in the state. HashMap $sMap$ is key-value pairs where key is a vertex id and value is a list of edges whose source is the same vertex id. HashMap $dMap$ is key-value pairs where key is a vertex id and value is a list of edges whose destination is the same vertex id. A given edge has two pointers from $sMap$ and $dMap$ based on its source vertex id and destination vertex id. For each new stream element or edge, the internal state is checked to find possible reductions. The algorithm uses the two HashMaps to access the edges with possible reductions in $O(1)$ time without scanning through the entire state.

Figure 7 shows how a reduction through the source vertex of a new edge happens (line number 7 to 12 in Algorithm 1). Edges $\langle v_1, v_3 \rangle$ and $\langle v_2, v_3 \rangle$ are already present in the local state and $\langle v_3, v_4 \rangle$ is the new edge. When the algorithm calculates the new state, $\langle v_1, v_3 \rangle$ and $\langle v_2, v_3 \rangle$ are added to the list of edges to be deleted ($eDel$), and $\langle v_1, v_4 \rangle$ and $\langle v_2, v_4 \rangle$ are added to the list of new edges to be added ($eAdd$).

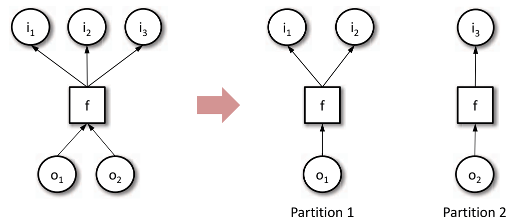


Fig. 8. Partitioning provenance from a function execution

There are two operations to process a single stream element (*addEdge*) and a group of stream elements together (*addEdgeGroup*). This algorithm does not depend on the order of provenance edges in the stream within the configured *batch size* as it keeps unreduced edges in the state for future reductions. Space complexity is bounded by the *local batch size* for local reduction and *global batch size* for global reduction.

B. Partitioning a Provenance Stream

In order to handle high rates of provenance from large scale DICs, our streaming system should be scalable. As

Algorithm 1 Provenance Stream Processing Algorithm

```
1: sMap                          ▷ map of reduced edge lists by source id
2: dMap                          ▷ map of reduced edge lists by destination id
3: procedure ADDEDGEGROUP(newEdges) ▷ newEdges: group of new stream elements
4:     list eDel                                        ▷ edges to delete
5:     list eAdd                                        ▷ edges to add
6:     for (ne in newEdges) do
7:         if (dMap.containsKey(ne.source)) then
8:             list edgesIntoSource = dMap.get(ne.source)
9:             for (e in edgesIntoSource) do
10:                eAdd.add(new Edge(e.source, ne.dest))
11:            end for
12:            eDel.addAll(edgesIntoSource)
13:        else if (sMap.containsKey(ne.dest)) then
14:            list edgesFromDest = sMap.get(ne.dest)
15:            for (e in edgesFromDest) do
16:                eAdd.add(new Edge(ne.source, e.dest))
17:            end for
18:            eDel.addAll(edgesFromDest)
19:        else
20:            INSERTEDGE(ne)           ▷ add new edge into state
21:        end if
22:    end for
23:    if (!eAdd.isEmpty()) then
24:        ADDEDGEGROUP(eAdd)           ▷ further reductions
25:    end if
26:    for (edge in eDel) do
27:        DELETEEDGE(edge)             ▷ delete edge from state
28:    end for
29: end procedure
30: procedure ADDEDGE(newEdge)       ▷ newEdge: new stream element
31:    list newEdges
32:    newEdges.add(newEdge)
33:    ADDEDGEGROUP(newEdges)
34: end procedure
35: procedure INSERTEDGE(edge)       ▷ inserts entries into sMap and dMap for given new edge
36: end procedure
37: procedure DELETEEDGE(edge)▷ delete entries in sMap and dMap for given edge
38: end procedure
```

shown in Figure 6, we split the stream of provenance into partitions and process them in parallel. Locally processed results from parallel stream processors are periodically merged to compute the current global state of backward and forward provenance. The partitioning strategy is extremely important for the efficiency of the system. Partitions should be created so that the local reductions are maximized.

