Computing Location-Based Lineage from Workflow Specifications to Optimize Provenance Queries

Saumen Dey¹(✉), Sven Köhler¹, Shawn Bowers², and Bertram Ludäscher¹

¹ Department of Computer Science, University of California, Davis, Davis, USA
scdey@ucdavis.edu

² Department of Computer Science, Gonzaga University, Spokane, USA

Abstract. We present a location-based approach for executing provenance lineage queries that significantly reduces query execution cost without incurring additional storage costs. The key idea of our approach is to exploit the fact that provenance graphs resemble the workflow graphs that generated them and that many workflow computation models assume workflow steps have statically defined data consumption-production (i.e., data input-output) rates. We describe a new lineage computation technique that uses the structure of workflow specifications together with consumption-production rates to pre-compute (i.e., to forecast) the access paths of all dependent data items prior to workflow execution. We also present experimental results showing that our approach can significantly out perform traditional data lineage query techniques.

1 Introduction

Scientific workflow systems are increasingly used to automate data processing, analysis, and visualization steps [1]. These systems typically capture the processing history (i.e., the provenance) of all steps involved in a workflow run and store this information as a provenance graph [2, 3]. Provenance graphs can be used for a number of purposes including: (i) to help explain how input data is processed to produce output data products; (ii) to help debug workflow designs by identifying processes responsible for workflow failure and detecting workflow steps that were affected; and (iii) to help in the reproduction of data products, e.g., by recording the steps involved in a workflow run (along with their corresponding parameter settings).

Each of these examples require the ability to determine how a data product (or workflow step) depends on input data (or other workflow steps), e.g., by posing queries over provenance graphs. In these cases, provenance queries return subgraphs of the given provenance graph [4], where the subgraph is often referred to as the *lineage* of the data products in question. Answering such queries requires recursion, making lineage queries potentially expensive to execute [5]. In particular, if E is the set of edges in the provenance graph, these queries may require as many as $|E|$ recursive steps (i.e., traversals of dependency edges).

A better approach is to use semi-naïve evaluation [6], where the number of traversals is bounded by the diameter k of the provenance graph with $k < |E|$ in typical cases. An alternative to employing recursion is to compute and store the transitive closure of edges in the provenance graph [4, 5]. Because the transitive closure can be computed once and reused for all lineage queries over the graph, the time complexity required to compute the closure is often not a concern (since the cost can be aggregated). Using the transitive closure, if V is set of nodes in the provenance graph, the time to evaluate a lineage query is $\mathcal{O}(\log|V|)$ with storage cost $\mathcal{O}(|V|^2)$. Thus, for large provenance graphs, the recursive approach is space efficient, but not time efficient, whereas computing transitive closures is time efficient, but not space efficient. In this paper, we propose a new technique called *location-based lineage* for answering provenance queries that is both time and space efficient.

Fig. 1. An example workflow consisting of actors (rectangles), data containers (circles), and data flow edges annotated with consumption-production rates.

The main idea behind location-based lineage is to exploit the fact that a workflow specification provides a blue print (or “schema”) for the provenance graphs that they generate. The constraints imposed on provenance graphs by a workflow specification arise from both the structure of the workflow as well as the underlying computational model used by a workflow engine during workflow execution. As a simple example, consider the workflow specification in Fig. 1, where U through Z denote dataflow channels and A through D denote processing steps (i.e., actors). Based on the structure of the workflow, a data product on channel X (output by an invocation of B) may be dependent on a data product (input by the invocation of B) on channel Y , but cannot depend on a data product, e.g., on channel W . The constraints imposed by workflow computation models define the general order in which actors can be invoked as well as the number of data items that can be consumed and produced by each actor invocation. For instance, many workflow systems model actors as simple function calls that take a fixed number of arguments, i.e., input values, and return a fixed number of output values (e.g., VisTrails [7]). The synchronous dataflow (SDF) computation model [8] extends this by allowing workflow designers to specify the number of data items an actor needs on each input channel for the actor to be invoked, and the number of outputs produced on each channel by an invocation. Data items are buffered on channels until the needed number of items are received by an

actor.¹ Many of the scientific workflows developed in Kepler use the SDF model of computation (e.g., see [9,10]).

Location-based lineage uses the structure of the workflow graph together with the data consumption and production rates to precompute data dependency information prior to workflow execution. In particular, we provide an algorithm to compute the location of data consumed and produced by actors within channels statically (before workflow execution) and show how this information can be used to more efficiently answer lineage queries.

