Analysis of Memory Constrained Live Provenance

preprint version: forthcoming in IPAW 2016

Peng Chen[1], Tom Evans[2], and Beth Plale[1]

[1] School of Informatics and Computing, Indiana University Bloomington
{chenpeng,plale}@indiana.edu
[2] Department of Geography, Indiana University Bloomington
evans@indiana.edu

Abstract. We conjecture that meaningful analysis of large-scale provenance can be preserved by analyzing provenance data in limited memory while the data is still in motion; that the provenance needs not be fully resident before analysis can occur. As a proof of concept, this paper defines a stream model for reasoning about provenance data in motion for Big Data provenance. We propose a novel streaming algorithm for the backward provenance query, and apply it to the live provenance captured from agent-based simulations. The performance test demonstrates high throughput, low latency and good scalability, in a distributed stream processing framework built on Apache Kafka and Spark Streaming.

Keywords: live data provenance, stream processing, agent-based model

1 Introduction

The traditional persistent approach that operates on static provenance is not suitable for continuously generating provenance data. Our earlier work [6] showed that data provenance enables deeper analysis of the internal dynamics of agent based models (ABMs), by exposing dynamics that were previously hidden inside what is effectively a black box. However, [6] further shows that vast and unwieldy amounts of provenance can be captured continuously from running simulations with even a modest tens of thousands of interacting components (agents). In this case it quickly becomes infeasible to store all of the provenance data, requiring reassessment and reinterpretation of analysis techniques to operate over *live provenance data*, that is, before it gets written to disk.

Tasks such as debugging and model calibration are refinement processes, requiring repeated runs. When a refinement process requires an experiment to either fail or finish, it can be very time consuming. Our earlier work demonstrated that provenance data is useful for both debugging [7] and model analysis [6], and since provenance captures the dependencies between input parameters and simulated results that do not match real data, it is suitable for model calibration as well. The challenge, however, is to overcome storing and wading through vast volumes of information to quickly isolate behavior of interest.

An approach to faster and more targeted intervention is to process provenance continuously as a stream of data. Algorithms under this model are constrained to processing a potentially unbounded stream in the order it arrives while using limited memory [3]. Earlier work on data streams [1, 3, 21] often modeled the stream as a sequence of timestamped events and generally assumed homogeneous streams that could be centrally processed. Provenance data lends itself to being modeled as a graph, and a general graph stream consists of undirected edges arriving in random-order [17]. Recent work on graph-based streams focuses on the semi-streaming model [12], in which the data stream algorithm is permitted $O(n\text{polylog}n)$ space, where n is the number of nodes in the graph. This space use is not well suited to voluminous provenance, nor does it meet the constraints of continuous processing for exclusive in-memory use.

In this paper, we distinguish a provenance stream from a general graph stream by emphasizing the temporal order in a provenance graph. From that we develop an algorithm for the backward provenance query on streaming data that has a space complexity limited by the maximum number of data values that the program can access at any given time during its execution. In an agent-based model this number is proportional to the number of declared variables. We extend our earlier tool [6] to automatically capture provenance streams from running NetLogo [28] simulations and store them into Apache Kafka [15]. We then implement our proposed algorithm on Apache Spark Streaming [30] to support the parameter readjustment and online debugging for agent-based model. The performance evaluation shows high throughput, low latency and good scalability.

The remainder of the paper is organized as follows: Section 2 discusses related work. Section 3 defines the stream model of dependency provenance. Section 4 introduces our framework that supports the capture and query of provenance streams. Section 5 presents the evaluation on a real-world environmental agent-based model. Section 6 concludes the paper and discusses future work.

2 Related Work

Research on stream provenance focuses on the provenance about data streams. It can be categorized with coarse-grained provenance methods that identify dependencies between streams or sets of streams [27, 26], and fine-grained methods that identify dependencies among individual stream elements [23, 10, 18, 22]. Sansrimahachai et al. [23] propose the Stream Ancestor Function – a reverse mapping function to express precise dependencies between input and output stream elements (fine-grained). Our study focuses on the continuous processing of large provenance data streams, a problem that has received less attention. The most closely related work is Sansrimahachai et al. [22] who track fine-grained provenance in stream processing systems through an on-the-fly provenance tracking service that performs provenance queries dynamically over streams of provenance assertions without requiring the assertions to be stored persistently. However, their focus is on provenance tracking by essentially pre-computing the query re-

sults at each stream operation and storing results into provenance assertions as the provenance-related property.