We evaluate three different partitioning strategies. In order to not lose dependency paths during the partitioning process, we introduce a constraint that applies for all partitioning strategies: *all provenance edges generated during a single function execution must belong to the same partition in the stream.* A function in a DIC is executed many times on small fractions of input data and here we focus on such single execution of a function. Consider the scenario in Figure 8 where provenance from a single function execution is split into two partitions. The reduced output, then, from the first partition is ($\langle o_1, i_1\rangle$, $\langle o_1, i_2\rangle$) and from the second partition is ($\langle o_2, i_3\rangle$). These two outputs are received by the global reducer which outputs ($\langle o_1, i_1\rangle$, $\langle o_1, i_2\rangle$, $\langle o_2, i_3\rangle$). This output is incorrect as it lacks three valid dependency paths $\langle o_2, i_1\rangle$, $\langle o_2, i_2\rangle$ and $\langle o_1, i_3\rangle$. The constraint avoids this issue by sending all provenance edges from a single function execution to the same partition. In all three partitioning strategies described below, the smallest non-separable unit for partitioning the provenance stream is a collection of edges from a single function execution.

Horizontal Partitioning: Provenance elements from each function (all its function executions) in the DIC are directed to a separate partition. Each partition could only perform a one-step reduction which creates dependencies between the inputs and outputs of the relevant function. Horizontal partitioning for a stream of provenance from a DIC which consists of two functions is shown in Figure 9 in which provenance elements from functions f^1 and f^2 will be processed by separate local stream processors. Uneven load distribution among partitions can be expected with horizontal partitioning as different functions deal with different sizes of input data.

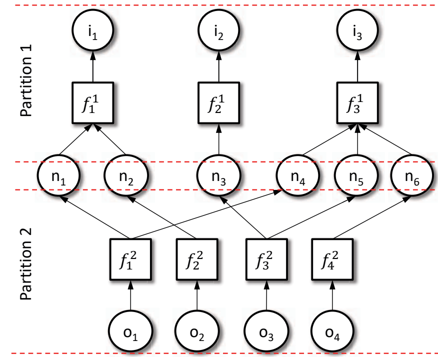


Fig. 9. Horizontal partitioning

Vertical Partitioning: Provenance stream is partitioned vertically along the derivation paths between inputs and outputs. Figure 10 shows the vertical partitioning for the same example in Figure 9. The idea is to preserve derivation paths within partitions as much as possible and maximize local reduction. DIC frameworks like Hadoop and Spark consider data locality as a major factor when scheduling functions on slave nodes. For a group of cluster nodes located close to each other, there is high chance that higher percentage of functions processing the data stored on them are executed within themselves. Therefore, as a vertical partition strategy, we propose to partition the stream based on the node which generated each stream element. The cluster of nodes is partitioned based on the locality and provenance stream elements from each cluster

partition creates a separate vertical partition in the provenance stream.

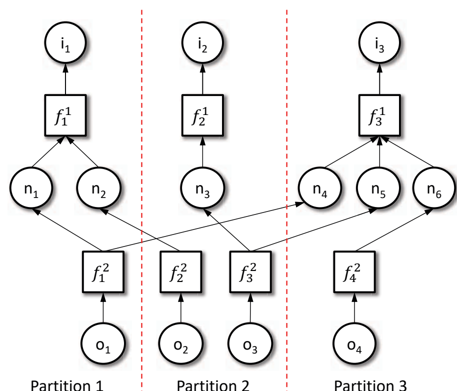
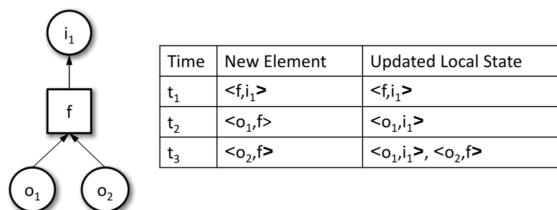


Fig. 10. Vertical partitioning

Random Partitioning: Randomly distributes provenance from function executions among parallel local stream reducers. Degree of local reduction depends on the percentage of provenance from nearby function executions which goes into the same partition. When the number of partitions increases, performance of random partitioning decreases as the probability of reducible edges falling into the same partition decreases. If the number of partitions is small random partitioning may perform better than horizontal partitioning.

C. Early Elimination Problem

In our *parallel-prov-stream* algorithm, one vertex is permanently removed during each reduction. When edges $\langle v_i, v_j \rangle$ and $\langle v_j, v_k \rangle$ are reduced to $\langle v_i, v_k \rangle$, vertex v_j is removed and no longer available for further reductions. This leads to incorrect results if v_j participates in other edges which have not been received by the processor yet. We call this as the *early elimination problem*. Consider the scenario in Figure 11 in which a function uses a single input and generates two outputs. At time t_2, the local state is reduced to $(\langle o_1, i_1 \rangle)$ by removing f. When $\langle o_2, f \rangle$ arrives at t_3, there is no way to derive its dependency on i_1. We evaluate two strategies to avoid this problem.



