Versioned-PROV: A PROV extension to support mutable data entities

João Felipe N. Pimentel$^{1[0000-0001-6680-7470]}$, Paolo Missier$^{2[0000-0002-0978-2446]}$, Leonardo Murta$^{1[0000-0002-5173-1247]}$, and Vanessa Braganholo$^{1[0000-0002-1184-8192]}$

^{1} Instituto de Computação, Universidade Federal Fluminense, Niterói, Brazil,
{jpimentel, leomurta, vanessa}@ic.uff.br
^{2} School of Computing, Newcastle University, Newcastle upon Tyne, UK,
paolo.missier@newcastle.ac.uk

Abstract. The PROV data model assumes that entities are immutable and all changes to an entity e are represented by the creation of a new entity e'. This is reasonable for many provenance applications but may produce verbose results once we move towards fine-grained provenance due to the possibility of multiple binds (i.e., variables, elements of data structures) referring to the same mutable data objects (e.g., lists or dictionaries in Python). Changing a data object that is referenced by multiple immutable entities requires duplicating those immutable entities to keep consistency. This imposes an overhead on the provenance storage and makes it hard to represent data-changing operations and their effect on the provenance graph. In this paper, we propose a PROV extension to represent mutable data structures. We do this by adding reference derivations and checkpoints. We evaluate our approach by comparing it to plain PROV and PROV-Dictionary. Results indicate a reduction in the storage overhead for assignments and changes in data structures from $O(N)$ and $\Omega(R \times N)$, respectively, to $O(1)$ in both cases when compared to plain PROV (N is the number of members in the data structure and R is the number of references to the data structure).

Keywords: Provenance · Specification · Interoperability.

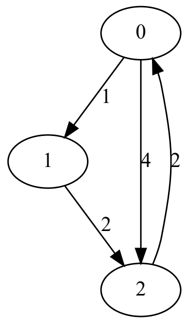
1 Introduction

The PROV data model [6] is an extensible domain-agnostic model that describes the provenance of entities through their relationships with activities, agents, and other entities. An *entity* is a term used to represent any data, physical object, or concept whose provenance may be obtained. The *activity* term describes actions or processes that use entities and generate other entities. Finally, the *agent* term describes roles in activities.

PROV (and its predecessor, OPM [10]) has been applied to describe the provenance gathered from operating systems [11], workflow systems [2], and scripts [1]. Tools that collect operating system provenance map users as agents, file objects and program arguments as entities, and program executions and system calls as activities [11]. Workflow systems map data as entities and processing steps as activities [2]. Finally, tools that collect coarse-grained provenance from scripts map data in function arguments and data values obtained from return statements as entities, and function calls as activities [1].

```
m = 10000 # max value
result = dist = [
    [0, 1, 4],
    [m, 0, 2],
    [2, m, 0]]
nodes = len(dist)
indexes = range(nodes)
for k in indexes:
    distk = dist[k]
    for i in indexes:
        if i == k: continue
        disti = dist[i]
        for j in indexes:
            if j == i or j == k: continue
            ikj = disti[k] + distk[j]
            if disti[j] > ikj:
                disti[j] = ikj
print(result[0][2])
```

(A)



(B)

Fig. 1. *Floyd-Warshall* implementation (A) and encoded input graph (B).

In the aforementioned approaches, entities are immutable data that go through processing steps (modeled as activities) to produce new immutable data (modeled as entities). The assumption of immutable entities also exists in the PROV data model, where changes to an entity e are explicitly represented by the creation of a new entity e' generated by the activities that use the original e.

No known approaches use PROV to describe fine-grained provenance from scripts, with support for variables and mutable data structures. Our goal is to extend the well-known concepts of coarse-grained provenance for scripts, which is limited to function arguments and function calls, to (1) script variables, (2) expressions with operators, and (3) assignments, thus realizing fine-grained provenance for scripts. Specifically, we note that we can map script variables to entities, expressions with operators to activities that generate new entities, and assignments to activities that produce derivations, i.e., from expression results to variables. For example, `a = b + c` can be mapped as an activity + that uses the entities `b` and `c` to generate the derived entity `sum`, and an assignment activity that uses `sum` to generate the derived entity `a`.