There is research on provenance collection that treats provenance data as continuously generating events. For example, Komadu [24] receives provenance events and attributes as XML messages in a separate standalone system and can infer relationships between events after their arrival; SPADEv2 [13] has reporters that transparently transform computational activity into provenance events. However, neither system models provenance events as a stream. In contrast, we present our early work on a stream model for provenance events. Our preliminary model only covers the dependencies between data products (analogous to the "Derivation and Revision" in W3C PROV [20]) and their temporal ordering. We demonstrate that this model is sufficient for the continuous backward provenance query.

Provenance can be represented as a DAG, and there is work on querying provenance graph databases [19, 25]. Our study focuses on the continuous querying of massive provenance data streams. McGregor [17] surveys algorithms for processing massive graphs as streams, which focus on the semi-streaming model of $O(n\text{polylog}n)$ space. There has been little research on the stream processing of graph queries, and the most closely related work is the stream processing of XPath queries [14] and SPARQL queries [4, 2]. XPath queries need to consider the relationships between XML messages (similar to graph edges), and SPARQL queries are performed on RDF graphs. However, these extended SPARQL languages are developed for specific goals such as to support semantic-based event processing and reasoning on abstractions, not to support typical graph analysis based on the node/path patterns. The same holds true for XPath queries.

3 Stream Model of Provenance Graph

An agent-based model (ABM) is a simulation of distributed decision-makers (agents) who interact through prescribed rules. We demonstrate in [6] that the dependency provenance in an ABM can explain certain results tracked to input data, and can yield insight into cause-effect relations among system behaviors. The concept of dependency provenance [9, 8] is based on the dependency analysis techniques used in program slicing, which is different from "where-provenance" and "data lineage", but similar to "how-provenance" or "why-provenance" [5] in that it identifies a data slice showing the input data relevant to the output data. In this paper, we focus on the dependency provenance that consists of the data products and their dependencies, which can be considered as analogous to the "Derivation and Revision" in W3C PROV [20].

We use the same mapping to W3C PROV as in [6] to express the dependency provenance in ABM. PROV models provenance as a static graph, but the provenance capture can be viewed as a process of appending node/edge to the graph in their generation order. While a general graph can be streamed into a sequence of undirected edges in random-order, a provenance graph could be represented

as a sequence of directed edges following the order of node/edge generation (see Fig. 1 for an example).

We propose a limited stream model of provenance that captures just a subset of provenance relationships and their ordering. We denote a dependency provenance graph as $G = (V, E, A)$, where $V = \{v_1, v_2, ..., v_n\}$ is the set of data products (nodes); $E = \{e_1, e_2, ..., e_m\}$ is the set of dependency relationships (edges), in which an edge $e = \langle v_i, v_j \rangle$ specifies that v_i depends on v_j; $A(v) = \{a_1, a_2, ...\}$ represents an arbitrary number of attributes of v.

Definition 1. *A stream of dependency provenance consists of a sequence of time ordered pairs $e = \langle v_i, v_j \rangle$:*

$$S = \{e_1, e_2, ..., e_n, e_{n+1}\} \tag{1}$$

where $e = \langle v_i, v_j \rangle$ is a dependency relationship from v_i to v_j in V, and $timestamp(e_n) < timestamp(e_{n+1})$.