Time	New Element	Updated Local State
t_1	<f,i_1>	<f,i_1>
t_2	<o_1,f>	<o_1,i_1>
t_3	<o_2,f>	<o_1,i_1>, <o_2,f>

Fig. 11. Early elimination example

Grouping: When multiple data items are used or generated by a function, the provenance collector sends all usage or generation edges as a single group of elements in the stream. The *parallel-prov-stream* algorithm processes the group together through *addEdgeGroup* operation which makes sure that the vertices are deleted only after considering all elements in the group for reductions.

Sliding Window: Each stream processor maintains a sliding window which retains a configurable but limited number of past stream elements. Both the internal state and the sliding window is checked (an extension to above algorithm) at the arrival of each element to compute the new local state. If a certain deleted element in the local state is not also found in the sliding window, dependencies will be lost and the final result will suffer in accuracy.

V. Parallel Provenance Stream Implementation

We use a workflow consisting of two batch processing DICs to process Twitter data and apply our parallel provenance stream processing technique on provenance generated by the DICs. As shown in Figure 12, a Twitter client is used to collect tweets through the Twitter public streaming API and store in HDFS over a period of time. For each tweet, the client captures the Twitter handle of the author, time, language and the full message and writes a record into an HDFS file. First DIC which is implemented using Hadoop (v2.8.1) counts the occurrences of each hash tag in the full Twitter dataset and writes the results into a new HDFS file. The second DIC in the workflow is implemented using Spark (v2.2.1) and it produces aggregated tweet counts according to categories (sports, movies, politics etc).

We implement our *parallel-prov-stream* algorithm on top of the Flink Streaming framework (v1.3.2) [8] since Flink provides support for stateful stream processing while producing high throughput and low latency. We employ the Kafka (v0.11.0.1) [23] distributed log store to persist the provenance streams generated from DICs in the workflow and to handle partitioning. Kafka retains stream elements for a configurable period of time and controls the data rate going into the streaming system. Flink provides built-in Kafka connectors which can be configured to pull stream elements from Kafka partitions into Flink consumers.

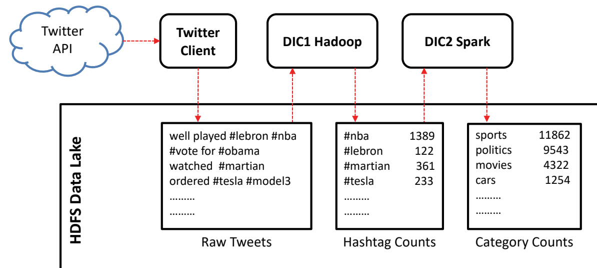


Fig. 12. DIC workflow to categorize hash tags in twitter data

The streaming solution is applied for provenance streams from each DIC in the workflow. Figure 13 shows the overall architecture of the system for our DIC workflow. The functions in Hadoop and Spark jobs are instrumented to capture provenance in W3C PROV JSON format; the Kafka

producer API is used to write streams into Kafka. There are different approaches [21] [4] to capture provenance depending on the system and trade-offs [35] associated with them. Our streaming solution is independent of the provenance capture method.

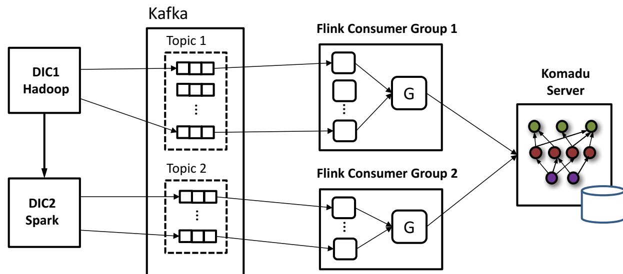


Fig. 13. Provenance stream processing architecture

Horizontal partitioning is done by assigning a separate partition for each function in a DIC and vertical partitioning is done by assigning a separate partition for a subset of nodes in the Hadoop or Spark cluster. Each DIC is assigned a separate Kafka topic and each partition in the provenance stream maps to a separate partition in the Kafka topic. Both local and global processors in Flink implement the *parallel-prov-stream* algorithm. Each local steam processor consumes a single partition from Kafka and performs local reduction. Local reducers periodically emit the reduced results into the global reducer which further reduces the global state. If no new stream elements are received for a configurable time limit, the global reducer deduces end of stream is reached and completes the output reduced provenance graph. This reduced graph is then stored to Komadu's central provenance repository. When the DIC workflow continues to run, reduced provenance graphs from all DICs are stored in Komadu and merged together using unique identifiers assigned for data items. Backward and forward provenance for the entire workflow is obtained by merging reduced graphs from all DICs.