Paper Organization. Section 2 describes the general workflow, computation, and provenance models we assume for location-based lineage. Section 3 presents our approach for computing lineage queries for various workflow patterns and actor types. Section 4 describes the overall algorithm for computing location-based lineage. Section 5 explains the experiments we performed to validate our technique and analyzes the results. In Sect. 6, we discuss the recent efforts toward finding efficient lineage computation and finally in Sect. 7, we conclude with future directions.

2 Preliminaries: Workflow, Computation, and Provenance Models

Here we briefly describe the assumptions made concerning workflow graphs, provenance graphs, and workflow execution in our location-based lineage approach.

Workflow Model. We assume a workflow specification $W = (V, E)$ can be represented as a directed graph whose nodes $V = A \cup C$ are partitioned into *actors* A and *containers* C (e.g., see Fig. 1). Actors represent computational entities that can be executed (i.e., invoked). Each invocation of an actor can consume and produce data *tokens* representing either primitive or structured values or references to external data products (e.g., a file). Containers represent buffers (often implemented as FIFO queues) that can hold data tokens during communication between actors. The edges of a workflow graph $E = In \cup Out$ are either *input* edges $In \subseteq C \times A$ or *output* edges $Out \subseteq A \times C$. Actors can consume tokens from one or more containers and can produce tokens on one or more containers. Additionally, we assume input and output edges are annotated with token *consumption* and *production rates*, respectively. A consumption rate is a positive integer that specifies the number of input tokens needed to invoke an actor, and similarly, a production rate is a positive integer that specifies the number of output tokens generated by one invocation of an actor.

Computation Model. We make similar assumptions concerning workflow execution as used in the synchronous dataflow (SDF) model. In particular, actors can be invoked when their required number of input tokens on each channel

¹ Petri net based models, although not typically used for scientific workflow systems, also have similar constraints represented through edge multiplicities.

become available. Figure 1 is an example SDF workflow in which actor A can be invoked when u_c tokens in container U are available resulting in v_p and w_p tokens being output to containers V and W , respectively. For many actors (e.g., those representing simple function calls) the consumption-production rates will be 1 for each input and output. We also make a distinction between stateful and stateless actors. In particular, a stateful actor maintains one or more data tokens across its invocations within a workflow run and uses these tokens (i.e., the state) to compute output values. We consider two variants of stateful actors: (1) an invocation consumes all tokens that one or more of its previous invocations received, and (2) each invocation maintains a constant number of tokens that were consumed by its most recent invocation. Finally, to enable lineage queries based on specific data values we assume that as a workflow is executed, the contents of each container are persisted.

Provenance Model. We assume provenance graphs that generally follow the Open Provenance Model [11] in which provenance information can be represented as a directed graph $P = (V, E)$ whose nodes $V = D \cup I$ represent either data tokens D or actor invocations I and whose edges $E = Used \cup GenBy$ are either used edges $Used \subseteq I \times D$ or generated-by edges $GenBy \subseteq D \times I$. An used edge $(i, d_1) \in E$ implies that an invocation i consumed token d_1 as input, while a generated-by edge $(d_2, i) \in E$ implies that a token d_2 was output by i . In this case we say that d_2 depended on d_1 (i.e., d_2 is part of d_1 's lineage). The complete set of data tokens, used, and generated-by edges that led to (i.e., that lie on a path to) a data token d denote the lineage of d . We use the following auxiliary relation to compute the data dependencies.

$$ddep(D_1, D_2) :- genBy(D_1, I), used(I, D_2).$$

The $ddep(D_1, D_2)$ relation specifies that the data D_1 depends on the data D_2 . Additionally, given the workflow graph W that produced the provenance graph P , where A and C are the actors and containers of W , respectively, we assume the relation $invoc \subseteq I \times A$ connects each invocation $i \in I$ with its corresponding actor $a \in A$ and the relation $loc \subseteq D \times C \times L$ connects each data token $d \in D$ to its corresponding container $c \in C$ such that d is located at position $l \in L$ in c 's persistent queue.