This is a challenging goal because using PROV to represent fine-grained provenance suffers from two main problems: (P1) when an entity that represents a collection is changed (e.g., a list is updated to add an element), a new entity should be created, together with multiple new relationships, connecting the new entity to each of the existing or new entities that represent the elements of the collection; and (P2) when more than one variable is assigned to the same collection, and one of the variables changes, all other variables should also change, as they refer to the same memory area. This means that a new entity should be created for each variable that contains the collection, together with edges for all entities that represent the elements of the collection. As we show in Section 4, these problems lead to $O(N)$ and $\Omega(R \times N)$ extra elements in the provenance graph, respectively, for collections with N elements and R references.

PROV-Dictionary [8] improves the support for data structures in PROV by adding derivation statements that indicate that a new collection shares most elements of the old one, but with the insertion or removal of specific elements. This solves P1 since it reduces the number of edges to 1. However, it still suffers from P2, since it requires updating all entities that refer to the same collection when it changes, which leads to $\Omega(R)$ extra elements.

We propose Versioned-PROV, an extension that adds *reference sharing* and *checkpoints* to PROV. Checkpoints solve problem P1 in $O(1)$ by allowing the representation of multiple versions of collections with a single entity. Reference sharing solves problem P2 in $O(1)$ by allowing collections to be represented only once and referred to by other entities through reference derivations plus checkpoints to indicate states.

This paper is organized as follows. Section 2 presents a running example, which is based on the *Floyd-Warshall* algorithm [3]. Section 3 introduces Versioned-PROV. Section 4 evaluates the approach by comparing it to PROV and PROV-Dictionary. Section 5 discusses related work, and Section 6 concludes the paper.

2 Running Example

While Versioned-PROV intends to be generic enough for any situation that requires sharing references to mutable collections in PROV, we use fine-grained script provenance as a case study for presenting our extension. More specifically, we use the *Floyd-Warshall* algorithm [3] as a base to describe and evaluate the mapping of fine-grained provenance from scripts using Versioned-PROV. This algorithm has relevant applications, such as finding the shortest path between two addresses in a navigation system.

The algorithm calculates the length of the shortest path between all pairs of nodes in a weighted graph. It achieves this by updating the distance of the path from node `i` to node `j` if there is a node `k` for which the distance of the path from `i` to `k` plus the distance of the path from `k` to `j` is shorter than the distance from `i` to `j`. The result of *Floyd-Warshall* is the set of shortest distances among all pairs of nodes, but it does not produce the actual shortest paths. However, observing that the path between two nodes is defined by the sum of two other paths, here we show that we can use the fine-grained provenance of a given output distance to obtain the actual paths that have that distance.

Fig. 1 presents a Python implementation of *Floyd-Warshall* with a predefined input graph. Line 18 prints the distance of the shortest path from `0` to `2`. While there is a direct edge with cost 4, the actual result is 3, because the shortest path goes from `0` to `1`, with cost 1, and then from `1` to `2`, with cost 2. After the algorithm changes the result matrix, querying the provenance of `result[0][2]` in line 18 should indicate that it derives from `result[0][1]` and `result[1][2]`.

3 Versioned-PROV

Versioned-PROV adds the concepts of checkpoints, reference sharing, and accesses to PROV. Different from plain PROV, which assumes immutable entities, a Versioned-PROV entity may represent multiple versions of a data object. We present Versioned-PROV concepts in Section 3.1. In Section 3.2, we detail Versioned-PROV by presenting a mapping of a part assignment in the *Floyd-Warshall* algorithm, and contrasting it to PROV and PROV-Dictionary.

Table 1. Versioned-PROV types.

Type	Statement	Meaning
Reference	wasDerivedFrom	The generated entity derived from the used entity by reference, indicating that both have the same numbers.
Put	hadMember	Put a member into a collection *key* position at a given *checkpoint*. Using a placeholder as member indicates a deletion.

Table 2. Versioned-PROV attributes.

Attribute	Range	Statement	Meaning
checkpoint	Sortable Value	hadMember	Checkpoint of the collection update. Required for *hadMember* with type *Put*.
checkpoint	Sortable Value	Events (e.g., used, wasDerivedFrom)	Checkpoint of the event. Required for *wasDerivedFrom* with type *Reference*.
key	String	hadMember	The position of *Put*.
key	String	wasDerivedFrom	The position of the accessed *collection* entity.
collection	Entity Id	wasDerivedFrom	Collection entity that was accessed or changed.
access	'r' or 'w'	wasDerivedFrom	Indicates whether an access reads ('r') an element from a collection or writes ('w') into it.