VI. Experimental Evaluation

The evaluation is three part: first we evaluate the accuracy of our provenance stream processing algorithm for out-of-order provenance streams. We use a simulator to produce out of order streams and measure the accuracy for both "Grouping" and "Sliding Window" methods. Second, we evaluate the scalability of our streaming solution under increased parallelism. Third, efficiency of the three partitioning strategies is evaluated by measuring the degree of local reduction.

Environment. Experiments are run on virtual machines from the Jetstream cloud environment [14]. We use medium size VMs which consist of 6 CPU cores of 2.5 GHz speed, 16 GB of RAM and 60 GB of disk space per instance. The Hadoop cluster consists of six nodes with one master and five slave nodes. Total of 10.1 GB twitter data was collected and stored on HDFS. Two nodes are allocated for the Kafka cluster where the master node runs the zookeeper instance and both nodes run Kafka brokers. Up to six nodes are used for Flink streaming cluster depending on the experiment where one node is always used as the master and all others acting as slaves. Another VM is allocated for Komadu which persists reduced provenance graphs coming out of the streaming system.

Workload. The workload is a DIC workflow composed of two DICs: *DIC1* runs on Hadoop and *DIC2* runs on Spark, as shown in Figure 13. Each DIC produces a separate provenance stream and each is processed in isolation. The provenance results, once reduced to backward and forward provenance for both DICs, are brought together in Komadu [38] to build workflow level provenance.

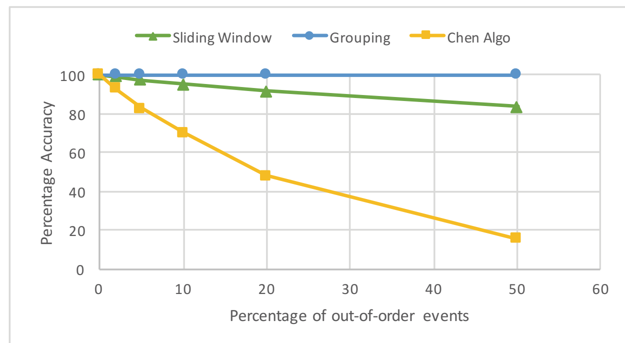


Fig. 14. Percentage accuracy of backward and forward provenance results against the percentage of out-of-order elements for "Grouping" and "Sliding Window" methods of our algorithm and Chen algorithm

In the first experiment, we measure the accuracy of our *parallel-prov-stream* algorithm for out-of-order provenance streams and compare results with [9]. Accuracy is measured by calculating the percentage of correct backward and forward provenance relationships exist in the output provenance graph. The algorithm in [9] is not designed to handle out-of-order stream elements. Therefore, the intention of this experiment is not to show that our algorithm performs better. But we use their algorithm as a base case to show the importance of handling out-of-order elements for accuracy.

As [9] is not a parallel algorithm, the provenance stream from *DIC1* is considered as a single partition and consumed by only one stream processor in this experiment. Same experiment is executed against both algorithms to measure the accuracy of the final results against varying out-of-order levels. As we need to vary the percentage of out-of-order elements in the stream, we developed a simulator tool which produces streams with required levels of ordering errors, consuming a recorded file with correctly ordered provenance elements (from *DIC1*). A subset of 1.1 GB input data was used for this experiment and it generated 2.86 GB of provenance.

We measure the accuracy of both "Grouping" and "Sliding Window" (window size set to 1000 elements) approaches of avoiding early elimination problem. According to results shown in Figure 14, all three algorithms provide 100% accuracy for perfectly ordered streams (0% out-of-order elements). As the fraction of out-of-order elements increases, our

algorithm remains 100% accurate with "Grouping" method while "Sliding Window" method showing some inaccuracy. This is expected as the *early elimination problem* can not be completely solved using "Sliding Window" method using a finite window size. Accuracy of Chen's algorithm reduces considerably as it does not handle out-of-order elements. We use "Grouping" method in all remaining experiments as it provides best accuracy.

The second experiment evaluates the scalability of the system through speedup: the proportional reduction in execution time for increasing parallelism against a fixed load. The dataset used is 10.1 GB Twitter dataset; the local batch size is set to 20000 and the global batch size is set to 100000. For each level of parallelism in Flink streaming job, the same number of Kafka partitions are created. Vertical partitioning is used as the partitioning strategy where provenance from each node in the Hadoop cluster contributes to a single partition.