3 Precomputing Dependency Tables

Given a workflow W and a provenance graph P , we statically compute the lineage of all data tokens in three steps. The first step dependencies containers using the following Datalog rules.

$$\begin{aligned} cdep(C_1, C_2) &:- out(P, C_1), in(C_2, P). \\ cdep^*(C_1, C_2) &:- cdep(C_1, C_2). \\ cdep^*(C_1, C_2) &:- cdep(C_1, C), cdep^*(C, C_2). \end{aligned}$$

The relation $\text{cdep}^*(C_1, C_2)$ captures all the containers C_2 on which the container C_1 depends, i.e., some token in C_1 may be derived either directly or transitively from tokens in C_2 . We call C_2 the container dependency of each token in C_1 . Note that while all tokens in a container C_1 have the same set of container dependencies, they may depend on different tokens within those containers.

The second step computes the positions of all the tokens in all container dependencies C_2 on which a token at position l in container C_1 depends by using the consumption and production rates. The result of this step is a relation

$$\text{dependency}(D, C_1, L, C_2, L_S, L_E)$$

where D is a data token in container C_1 at position L that depends on the tokens in container C_2 starting at position L_S and ending at position L_E . We describe how this relation is computed in the rest of this section and in Sect. 4. Finally, the third step uses the dependency relation to answer lineage queries, which is also further described in Sect. 4. The result of this step is a (virtual) relation

$$\text{lineage}(D_1, D_2, D_3)$$

where data tokens D_2 and D_3 form a dependency edge that lies on the lineage path of D_1 . Thus, $\text{ddep}(D_1, D_2)$, $\text{ddep}(D_1, D_3)$, and $\text{ddep}(D_2, D_3)$ hold such that given a specific token d , $\text{lineage}(d, D_2, D_3)$ gives the set of dependency ($\text{ddep}(D_2, D_3)$) edges that represent the lineage of d .

The following definitions are used to compute the dependency and lineage information. We assume below that x is a container dependency of y and that $y[k]$ denotes the k^{th} position in y .

- $\text{end}_x(y[k])$ is the last position in x that the token at $y[k]$ depends.
- width_{xy} is the number of consecutive positions in x that tokens in y depend on.
- $\text{start}_x(y[k])$ is the first position in x that the token at $y[k]$ depends on such that $\text{start}_x(y[k]) = \text{end}_x(y[k]) - \text{width}_{xy} + 1$.
- $\text{dep}_x(y[k])$ is the sequence of positions in x that the token at $y[k]$ depends on such that $\text{dep}_x(y[k]) = [\text{start}_x(y[k]), \text{start}_x(y[k]) + 1, \dots, \text{end}_x(y[k])]$.

The rest of this section describes how to compute $\text{end}_i(j[k])$ and width_{ij} for various types of actors and workflow patterns. The $\text{start}_i(j[k])$ is then computed using $\text{end}_i(j[k])$ and width_{ij} .

Stateless Actors. Consider the actor B in Fig. 1 (which we assume here is stateless). An invocation of B consumes v_c tokens from container V and produces x_p tokens in container X . Let's assume that we want to know the dependencies of the k^{th} token in X on the tokens in V . To do so, we need to know the invocation of B that produced the k^{th} token in X as well as all of the tokens from V that were consumed. Since in each invocation, B outputs x_p tokens into X , $\lceil \frac{k}{x_p} \rceil$ is the invocation during which the k^{th} token was produced in X and as B consumes v_c tokens from V per invocation, $\text{end}_v(x[k]) = v_c * \lceil \frac{k}{x_p} \rceil$ and $\text{width}_{vx} = v_c$. Thus,

tokens from positions $start_v(x[k])$ through $end_v(x[k])$ in V were consumed to produce the k^{th} token in X .

Now, let's assume that we want to know the dependencies of the k^{th} token in container X on the tokens in container U in Fig. 1 (again, assuming A is stateless). To do so, we first compute $end_v(x[k])$ and $width_{vx}$ as above and then use these two values to compute $end_u(x[k])$ and $width_{ux}$, where $end_u(x[k]) = u_c * \lceil \frac{end_v(x[k])}{v_p} \rceil$ and $width_{ux} = u_c * \lceil \frac{width_{vx}}{v_p} \rceil$. We extend this approach to a chain of n actors, where we want to know the dependencies of the k^{th} token in the j^{th} container on the tokens in the i^{th} container. We use the following formulas to compute $end_i(j[k])$ and $width_{ij}$.