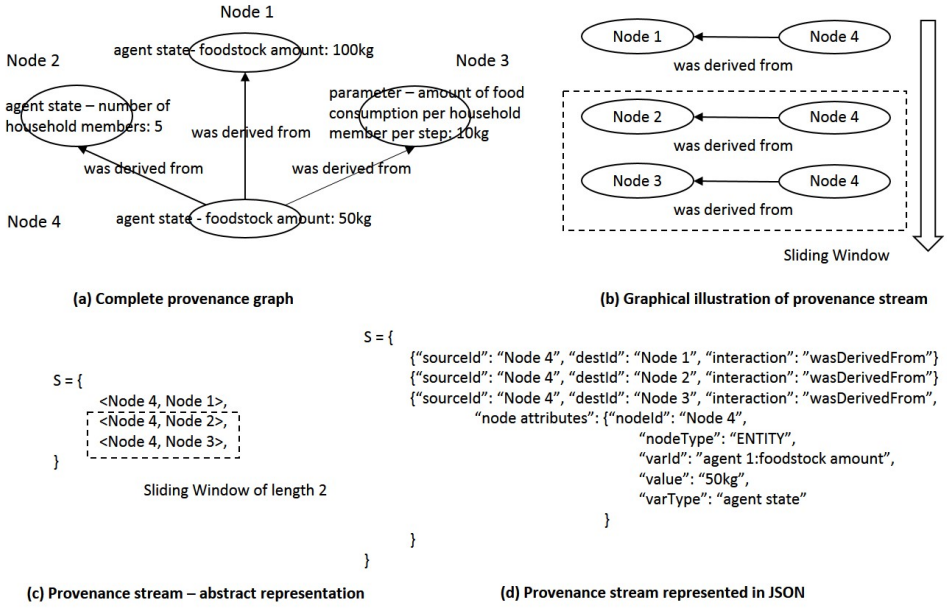

Fig. 1. Illustration of the provenance stream model.

If the temporal order of node/edge generation is properly preserved during capture, the provenance stream will follow a partial order specified below:

Property 1. For any two edges $e_l = \langle v_i, v_j \rangle$ and $e_m = \langle v_k, v_i \rangle$ that share a common node v_i, $timestamp(e_l) < timestamp(e_m)$

The provenance stream is append only and is potentially unbounded in size. Once an element of the stream has been processed it is discarded, and a query can only be evaluated over a limited internal state and/or the sliding window (with length w) of recently processed elements (at time t):

$$W = \{e_{t-w+1}, ..., e_t\} \tag{2}$$

There are two processing models in current distributed processing systems: the *record-at-a-time* processing model that receives and processes each individual record; and the batched processing model that treats streaming workloads as a series of batch jobs on small batches of streaming data. We implement our streaming algorithm in Spark Streaming [30] that is based on a batched processing model called *D-Streams*, and thus the provenance edges in S are received and processed during each batch interval (denoted by *bInterval*).

4 Provenance Stream Capture and Processing

We develop a scalable framework to support the capture and processing of live provenance streams generated from simulations running in NetLogo. Fig. 2 gives an overview of its two major components. The *provenance stream capture* component captures live provenance streams from agent-based simulations and stores them into a Kafka messaging cluster. The *provenance stream processing* component is built on a Spark Streaming cluster to support query and other analytical operations. The details are illustrated in the rest of the section.

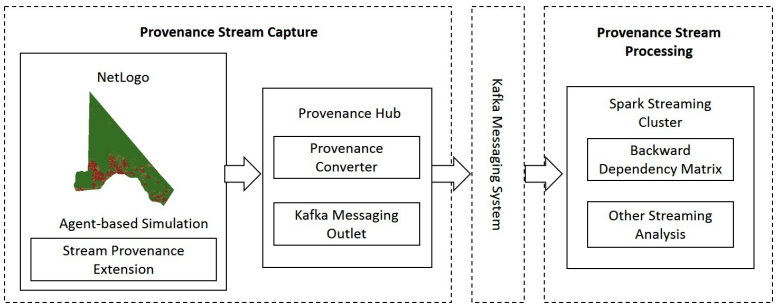

Fig. 2. Architecture of the provenance stream capture and processing framework.

4.1 Provenance Stream Capture

In [6] we capture the provenance traces of a NetLogo simulation through probes added to the model's source code, and develop a NetLogo extension that collects and saves the provenance traces to be processed offline. In this paper, we modify the NetLogo extension to send the provenance traces directly to a converter

that converts provenance traces into a live stream of provenance edges (in JSON format), which are then forwarded to the Kafka messaging system and processed in real-time. This provenance capture mechanism is illustrated in Fig. 3.