Our approach shows sub-linear speedup with the increasing parallelism, as shown in Figure 15. Speedup deviates from ideal due to cross partition edges which do not allow 100% reduction within a partition and possible communication overheads in the streaming system with increasing parallelism. The system shows a maximum throughput of 6.044 MB/s per partition during this experiment.

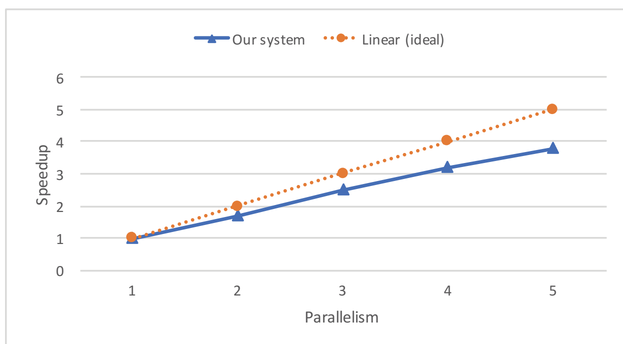


Fig. 15. Speedup (parallelism = 1 time/parallelism = n time) of the system against increasing parallelism. Input data size is fixed at 10.1 GB and local batch size is 20000.

In the third experiment, we evaluate the efficiency of horizontal, vertical and random partitioning strategies by measuring the degree of local reduction for the same computation under same range of local batch size. The stream partitioning strategy and local batch size are both determinant factors affecting the degree of local reduction. The degree of local reduction is measured by counting the total number of edges emitted by parallel local reducers towards the global reducer during the entire computation. We use the full 10.1 GB dataset for this experiment and run five Hadoop slaves and five Flink slaves.

Vertical partitioning performs best as it leads to maximum reduction along derivation paths, see Figure 16. Random partitioning also shows better reduction with increasing local batch size. This is due to the increasing probability of provenance from functions sharing same data products falling under the same partition. However the degree of reduction is less compared to vertical partitioning. Horizontal partitioning shows almost constant reduction with varying local batch size. This is expected as a horizontal partition can only have provenance from function executions of a single function. Reductions across multiple function executions are not possible and the chance of reductions does not increase with the batch size.

TABLE I
SIZE (IN GB) AND NUMBER OF EDGES IN PROVENANCE GRAPHS

	Input	Local out	Global out
Size (GB)	28.43	8.11	1.47
Num of edges (millions)	127.73	32.98	17.13

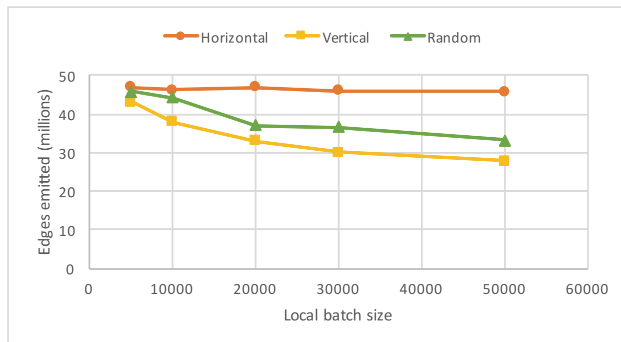


Fig. 16. Number of edges emitted by local reducers against the *local batch size* (in number of elements) for horizontal, vertical and random partitioning strategies

Table I shows the comparison of size (in GB) and number of edges (in millions) among input, local output and global output provenance graphs generated for the full twitter dataset of 10.1 GB with vertical partitioning and local batch size set to 20000. The results show that our parallel stream processing solution leads to a size reduction ratio of 19.34 and edge reduction ratio of 7.46 between the input and output provenance graphs while preserving backward and forward provenance.

VII. RELATED WORK

Provenance capture, query and visualization on traditional scientific workflows is a well studied area. Early systems like Chimera [15] and MyGrid [44] and provenance protocols such as Groth *et al.* [18] focus on capturing coarse-grained provenance from scientific workflows running on grids. Karma [34] and its predecessor, Komadu [38], are standalone provenance repositories; independent of any workflow system and ingesting events on topic-based channels.