$$end_i(j[k]) = \begin{cases} i_c * \lceil \frac{end_{i+1}(j[k])}{(i+1)_p} \rceil & \text{if } j > i + 1 \\ i_c * \lceil \frac{k}{j_p} \rceil & \text{if } j = i + 1 \end{cases}$$

$$width_{i,j} = \begin{cases} i_c * \lceil \frac{width_{i+1,j}}{(i+1)_p} \rceil & \text{if } j > i + 1 \\ i_c & \text{if } j = i + 1 \end{cases}$$

Feedback Loops. A workflow has a feedback loop if there is a cycle among the actors and containers as shown in Fig. 2(a) and (b). In Fig. 2(a), actor A is connected to container X with consumption and production rates x_c and x_p , respectively. To prevent deadlock², x_c tokens are initially provided in X before invocations are started. In this case, tokens from $(x_c + 1)^{th}$ through $(2 * x_c)^{th}$ positions in X , which are generated during the 1^{st} invocation of A , will depend on the first x_c tokens in X . Subsequently, the p^{th} set of x_c tokens in X , which were generated during the $(p - 1)^{th}$ invocation of A , will depend on the $(p - 1)^{th}$ set of x_c tokens in X . Thus, any token generated during the p^{th} invocation will depend on the 1^{st} through $x_c * (p - 1)$ tokens in X . Using this idea we compute $end_x(x[k])$ as shown below. Here $width_{xx} = end_x(x[k])$, i.e., $start_x(x[k]) = 1$.

Fig. 2. Two example workflows containing feedback loops.

$$end_x(x[k]) = \begin{cases} x_c * (\lceil \frac{k}{x_p} \rceil - 1) & \text{if } k > x_p \\ 0 & \text{otherwise} \end{cases}$$

² Actors in a feedback loop would be in deadlock as an actor in the loop would expect input tokens in its input ports. But, all actors in the loop expects the same and thus, it would get into a deadlock [8].

In Fig. 2(b), a *SimpleDelay* (*DL*) actor is used to avoid deadlock and we assume that W is the starting container into which *DL* initially outputs n tokens, where $2 * w_c > n \geq w_c$ [8]. Here, containers V , W , and X are contained in a workflow loop. Now assume we want to know the dependencies of the k^{th} token in the j^{th} container on the tokens in the i^{th} container. In this case, if the j^{th} container depends on all of the containers in the loop, we use the following formula to compute $end_i(j[k])$. Here $width_{ij} = end_i(j[k])$, i.e., $start_i(j[k]) = 1$.

$$end_i(j[k]) = \begin{cases} i_c * \lceil \frac{end_{i+1}(j[k]) - n}{(i+1)_p} \rceil & \text{if } i+1 \text{ is the starting container, e.g., } W \\ i_c * \lceil \frac{end_{i+1}(j[k])}{(i+1)_p} \rceil & \text{if } i+1 \text{ is not the starting container} \\ 0 & \text{if } k \leq n \text{ and } i+1 \text{ is the starting container} \end{cases}$$

If the j^{th} container does not depend on any of the containers in the loop, e.g., if we want to know $end_y(z[k])$ in Fig. 2(b), then we use the formulas discussed above for stateless actors.

Stateful Actors. Stateful actors vary based on how they buffer and pass tokens from one invocation to the next. As discussed above, we consider two variations: (1) *Fixed Buffering*, and (2) *Dynamic Buffering*. Let's assume actor A is a *Fixed Buffering* actor with an input container X and an output container Y such that during any invocation, A consumes x_c tokens from X and produces y_p tokens into Y . When an invocation starts, actor A first fills the buffer by consuming x_c tokens per invocation and once the buffer is full, in subsequent invocations it removes x_c tokens from the buffer (i.e., the queue) consuming x_c new tokens, while keeping the buffer size at x_s . Thus, to know the dependencies of the k^{th} token in Y on the tokens in X , we compute $end_x(y[k])$ and $width_{xy}$, where $end_x(y[k]) = x_c * \lceil \frac{k}{y_p} \rceil$ and $\lceil \frac{k}{y_p} \rceil$ is the invocation during which the k^{th} token was generated. Similarly, $width_{xy} = x_s$ if the buffer is full, otherwise $width_{xy} = x_c * \lceil \frac{k}{y_p} \rceil$. Thus, given a chain of actors, to compute the dependencies of the k^{th} token in the j^{th} container on the tokens in the i^{th} container, we use the following formulas for $end_i(j[k])$ and $width_{ij}$.