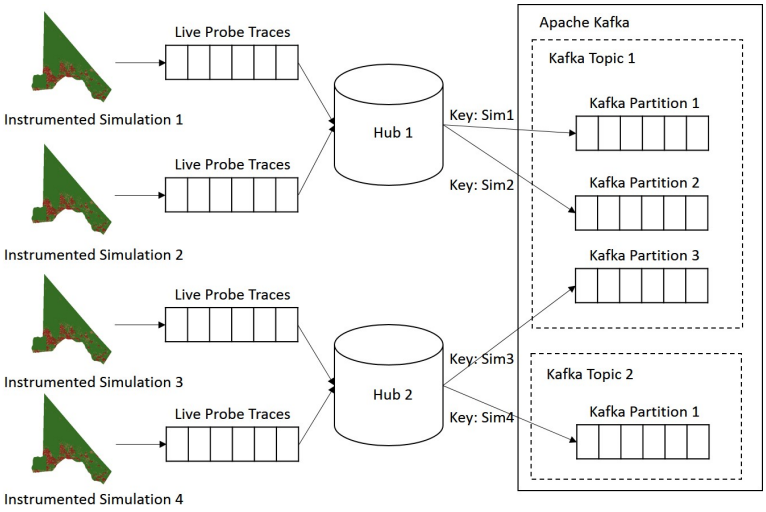

Fig. 3. Provenance capture from multiple running NetLogo simulations.

Note that each provenance hub uses multi-threading to receive probe traces from multiple simulations and send them to Apache Kafka [15], which is a distributed publish-subscribe messaging system that is designed to be fast, scalable, and durable. The provenance streams from different simulations can be separated by keys (uniquely formed by combining the hub ID with the stream ID). Each provenance hub can be configured either to send its streams into different partitions within one Kafka topic or into separate Kafka topics. This flexibility in organizing streams by topics and partitions can be used to improve the throughput and the level of parallelism of stream processing in Spark Streaming (see Section 5). For agent-based simulations distributed across multiple machines, we can deploy one or more provenance hubs on each machine.

4.2 Stream Processing Algorithm

Now we present our Backward Dependency Matrix (BDM) algorithm, which uses a dependency matrix to answer the backward provenance query for the most recent provenance nodes (*i.e.,* data products) in the stream. Given the temporal order defined in Section 3, we use a dynamic matrix to store and calculate the dependencies between all provenance nodes and the input/global parameters. For a newly arriving provenance node, the matrix is consulted to find the input/global parameters on which it depends. Fig. 4 illustrates the dynamic

matrix, with rows and columns added and removed on demand. The rows in the matrix correspond to provenance nodes (data products), and the columns correspond to input/global parameters. A cell of value 1 in the matrix means a backward dependency from its row to its column. Each time a new provenance edge $e = \langle v_i, v_j \rangle$ arrives, we extract the backward dependencies of v_j (value 1s in its row), and add them into the backward dependencies of v_i. The temporal order guarantees that all the backward dependencies of v_j arrive before e. In this way, we can calculate the backward dependencies for all provenance nodes, with the matrix size being potentially unbounded. However, under the constraints of our stream model, we can only use an internal state of limited size.

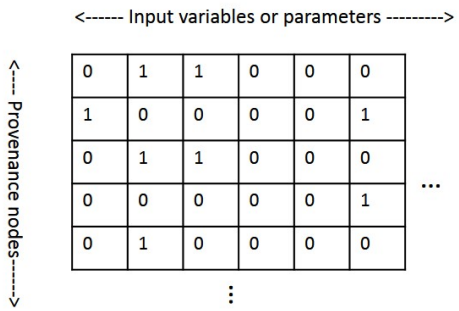

Fig. 4. Dynamic dependency matrix (0: dependent; 1: independent).

One observation on the agent based model in NetLogo, and in many other applications too, is that there exists only one instance (or value) of any variable at any moment – a universal value of a global variable, one copy of an agent variable within each agent, and one value of a local variable inside a function invocation – and we only need to query the backward dependencies for the current value of a variable. Thus the matrix only needs to keep the dependencies of the current variable instances, and those that could be used in future calculations.