Provenance from Big Data processing frameworks pose challenges in storage, scalability, and querying [41] [17]. Wang *et al.* [12] captures provenance in MapReduce workflows by integrating Hadoop into Kepler and using provenance capabilities of Kepler. RAMP [21] extends Hadoop to capture provenance by propagating input identifiers through the computation. They have built wrappers for Hadoop which automatically record provenance when a job is executed. HadoopProv

[4] modifies Hadoop to reduce provenance capturing overhead. Both RAMP and HadoopProv capture fine-grained provenance which include intermediate data as well. They persist full provenance into HDFS files and are capable of supporting both backward and forward provenance between inputs and outputs. Titian [22] is a modified version of Spark which automatically captures provenance from any function applied on a dataset. However all these systems suffer the issues of handling and querying Big Provenance as they store full provenance for offline processing.

There have been multiple attempts to address the challenges of the sheer volume of provenance through techniques utilizing distributed file systems [43], NoSQL stores [1], and graph databases [16]. However the cost of storage and querying [20] [24] could still be challenging when applied in the context of DICs.

As a provenance stream is frequently represented as a graph, graph streaming techniques are applicable in provenance stream analysis. Application of static graph analysis methods on graph streams is considered a difficult problem specially when the graph is directed [25]. Algorithms for problems like triangle count [7] and page rank [33] have been studied. There are few studies on graph stream clustering [2] and mining [11] as well.

Vijayakumar *et al.* [40] develops techniques to capture coarse-grained provenance among data streams whereas [26], [31], [32] studies fine-grained provenance capturing. Sansrimahachai *et al.* [31] treat provenance captured from streaming data as a separate stream and apply on-the-fly queries on top of it.

To the best of our knowledge, our earlier work [9] is unique in using stream processing techniques to compute backward or forward provenance. It maintains a dependency matrix of input parameters and variable values in real time, and can process unlimited amount of data with limited memory. It has been applied to interpreting provenance emerging from Agent Based Models (ABMs). The research described here complements this earlier work along several dimensions. The earlier definition of a provenance stream includes only derivation relationships; a limitation in most applications. Too, the stream of provenance is ordered by time of generation. We lift both assumptions to support multiple provenance relationships, and be resilient to out of order events as DICs are inherently distributed so subject to network delays, lost packets, retransmits etc. Finally the algorithm in [9] is not executable in parallel and we address this limitation too.

VIII. Conclusion and Future Work

We propose a provenance stream processing algorithm that processes fine-grained provenance in parallel and on-the-fly to reduce the stream while preserving backward and forward provenance in support of large scale data-intensive computations. We demonstrate that our algorithm performs well for out-of-order provenance streams and scales well with increasing parallelism.

An important area of ongoing work is to lift the assumption that DICs are batch oriented and the provenance stream is finite. Infinite provenance streams continuously generated from stream processing DICs requires provenance queries to be executable in real-time on the streaming system instead of queried after the provenance is persisted to a provenance repository.

Watermarks [3] is an alternate solution to the *early elimination problem* that merits evaluation. In addition, we plan to extend our system to other real-time provenance stream analyses such as detecting slow nodes in a compute cluster and detecting missing intermediate data items in the future.

The provenance stream processing algorithm and associated tools can be found at the following Git repositories:

- https://github.com/Data-to-Insight-Center/streaming-prov
- https://github.com/Data-to-Insight-Center/komadu

Acknowledgment

This work is funded in part by a grant from the National Science Foundation under grant #0940824. Thanks to Peng Chen, Instagram for valuable feedback on this manuscript.

References

[1] J. Abraham, P. Brazier, A. Chebotko, J. Navarro, and A. Piazza. Distributed storage and querying techniques for a semantic web of scientific workflow provenance. In *2010 IEEE Int'l Conference on Services Computing*, pages 178–185, July 2010.

[2] C. C. Aggarwal, Y. Zhao, and P. S. Yu. On clustering graph streams. In *Proc SIAM Int'l Conference on Data Mining*, pages 478–489. SIAM, 2010.

[3] T. Akidau, A. Balikov, K. Bekiroğlu, S. Chernyak, J. Haberman, R. Lax, S. McVeety, D. Mills, P. Nordstrom, and S. Whittle. Millwheel: Fault-tolerant stream processing at internet scale. *Proc. VLDB Endow.*, 6(11):1033–1044, Aug. 2013.

[4] S. Akoush, R. Sohan, and A. Hopper. Hadoopprov: Towards provenance as a first class citizen in mapreduce. In *Proc 5th USENIX Workshop on the Theory and Practice of Provenance*, pages 11:1–11:4. USENIX Association, 2013.