$$end_i(j[k]) = \begin{cases} i_c * \lceil \frac{end_{i+1}(j[k])}{(i+1)_p} \rceil & \text{if } j > i+1 \\ i_c * \lceil \frac{k}{j_p} \rceil & \text{if } j = i+1 \end{cases}$$

$$width_{i,j} = \begin{cases} i_c * \lceil \frac{width_{i+1,j}}{(i+1)_p} \rceil + i_s - i_c & \text{if } \lceil \frac{i_s}{i_c} \rceil \leq \lceil \frac{end_{i+1}(j[k])}{(i+1)_p} \rceil - \lceil \frac{width_{i+1,j}}{(i+1)_p} \rceil + 1 \\ i_c * \lceil \frac{end_{i+1}(j[k])}{(i+1)_p} \rceil & \text{Otherwise} \end{cases}$$

If an actor instead uses *Dynamic Buffering*, it will consume all of its buffered tokens in each of its previous invocations. Note that the dependency computation for this type of actor is exactly the same as with feedback loops with a single actor as discussed above.

Example. We now show (by example) how to use the formulas discussed in this section. Consider the example workflow shown in Fig. 1 and assume that all the

Fig. 3. This is partial execution details of the workflow in Fig. 1. In (a), and (b) we show partial invocation details of actors *B*, and *A* respectively. In (c) we show the relationship among *U* and *X* channel, which are transitively dependent.

actors are stateless. Assume we want to know the dependencies of the 3rd token in container *X*, i.e., on which all tokens in containers *V* and *U* the 3rd token in container *X* depends. We use the $\text{cdep}^*(C_1, C_2)$ to find out that any token in *X* depends on tokens in *V* and *U*. First, we find the dependencies of $x[3]$ on the tokens in *V*. Here, $x_p = 2$, $k = 3$, and $v_c = 3$ using the stateless actor formulas from which we get $\text{end}_v(x[k]) = 6$ and $\text{weight}_{vx} = 3$. That is, the token at $x[3]$ depends on the tokens at $v[4]$, $v[5]$, and $v[6]$, as shown in Fig. 3(a). These dependencies are captured in the dependency relation as $\text{dependency}(id, x, 3, v, 4, 6)$, where *id* is assumed to be the token identifier for $x[3]$. Second, we need to find the dependencies of $v[4]$, $v[5]$, and $v[6]$ on the tokens in *U*. Here, $v_p = 2$, $k = 6$, and $u_c = 2$ and thus we get $\text{end}_u(v[k]) = 6$ and $\text{weight}_{uv} = 4$. Then $v[4]$, $v[5]$, and $v[6]$ tokens depend on the $u[3]$, $u[4]$, $u[5]$, and $u[6]$ tokens, which is shown in Fig. 3(b). These dependencies are captured in the dependency relation as $\text{dependency}(id, x, 3, u, 3, 6)$.

In Sect. 4, we discuss how to compute the **lineage** relation once all the **dependency** tuples have been obtained.

4 Querying Lineage Using Dependency Tables

Our approach allows users to ask for the lineage of one or more data tokens within a single query. Here we assume that each of the data tokens D_1 from which lineage should be computed is stored in a relation $\text{input}(D_1)$. From $\text{dependency}(D_1, C_1, L_1, C_2, L_S, L_E)$, we know that D_1 is a data token in container C_1 at position L_1 and that it depends on the tokens in container C_2 starting at position L_S and ending at position L_E . We also assume a relation $\text{loc}(D_2, C_2, L_2)$ that captures the tokens stored within each container during workflow execution such that a token D_2 was stored in container C_2 at location L_2 . Given these relations, we use the following Datalog rules to compute the **lineage** relation.

$$\begin{aligned} \text{depData}(D_1, D_2) &:- \text{input}(D_1), \text{dependency}(D_1, C_1, L_1, C_2, L_S, L_E), \\ &\quad \text{loc}(D_2, C_2, L_2), L_2 \geq L_S, L_2 \leq L_E. \\ \text{lineage}(D_1, D_2, D_3) &:- \text{depData}(D_1, D_2), \text{depData}(D_1, D_3), \text{ddep}(D_2, D_3). \end{aligned}$$

As shown, the temporary $\text{depData}(D_1, D_2)$ relation computes the data tokens D_2 that D_1 has as a dependency by comparing D_2 's position in container C_2 against L_S and L_E . This information is then used to build the final $\text{lineage}(D_1, D_2, D_3)$ relation.