In our stream model of provenance graph, each node is assigned with a node ID (unique within the stream) and a variable ID during the provenance capture (see Fig. 1(d)). The variable ID is formed by concatenating the context information and the declared name of that variable. For example, "global:variable 1", "agent 1:variable 2", and "procedure 1, level 1:variable 3" ("level" specifies the depth of recursion). Two provenance nodes with different node ID but same variable ID represent different values of the same variable. We keep dependencies of the most recent provenance node for each variable ID, except in the case that the most recent value of a variable depends on its earlier value – we use a cache matrix to temporarily store the dependencies of its earlier value. The algorithm is shown in Fig. 5. It has a space complexity of $O(N)$, where N is the number of variables declared in the model that is independent of the unbounded stream

length. The matrix *state.current* stores the dependencies of current nodes to input data that can be queried using the function **getBackwardProvenance**.

```
1: function updateState(element, state)  ▷ element: a newly arrived element (dependency edge) in the stream; state: the internal state with two dynamic matrices current and purge, and one HashMap varIdToNodeId
2:     sourceNode ← element.source
3:     destNode ← element.dest
4:     if state.varIdToNodeId.containsKey(sourceNode.varId) and state.varIdToNodeId.get(sourceNode.varId) != sourceNode.nodeId then
5:         remove all dependencies from state.current whose sources match sourceNode.varId
6:         cache removed dependencies in state.purge  ▷ older dependencies in state.purge whose sources match sourceNode.varId are also purged
7:     end if
8:     state.varIdToNodeId.put(sourceNode.varId, sourceNode.nodeId)
9:     if destNode.varId is an input/global variable then
10:        add new dependency sourceNode.varId ⇒ destNode.varId into state.current
11:    end if
12:    if destNode.varId == sourceNode.varId then
13:        inputVars ← getBackwardProvenance(destNode.varId, state.purge)
14:    else
15:        inputVars ← getBackwardProvenance(destNode.varId, state.current)
16:    end if
17:    for var in inputVars do
18:        add new dependency sourceNode.varId ⇒ var into state.current
19:    end for
20: end function

21: function getBackwardProvenance(varId, matrix)  ▷ varId: ID of the variable that we want to find its related input/global parameters; matrix: a dependency matrix.
22:    dependencies ← all dependencies in matrix whose sources match varId
23:    return destinations of dependencies
24: end function
```

Fig. 5. The BDM algorithm that maintains a dependency matrix to support the backward query on provenance stream.

4.3 Stream Processing Implementation

We implement the proposed algorithm inside a stream processing platform called Apache Spark Streaming [30]. Apache Spark [29] is a batch processing framework with the Spark Streaming extension to support continuous stream processing. We choose Spark Streaming because the provenance stream usually has a very

high speed (thousands of events per second) and Spark Streaming achieves higher throughput compared with other streaming platforms like Storm [16].

Spark Streaming uses a resilient distributed dataset (RDD) as the basic processing unit, which is a distributed collection of elements that can be operated on in parallel. There are two approaches to fetching messages from Kafka: the first is the traditional approach using Receivers and Kafka's high-level API to communicate with ZooKeeper; the second is a direct mode, introduced with Spark 1.3, which directly links and fetches data from Kafka brokers. We integrate Spark Streaming with Kafka using the latter approach for its better efficiency and simplified parallelism – it creates one RDD partition for each Kafka partition (*i.e.,* each provenance stream). Since Kafka implements the per-partition ordering and each RDD partition is processed by one task (thread) in Spark Streaming, the temporal order of node/edge generation we defined in Section 3 is preserved in both provenance capture and processing. Finally, the Kryo serialization is enabled for the BDM algorithm to reduce both the CPU and memory overhead caused by its internal state (*i.e.,* two dynamic matrices and one HashMap).