[5] A. Alserafi, A. Abelló, O. Romero, and T. Calders. Towards information profiling: Data lake content metadata management. In *Proc. IEEE 16th Int'l Conference on Data Mining Workshops (ICDMW)*, pages 178–185. IEEE, 2016.

[6] ASF. Apache hadoop. Available: https://hadoop.apache.org.

[7] Z. Bar-Yossef, R. Kumar, and D. Sivakumar. Reductions in streaming algorithms, with an application to counting triangles in graphs. In *Proc 13th ACM-SIAM symposium on Discrete algorithms*, pages 623–632. Society for Industrial and Applied Mathematics, 2002.

[8] P. Carbone, A. Katsifodimos, S. Ewen, V. Markl, S. Haridi, and K. Tzoumas. Apache flink: Stream and batch processing in a single engine. *Bulletin IEEE Computer Society Technical Committee on Data Engineering*, 36(4), 2015.

[9] P. Chen, T. Evans, and B. Plale. Analysis of memory constrained live provenance. In M. Mattoso and B. Glavic, editors, *6th Int'l Provenance and Annotation Workshop: Provenance and Annotation of Data and Processes*, pages 42–54, Cham, 2016. Springer Int'l Publishing.

[10] J. Cheney, A. Ahmed, and U. a. Acar. Provenance as dependency analysis. *Mathematical. Structures in Comp. Sci.*, 21(6):1301–1337, Dec. 2011.

[11] G. Cormode and S. Muthukrishnan. Space efficient mining of multigraph streams. In *Proc 24th ACM SIGMOD-SIGACT-SIGART symposium on Principles of Database Systems*, pages 271–282. ACM, 2005.

[12] D. Crawl, J. Wang, and I. Altintas. Provenance for mapreduce-based data-intensive workflows. In *Proc 6th Workshop on Workflows in Support of Large-scale Science*, pages 21–30. ACM, 2011.

[13] G. Dong, L. Libkin, J. Su, and L. Wong. Maintaining transitive closure of graphs in sql. *Int'l Journal of Information Technology*, 51(1):46, 1999.

[14] J. Fischer, S. Tuecke, I. Foster, and C. A. Stewart. Jetstream: A distributed cloud infrastructure for underresourced higher education communities. In *Proceedings of the 1st Workshop on The Science of Cyberinfrastructure: Research, Experience, Applications and Models*, SCREAM '15, pages 53–61, New York, NY, USA, 2015. ACM.

[15] I. Foster, J. Vockler, M. Wilde, and Y. Zhao. Chimera: a virtual data system for representing, querying, and automating data derivation. In *14th Int'l Conference on Scientific and Statistical Database Management*, pages 37–46. 2002.

[16] A. Gehani and D. Tariq. Spade: support for provenance auditing in distributed environments. In *Proc 13th Int'l Middleware Conference*, pages 101–120. Springer-Verlag New York, Inc., 2012.

[17] 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*, pages 72–80. Springer-Verlag New York, Inc., 2014.

[18] P. Groth, M. Luck, and L. Moreau. A protocol for recording provenance in service-oriented grids. In *Proceedings of the 8th International Conference on Principles of Distributed Systems*, OPODIS'04, pages 124–139, Berlin, Heidelberg, 2005. Springer-Verlag.

[19] A. Halevy, F. Korn, N. F. Noy, C. Olston, N. Polyzotis, S. Roy, and S. E. Whang. Managing google's data lake: an overview of the goods system. *Data Engineering*, page 5, 2016.

[20] T. Heinis and G. Alonso. Efficient lineage tracking for scientific workflows. In *Proc. 2008 ACM SIGMOD Int'l conference on Management of data*, pages 1007–1018. ACM, 2008.

[21] R. Ikeda, H. Park, and J. Widom. Provenance for generalized map and reduce workflows. 2011.

[22] M. Interlandi, K. Shah, S. D. Tetali, M. Gulzar, S. Yoo, M. Kim, T. D. Millstein, and T. Condie. Titian: Data provenance support in spark. *PVLDB*, 9(3):216–227, 2015.

[23] J. Kreps, N. Narkhede, J. Rao, et al. Kafka: A distributed messaging system for log processing. In *Proc. NetDB*, pages 1–7, 2011.

[24] T. Malik, A. Gehani, D. Tariq, and F. Zaffar. *Sketching Distributed Data Provenance*, pages 85–107. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[25] A. McGregor. Graph stream algorithms: a survey. *ACM SIGMOD Record*, 43(1):9–20, 2014.