To better understand the performance of our location-based lineage technique we compare its runtime and space requirements to lineage computation techniques based on the semi-naïve query evaluation approach and the approach of directly storing the transitive dependency closure. We briefly describe these two techniques below.

Semi-Naïve Query Evaluation. First, we query the $\text{ddep}(D_1, D_2)$ relation to find the tokens D_2 , on which D_1 directly depends. Then, we compute the “transitive” dependencies of D_1 in rounds, where in each round we find new reachable data tokens. The $\text{dep}(D_1, D_2, D_3, J)$ relation captures the reachable token D_3 along with the token D_2 from which D_3 is reachable from D_1 . Here, J is the round number with at most N rounds, where N is the diameter of the data dependency graph (based on the $\text{ddep}(D_1, D_2)$ relation). This approach is implemented using the following Datalog rules and further details can be found in [6].

$$\begin{aligned} \text{delta}(D_1, D_2, D_3, I) &:- \text{ddep}(D_1, D_3), I = 1, \text{input}(D_1), D_2 = D_1. \\ \text{newDep}(D_1, D_2, D_3, J) &:- \text{delta}(D_1, D_2, D, I), \text{ddep}(D, D_3), J = I + 1. \\ \text{delta}(D_1, D_2, D_3, J) &:- \text{newDep}(D_1, D_2, D_3, J), \neg \text{dep}(D_1, D_2, D_3, I), I = J - 1. \\ \text{dep}(D_1, D_2, D_3, J) &:- \text{delta}(D_1, D_2, D_3, J). \\ \text{lineage}(D_1, D_2, D_3) &:- \text{dep}(D_1, D_2, D_3, -). \end{aligned}$$

Transitive Closure Based Query Evaluation. In this approach, the transitive closure of data dependencies is first computed and stored. Once stored, all subsequent lineage queries are answered directly from the closure. The following Datalog rules demonstrate the approach where the transitive closure of the $\text{ddep}(D_1, D_2)$ is stored in the $\text{ddep}^*(D_1, D_2)$ relation. Then, the $\text{lineage}(D_1, D_2, D_3)$ relation is computed using the $\text{ddep}^*(D_1, D_2)$ and $\text{ddep}(D_1, D_2)$ relations.

$$\begin{aligned} \text{ddep}^*(D_1, D_2) &:- \text{ddep}(D_1, D_2). \\ \text{ddep}^*(D_1, D_2) &:- \text{ddep}(D_1, D), \text{ddep}^*(D, D_2). \\ \text{lineage}(D_1, D_2, D_3) &:- \text{input}(D_1), \text{ddep}^*(D_1, D_2), \text{ddep}^*(D_1, D_3), \text{ddep}(D_2, D_3). \end{aligned}$$

5 Experiments and Results

Experiment Setup. We used three workflow patterns as shown in Fig. 4 to evaluate our *location-based lineage (LBL)* computation technique against the

Fig. 4. Different workflow patterns we used in our experiments.

two natural choices *Semi-Naive Query Evaluation (SNL)* and *Transitive Closure Based Query Evaluation (TCL)*³. In all our experiments, the workflow specification and provenance graphs were generated for all three workflows using the respective models presented in Sect. 2. We generated provenance graphs for first workflow shown in Fig. 4(a), which forms a *chain* pattern, with 30 tokens in the first container, all the actors with three invocations, and with both the consumption and production rates equal to 10 for all containers, while varying the number of actors in the chain. Similarly, for the second workflow shown in Fig. 4(b), which forms a *ladder graph* pattern, we assumed only one token for both the initial containers and assumed both the consumption and production rates equal to 1 for all containers, and we generated the provenance graphs by varying the number of actors in the graph. For the third workflow shown in Fig. 4(c), which forms a *binary tree* pattern, we assumed only one token to the initial container and assumed both the consumption and production rates to be 1 for all the containers, and generated provenance graphs by varying the height of the tree.

For all three lineage querying techniques, i.e., *LBL*, *SNL*, and *TCL*, discussed in this paper, we persist the provenance graph. In addition, for *LBL* we compute and persist the $\text{dependency}(D, C_1, L, C_2, L_S, L_E)$, and $\text{loc}(D, C, L)$ relations and for *TCL* we compute and persist the $\text{ddep}^*(D_1, D_2)$ relation.