3.1 Concepts

The PROV data model is based on the idea of instantaneous transition events that describe usage, generation, and invalidation of entities [6]. These events are important to describe the provenance timeline without explicit time and ordering. Versioned-PROV builds on top of PROV events and determines that a version of a data object changes on a generation event, and is accessed on a usage event. Instead of relying on the implicit ordering of events from PROV, Versioned-PROV uses *checkpoint* attributes to tag events and changes on entities. Then, it uses the explicit ordering of *checkpoints* to obtain a version of a data object. Hence, we require a total order to be defined on the set of checkpoints. Our implementation of *Floyd-Warshall* uses *timestamps* as checkpoints, but the figures in this paper use *sequential numbers*. Both can be ordered.

As an extension of PROV, Versioned-PROV follows its semantics. Thus, despite the goal of representing multiple versions of a data object, an entity in PROV can only be generated once, according to the unique-generation constraint of PROV [6]. Thus, the only mutability on the Versioned-PROV entities occurs in the memberships of collection entities. A collection may have different members at different moments, but the operations that put and delete members from a collection are incremental. It means that if a collection `c` had an entity `e1` at checkpoint `1` and an operation put the entity `e2` into a different position of `c` at checkpoint `2`, then `c` had both `e1` and `e2` at checkpoint `2`.

Different from PROV and PROV-Dictionary that use copy-by-value to represent data-structure assignments and derivations, Versioned-PROV uses copy-by-reference. Hence, it defines the data structure once and uses *reference sharing* to indicate that more than one entity refers to the same data structure. When generating and using Versioned-PROV entities, one must indicate a checkpoint to unfold the specific version of the data structure for any given event. When an entity associated with a data structure changes at

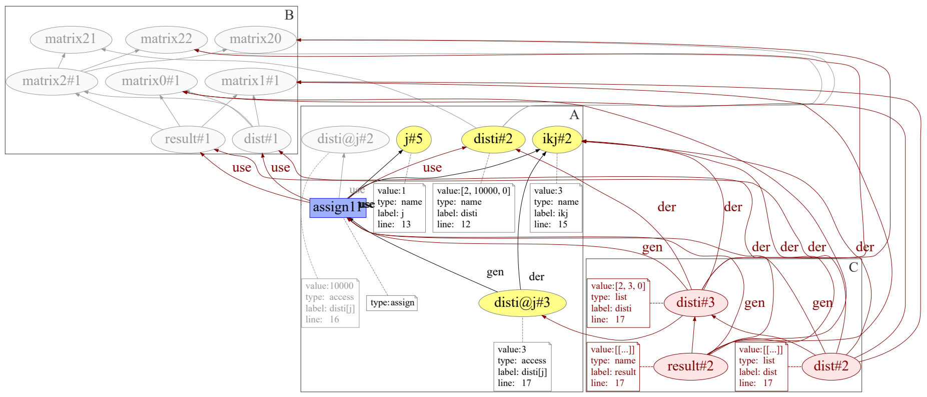


Fig. 2. Plain PROV mapping of `disti[j] = ikj`.

a given checkpoint, we can infer that all entities that share reference with it also changes at the same checkpoint, without any extra explicit statements.

Versioned-PROV uses PROV optional attributes and defines types to extend PROV. Table 1 presents the Versioned-PROV types, and Table 2 presents the Versioned-PROV attributes. The attributes *key*, *collection*, and *access* of *wasDerivedFrom* may only be used when the derivation is related to an access or collection update. Similarly, the type *Put* can only appear in data structures, to define their items. Differently, the attribute *checkpoint* and the type *Reference* can appear anywhere, despite affecting only collection entities. This keeps the model consistent in all situations that involve using and generating entities.

3.2 Mapping Example

We use the script example of Section 2 to detail Versioned-PROV in contrast to PROV and PROV-Dictionary. We map the execution provenance of the *Floyd-Warshall* algorithm (Fig. 1) to these three approaches. Due to space constraints, we present only the first execution of the part assignment in line 17 of Fig. 1 (i.e., `disti[j] = ikj`). The complete mapping is available at [13].