[26] A. Misra, M. Blount, A. Kementsietsidis, D. Sow, and M. Wang. Provenance and annotation of data and processes. chapter Advances and Challenges for Scalable Provenance in Stream Processing Systems, pages 253–265. Springer-Verlag, Berlin, Heidelberg, 2008.

[27] P. Missier, K. Belhajjame, and J. Cheney. The w3c prov family of specifications for modelling provenance metadata. In *Proc 16th Int'l Conference on Extending Database Technology*, pages 773–776. ACM, 2013.

[28] L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. V. den Bussche. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst.*, 27(6):743–756, June 2011.

[29] H. Park, R. Ikeda, and J. Widom. Ramp: A system for capturing and tracing provenance in mapreduce workflows. In *37th Int'l Conference on Very Large Data Bases (VLDB)*. Stanford InfoLab, August 2011.

[30] C. Quix, R. Hai, and I. Vatov. Metadata extraction and management in data lakes with gemms. *Complex Systems Informatics and Modeling Quarterly*, (9):67–83, 2016.

[31] W. Sansrimahachai, L. Moreau, and M. J. Weal. An on-the-fly provenance tracking mechanism for stream processing systems. In *2013 IEEE/ACIS 12th Int'l Conference on Computer and Information Science (ICIS)*, pages 475–481, June 2013.

[32] W. Sansrimahachai, M. J. Weal, and L. Moreau. Stream ancestor function: A mechanism for fine-grained provenance in stream processing systems. In *2012 Sixth Int'l Conference on Research Challenges in Information Science (RCIS)*, pages 1–12, May 2012.

[33] A. D. Sarma, S. Gollapudi, and R. Panigrahy. Estimating pagerank on graph streams. *Journal of ACM (JACM)*, 58(3):13, 2011.

[34] Y. L. Simmhan, B. Plale, and D. Gannon. Query capabilities of the karma provenance framework. *Concurrency and Computation: Practice and Experience*, 20(5):441–451, 2008.

[35] M. Stamatogiannakis, H. Kazmi, H. Sharif, R. Vermeulen, A. Gehani, H. Bos, and P. Groth. Trade-offs in automatic provenance capture. In *Proceedings of the 6th International Provenance and Annotation Workshop on Provenance and Annotation of Data and Processes - Volume 9672*, IPAW 2016, pages 29–41, Berlin, Heidelberg, 2016. Springer-Verlag.

[36] I. Suriarachchi and B. Plale. Crossing analytics systems: A case for integrated provenance in data lakes. In *Proc 2016 IEEE 12th Int'l Conference one-Science (e-Science)*, pages 349–354. IEEE, 2016.

[37] I. Suriarachchi and B. Plale. Provenance as essential infrastructure for data lakes. In *Int'l Provenance and Annotation Workshop*, pages 178–182. Springer, 2016.

[38] I. Suriarachchi, Q. Zhou, and B. Plale. Komadu: A capture and visualization system for scientific data provenance. *Journal of Open Research Software*, 3(1), 2015.

[39] I. Terrizzano, P. M. Schwarz, M. Roth, and J. E. Colino. Data wrangling: The challenging journey from the wild to the lake. In *CIDR*, 2015.

[40] N. N. Vijayakumar and B. Plale. Towards low overhead provenance tracking in near real-time stream filtering. In *Proc. 2006 Int'l Conference on Provenance and Annotation of Data*, IPAW'06, pages 46–54, Berlin, Heidelberg, 2006. Springer-Verlag.

[41] J. Wang, D. Crawl, S. Purawat, M. Nguyen, and I. Altintas. Big data provenance: Challenges, state of the art and opportunities. In *2015 IEEE Int'l Conference on Big Data (Big Data)*, pages 2509–2516, Oct 2015.

[42] M. Zaharia, M. Chowdhury, M. J. Franklin, S. Shenker, and I. Stoica. Spark: Cluster computing with working sets. In *Proc. 2Nd USENIX Conference on Hot Topics in Cloud Computing*, pages 10–10. USENIX Association, 2010.

[43] D. Zhao, C. Shou, T. Malik, and I. Raicu. Distributed data provenance for large-scale data-intensive computing. In *Proc 2013 IEEE Int'l Conference on Cluster Computing (CLUSTER)*, pages 1–8. IEEE, 2013.

[44] J. Zhao, C. Goble, R. Stevens, and S. Bechhofer. Semantically linking and browsing provenance logs for e-science. In *Semantics of a Networked World. Semantics for Grid Databases*, pages 158–176. Springer, 2004.