5 Experimental Evaluation

In evaluating the performance of our framework, we use a food security agent-based model we built for Monze District in Zambia, Africa [11]. In that model, 53,000 household agents make labor sharing and planting decisions biweekly based on a utility maximization approach within the context of local institutional regimes (*i.e.,* ward). The goal of the model is to understand the impact of climate change on adaptive change capacity among households. We use the source code analyzer [6] to add probes into the NetLogo code and the extended provenance extension to capture the live provenance stream while the simulation is running. The amount of raw provenance traces generated by running the model on one ward in the Monze District for one year is around 66MB, which is 357MB of provenance nodes/edges in JSON format. In our experiments, we run the model continuously for five growing seasons that generates about 1.7GB of provenance stream data to be processed in real-time. *Throughput* of our streaming framework is measured as below:

$$throughput = pSize * nSim/(nBatch * bInterval) \tag{3}$$

where $pSize$ is total amount of provenance data generated by one simulation (1.7GB in our evaluation), $nSim$ is number of simulations, $nBatch$ is number of batches taken to finish processing all data, and $bInterval$ is batch interval. *Latency* is measured as average total time to handle a batch (*i.e.,* sum of scheduling delay and processing time).

We run the experiments using the "Big Red II" supercomputer at Indiana University where each CPU-only compute node contains two 2.5GHz AMD Opteron 16-core CPUs and 64 GB of RAM, and is connected to a 40-Gb Infiniband network. In each experiment run, we use one node to run NetLogo (v5.2.0) simulations and our provenance hub, one to run the Kafka server and broker

(v0.8.2), and up to nine nodes of Spark Streaming (v1.5.1) standalone clusters – one master and eight slaves. The Kafka log directory and the Spark Streaming checkpoint directory are both placed in Big Red II's shared Data Capacitor II (DC2) Lustre file system, which is connected via a 56-Gb FDR InfiniBand network. By default, the Spark standalone cluster (v1.5.1) supports only a simple FIFO scheduler across applications. To allow multiple concurrent applications, we divide the resources by setting the maximum number of resources each application can use (*i.e.,* parameter "spark.cores.max"). Since the actual number of non-idle tasks is determined by the number of RDD partitions (a.k.a. the number of provenance streams), we also set "spark.default.parallelism" (the number of parallel tasks) equal to the number of provenance streams.

For a Spark Streaming application to be stable, the batch interval must be set so that the system can process data at the arrival rate. If the provenance arrival rate is consistently higher than the maximum processing speed, we can throttle it by slowing down the ABM simulation speed. However, in our evaluation, we do not throttle the arrival rate, instead we measure the maximum processing speed at different batch intervals, by enabling the "backpressure" feature in Spark Streaming – it automatically figures out the receiving rate so that the batch processing time is lower than the batch interval (see Fig. 6(a)).

We first measure the throughput and latency of the BDM algorithm running on a single-node Spark Streaming cluster, and the size of its internal state serialized in memory. To determine the maximum throughput under the condition of simply receiving stream elements, we also measure the Spark "collect" operation running alone. As can be seen from Fig. 6(b) and Fig. 6(c), our proposed BDM algorithm can achieve throughput as high as 10.8MB/s per stream (77% of the maximum throughput of 14MB/s), and latency as low as 1.5 seconds; when increasing the batch interval, the BDM algorithm will have higher throughput but also longer latency. In all scenarios, the maximum size of the internal state (an RDD cached in memory) is the same – 10.2MB.

Since our algorithm does not parallelize the processing within a provenance stream, we evaluate its scalability by measuring *Scaleup* – the ability to keep the same performance levels (response time) when both workload and compute resources increase proportionally. That is, we increase the number of provenance streams the same as the number of nodes in the Spark Streaming cluster.

There are two different approaches to sending provenance streams into processing: either creating a separate streaming application to process each provenance stream, or processing all provenance streams within one streaming application. The provenance hub organizes the provenance streams accordingly: one provenance stream per Kafka topic (the first approach), or one provenance stream per Kafka partition (the second approach). Fig. 6(d) shows the results. The second approach shows restricted scalability for the BDM algorithm for two reasons: the stateful operation "updateStateByKey()" maintains global states for all provenance streams; and the direct mode in Spark-Kafka integration has each Kafka partition occupying one CPU core per node for data receiving, thus limiting the number of streams one node can handle. While the first approach