We then evaluated lineage queries using the algorithms discussed in Sect. 4, where we ran all the queries 100 times and took an average query time.

Fig. 5. Comparisons of run times of computing lineage.

Analysis. When we review the chart in Fig. 5(a), we see that as the size of the workflow grow, i.e., the number of actors grow, *SNL* outperforms *TCL*.

³ we introduce these acronyms to be used in the charts presenting the results in Figs. 5 and 6.

This is because of the high growth rate of the $\text{ddep}^*(D_1, D_2)$ relation for TCL over the size of the workflow. In Fig. 5(b), TCL outperforms SNL . There are two reasons, (i) number of iterations for SNL , which is directly proportional to the size of the graph, and (ii) growth in data volume, which is not high in this case as both consumption and productions are 1. Thus, the growth in data volume of the $\text{ddep}^*(D_1, D_2)$ relation for TCL is not significant. Now, in case the containers have higher consumption of productions rates as in the case of Fig. 5(a), SNL would eventually outperform TCL . In Fig. 5(c), we find TCL to be non-linear, where as both SNL and LBL are linear with very low slopes. This is because of the properties of a binary tree. From any given leaf node, to find its lineage, SNL needs the number of iterations equal to the height of the binary tree and in each iteration, SNL only find one new edge, whereas the volume of the $\text{ddep}^*(D_1, D_2)$ relation for TCL is large, which be seen in Fig. 6(c). In all three charts in Fig. 5, we see that LBL to be linear with very low slopes and we observed that as the size of workflow and the consumption and production rates grows, LBL scales better compare to TCL and SNL . Here, the observations are (i) when the consumption and production rates grows, the volume of $\text{ddep}^*(D_1, D_2)$ relation grows rapidly adversely impacting TCL , but does not impact LBL , and (ii) when the size of the workflow grows, the number of rounds for SNL grows, which impacts its performance, without impacting LBL .

Fig. 6. Comparisons of additional space requirements of computing lineage

We discussed in *Experiment Setup* that for SNL we only store the provenance graph, but for both TCL and LBL we store additional metadata towards improving efficiencies of lineage queries. Thus, we compare these additional storage requirements by both TCL and LBL as shown in Fig. 6. In Fig. 6(a), we see that LBL is linear with the size of the workflow, whereas TCL is not. TCL maintains all pairs of token dependencies with a storage cost of $\mathcal{O}(|V|^2)$, whereas LBL maintains only one record for all the dependencies for a token to all tokens of another container with a storage cost of $\mathcal{O}(|V| * k)$, where $|V|$ and k are the number of tokens and the number of containers respectively, with $|V| \gg k$. Now, in case there is only one token in a container then storage requirements of both LBL and TCL become same. This is the reason why in Fig. 6(b) and (c) the space requirements for both LBL and TCL are same.

Thus, these experiments show that *LBL* outperforms the traditional lineage querying techniques and is more scalable both in query time and additional space requirements.

6 Related Work

The problem of efficiently evaluating lineage queries has been an active area of research and many approaches have been introduced. Heinis et al. [5] proposed an extension to tree-based interval encoding that supports DAGs. As part of this approach, a DAG representing provenance information is converted to a (compressed) tree structure. While this can improve query execution time (based on using interval encodings), the storage cost can significantly increase since shared portions of the graph are copied in the corresponding tree structure. Both [5] and *LBL* support lineage for all tokens, where only *LBL* is both space and time efficient.

The Zoom*UserView by Biton et al. [12] allows users to specify the relevant parts of a workflow, customize both the workflow and provenance based on that specification, and then query the reduced provenance graph based on a “virtual” workflow. Missier et al. [13] developed an efficient and scalable algorithm for querying fine-grained lineage information by exploiting the model of computation used in the Taverna workflow system [14]. *LBL* is similar to [13] as both techniques are exploiting the constraints of models of computation, but, *LBL* (i) precomputes lineage even before the execution of the workflow by forecasting the sizes of the input containers and later adjusts the lineage with the actual sizes, and (ii) enables lineage for all the tokens and thus expands the use of provenance, e.g., for focused data analysis where only input dependencies of an output is needed, and debugging where dependencies on the intermediate tokens are also needed.

Trio [15] and GridDB [16] use recursive query evaluation on a collection-based data model to answer lineage queries. Both [15, 16] support lineage for all tokens as *LBL* does, but, *LBL* is time efficient while incurring very little additional space cost.