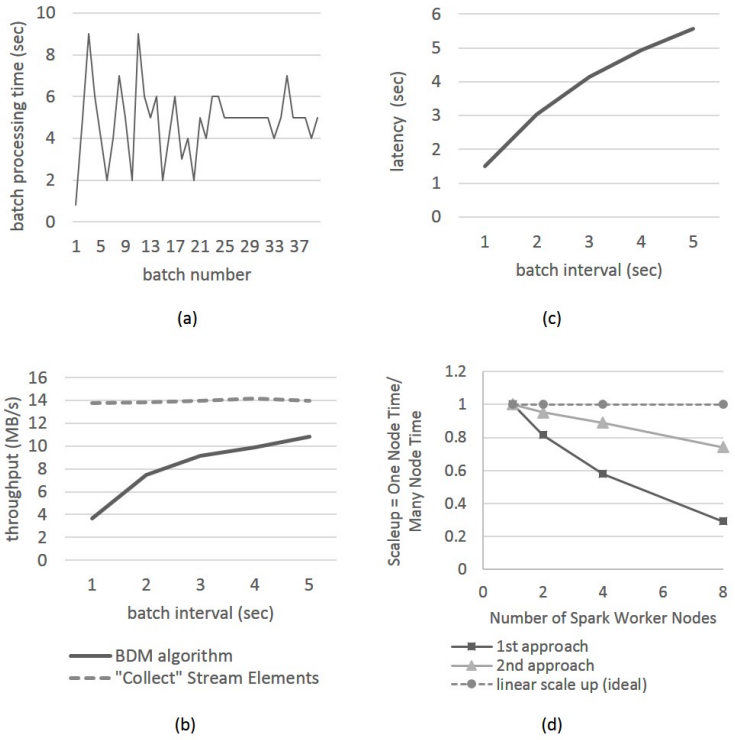

Fig. 6. (a) a BDM algorithm run with batch interval 5s and data receiving rate automatically controlled by Spark Streaming "backpressure" feature. Note that the trial and error at the beginning to find the right receiving rate; (b) throughput of running BDM algorithm at different batch intervals compared with maximum throughput when receiving provenance in Spark Streaming; (c) latency of running BDM algorithm at different batch intervals; (d) scalability test of BDM algorithm at 5s batch interval.

has better scalability, it complicates the joint-processing of multiple provenance streams, which can be supported naturally in one streaming application using the second approach.

6 Conclusion

This paper proposes a model of provenance streams and a framework that can automatically capture the live provenance stream from agent-based models. We propose a streaming (BDM) algorithm that supports backward provenance querying with limited space utilization. This can be used to calibrate the agent-based model – when observing a mismatch between real and simulated data during the simulation run, the BDM algorithm can return the relevant input parameters to be readjusted on-the-fly. The framework and the BDM algorithm

have been tested with a real-world environmental agent-based model that has thousands of household agents. The performance results show good throughput, latency and scalability.

Future work is to refine our definition of the stream model to include other types of provenance entities and relationships. In addition, how to handle out-of-order arrivals and how to parallelize the processing of one provenance stream remain open questions.

Acknowledgment

This work is funded in part by the National Science Foundation under award number 1360463.