7 Conclusion and Future Work

Lineage information plays a key role in helping users understand and reuse data generated by scientific workflow systems. Many applications of provenance within these systems rely on being able to easily pose and efficiently answer lineage queries, which for data-intensive workflows require evaluation techniques that are both time and space efficient. While semi-naïve query evaluation is generally space efficient, it may result in slow query execution time, whereas computing and storing transitive closures can result in faster query execution time at the cost of increased storage space. In this paper, we have developed a new location-based lineage approach that is both space and time efficient. Our approach exploits information available in workflow specifications, in particular,

container dependency information and the consumption-production rate constraints used in many workflow systems. Our experimental results demonstrate that the location-based lineage technique is both efficient and scalable for various types of workflow patterns and results in both faster query evaluation time and lower storage space requirements than using semi-naïve query evaluation and storing transitive closures. As future work, we are currently extending the location-based approach presented here to support more complex data structures (e.g., collections of data tokens) that are increasingly being developed for more general dataflow frameworks.

Acknowledgments. Supported in part by NSF ACI-0830944 and IIS-1118088.

References

1. Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., Myers, J.: Examining the challenges of scientific workflows. *Computer* **40**(12), 24–32 (2007)
2. Davidson, S.B., Boulakia, S.C., Eyal, A., Ludäscher, B., McPhillips, T.M., Bowers, S., Anand, M.K., Freire, J.: Provenance in scientific workflow systems. *IEEE Data Eng. Bull.* **30**(4), 44–50 (2007)
3. Miles, S., Deelman, E., Groth, P., Vahi, K., Mehta, G., Moreau, L.: Connecting scientific data to scientific experiments with provenance. In: *Proceedings of the IEEE International Conference on e-Science and Grid Computing*, pp. 179–186 (2007)
4. Anand, M.K., Bowers, S., Ludäscher, B.: Techniques for efficiently querying scientific workflow provenance graphs. In: *EDBT*, pp. 287–298 (2010)
5. Heinis, T., Alonso, G.: Efficient lineage tracking for scientific workflows. In: *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, pp. 1007–1018. ACM (2008)
6. Abiteboul, S., Hull, R., Vianu, V.: *Foundations of Databases*, vol. 8. Addison-Wesley, Reading (1995)
7. Koop, D., Freire, J., Silva, C.T.: Enabling Reproducible Science with VisTrails. *CoRR* abs/1309.1784 (2013)
8. Lee, E.A., Messerschmitt, D.G.: Synchronous data flow. *Proc. IEEE* **75**(9), 1235–1245 (1987)
9. Sun, S., Chen, J., Li, W., Altintas, I., Lin, A.W., Peltier, S., Stocks, K., Allen, E.E., Ellisman, M.H., Grethe, J.S., Wooley, J.C.: Community cyberinfrastructure for advanced microbial ecology research and analysis: the CAMERA resource. *Nucleic Acids Res.* **39**, 546–551 (2011)
10. Altintas, I., Wang, J., Crawl, D., Li, W.: Challenges and approaches for distributed workflow-driven analysis of large-scale biological data: vision paper. In: *EDBT/ICDT Workshops*, pp. 73–78 (2012)
11. Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., den Bussche, J.V.: The open provenance model core specification (v1.1). *Future Gener. Comput. Syst.* **27**(6), 743–756 (2011)
12. Biton, O., Cohen-Boulakia, S., Davidson, S.: Zoom* userviews: querying relevant provenance in workflow systems. In: *Proceedings of the 33rd International Conference on Very Large Data Bases*, pp. 1366–1369. VLDB Endowment (2007)

13. Missier, P., Paton, N.W., Belhajjame, K.: Fine-grained and efficient lineage querying of collection-based workflow provenance. In: EDBT, pp. 299–310 (2010)
14. Turi, D., Missier, P., Goble, C., De Roure, D., Oinn, T.: Taverna workflows: syntax and semantics. In: International e-Science and Grid Computing Conference, pp. 441–448 (2007)
15. Benjelloun, O., Sarma, A.D., Halevy, A., Theobald, M., Widom, J.: Databases with uncertainty and lineage. VLDB J. **17**(2), 243–264 (2008)
16. Liu, D.T., Franklin, M.J.: GridDB: a data-centric overlay for scientific grids. In: Proceedings of the Thirtieth International Conference on Very Large Data Bases, vol. 30, pp. 600–611. VLDB Endowment (2004)