References

1. Abadi, D.J., Ahmad, Y., Balazinska, et al.: The design of the Borealis stream processing engine. In: CIDR. vol. 5, pp. 277–289 (2005)
2. Anicic, D., Fodor, P., Rudolph, S., Stojanovic, N.: EP-SPARQL: a unified language for event processing and stream reasoning. In: WWW. pp. 635–644. ACM (2011)
3. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: PODS. pp. 1–16. ACM (2002)
4. Barbieri, D.F., Braga, D., Ceri, S., Della Valle, E., Grossniklaus, M.: C-SPARQL: SPARQL for continuous querying. In: WWW. pp. 1061–1062. ACM (2009)
5. Buneman, P., Khanna, S., Wang-Chiew, T.: Why and where: A characterization of data provenance. In: ICDT, pp. 316–330. Springer (2001)
6. Chen, P., Plale, B., Evans, T.: Dependency provenance in agent based modeling. In: eScience. pp. 180–187. IEEE (2013)
7. Chen, P., Plale, B.A.: Proverr: System level statistical fault diagnosis using dependency model. In: CCGrid. pp. 525–534. IEEE (2015)
8. Cheney, J.: Program slicing and data provenance. IEEE Data Eng. Bull. 30(4), 22–28 (2007)
9. Cheney, J., Ahmed, A., Acar, U.A.: Provenance as dependency analysis. In: Database Programming Languages. pp. 138–152. Springer (2007)
10. De Pauw, W., Leţia, M., Gedik, B., Andrade, H., Frenkiel, A., Pfeifer, M., Sow, D.: Visual debugging for stream processing applications. In: Runtime Verification. pp. 18–35. Springer (2010)
11. Evans, T., Plale, B., Attari, S.: WSC-Category 2 collaborative: Impacts of agricultural decision making and adaptive management on food security in africa (2014), National Science Foundation grant 1360463.
12. Feigenbaum, J., Kannan, S., McGregor, A., Suri, S., Zhang, J.: On graph problems in a semi-streaming model. Theoretical Computer Science 348(2), 207–216 (2005)
13. Gehani, A., Tariq, D.: SPADE: support for provenance auditing in distributed environments. In: Middleware. pp. 101–120 (2012)
14. Gupta, A.K., Suciu, D.: Stream processing of XPath queries with predicates. In: SIGMOD. pp. 419–430. ACM (2003)
15. Kreps, J., Narkhede, N., Rao, J., et al.: Kafka: A distributed messaging system for log processing. In: NetDB. pp. 1–7 (2011)

16. Lu, R., Wu, G., Xie, B., Hu, J.: Stream bench: Towards benchmarking modern distributed stream computing frameworks. In: UCC. pp. 69–78. IEEE (2014)
17. McGregor, A.: Graph stream algorithms: A survey. ACM SIGMOD Record 43(1), 9–20 (2014)
18. Misra, A., Blount, M., Kementsietsidis, A., Sow, D., Wang, M.: Advances and challenges for scalable provenance in stream processing systems. In: IPAW, pp. 253–265. Springer (2008)
19. Missier, P., Chirigati, F., Wei, Y., Koop, D., Dey, S.: Provenance storage, querying, and visualization in PBase. In: IPAW. vol. 8628, p. 239. Springer (2015)
20. Moreau, L., Missier, P., et al.: PROV-DM: The PROV Data Model. W3C Working Group Note 30 April 2013 (2013)
21. Plale, B., Schwan, K.: Dynamic querying of streaming data with the dQUOB system. TPDS 14(4), 422–432 (2003)
22. Sansrimahachai, W., Moreau, L., Weal, M.J.: An on-the-fly provenance tracking mechanism for stream processing systems. In: ICIS. pp. 475–481. IEEE (2013)
23. Sansrimahachai, W., Weal, M.J., Moreau, L.: Stream ancestor function: A mechanism for fine-grained provenance in stream processing systems. In: RCIS. pp. 1–12. IEEE (2012)
24. Suriarachchi, I., Zhou, Q., Plale, B.: Komadu: A capture and visualization system for scientific data provenance. Journal of Open Research Software 3(1) (2015)
25. Vicknair, C., Macias, M., Zhao, Z., Nan, X., Chen, Y., Wilkins, D.: A comparison of a graph database and a relational database: a data provenance perspective. In: ACMSE. p. 42. ACM (2010)
26. Vijayakumar, N., Plale, B.: Tracking stream provenance in complex event processing systems for workflow-driven computing. In: EDA-PS Workshop (2007)
27. Vijayakumar, N.N., Plale, B.: Towards low overhead provenance tracking in near real-time stream filtering. In: IPAW, pp. 46–54. Springer (2006)
28. Wilensky, U.: Netlogo (1999), `http://ccl.northwestern.edu/netlogo/`
29. Zaharia, M., Chowdhury, M., Franklin, M.J., Shenker, S., Stoica, I.: Spark: cluster computing with working sets. In: HotCloud. vol. 10, p. 10. USENIX (2010)
30. Zaharia, M., Das, T., Li, H., Shenker, S., Stoica, I.: Discretized streams: an efficient and fault-tolerant model for stream processing on large clusters. In: HotCloud. pp. 10–10. USENIX (2012)