Fig. 2, Fig. 3, and Fig. 4 present the part assignment mapped to plain PROV, PROV-Dictionary, and Versioned-PROV, respectively. In our mappings, we name entities based on their textual representations. Since a textual element (e.g., a variable) can be represented by multiple entities, we enumerate them. Thus, `ikj#2` denotes the second entity that represents the variable `ikj` (as defined in line 15 of Fig. 1). In addition to this numbering, we change the notation of accesses to avoid using escaping characters to represent square brackets. Instead, we use the collection name followed by "`@`" and the accessed key. For instance, we use `disti@j` to represent `disti[j]` (lines 16-17 of Fig. 1). Note in region A of these figures that we have both `disti@j#2` in gray, representing `disti[j]` of line 16, and `disti@j#3` in yellow, representing `disti[j]` of line 17. The latter is the result of the part assignment.

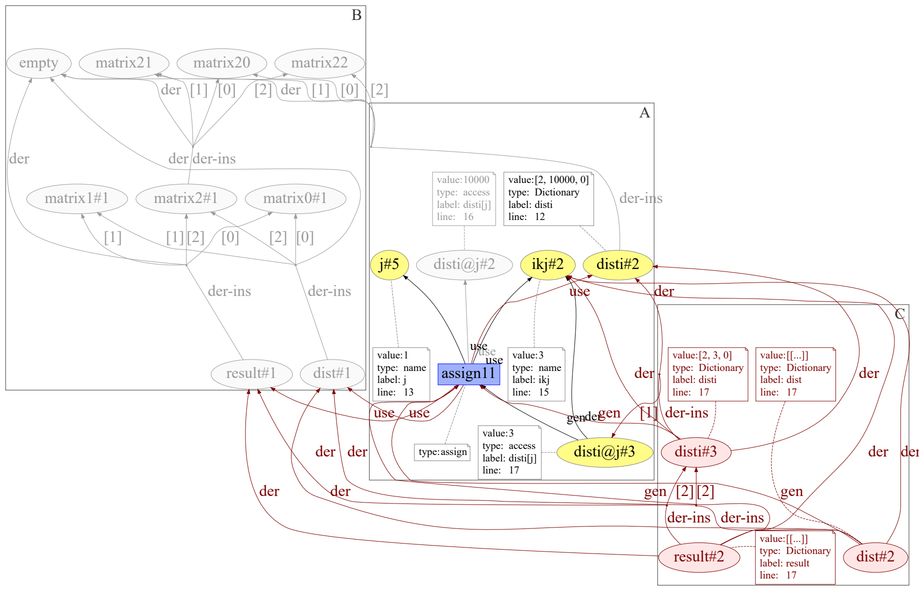


Fig. 3. PROV-Dictionary mapping of `disti[j] = ikj`.

We divide these figures into three regions: **A** represents the base part assignment that exists in all approaches; **B** represents a portion of the matrix that existed before this operation; and **C** represents the overhead entities (i.e., entities that are specific to an approach) that were generated as consequence of the part assignment. Note that Fig. 4 has no region C since Versioned-PROV does not have overhead entities. All the entities that exist in Versioned-PROV also exist in the other approaches.

We also use the color red to denote the overhead. Note that plain PROV has a bigger overhead than PROV-Dictionary, which has a bigger overhead than Versioned-PROV. This occurs due to the problems P1 and P2 mentioned in the introduction. Additionally, we use gray to indicate the portion of the provenance graph that is not related to the part assignment operation. As expected, all nodes and edges in region B are gray. The only gray node outside region B is `disti@j#2` in region A. This node appears due to the if condition in line 16 of Fig. 1. Hence, it is specific to this algorithm and not a generic node that occurs in all part assignments.

The operation `disti[j] = ikj` is putting the value of `ikj` into the position `j` of `disti`. In region A of all figures, `ikj#2` represents the variable `ikj`; `j#5` represents `j`; and `disti#2` represents `disti`. Additionally, `disti@j#3` represents the resulting `disti[j]`. Note that `disti` in this execution is the same list as `dist[2]`, represented by the entity `matrix2#1` (i.e., they point to the same memory area). Note also that `dist` and `result` are the same matrix.

Since entities are immutable in PROV and PROV-Dictionary, an update in a collection (`disti#2` in region A) requires the creation of a new collection (`disti#3` in region C) that contains the updated members. PROV suffers from P1, thus it reconstructs the membership of the new entity by using N *hadMember* relationships in a

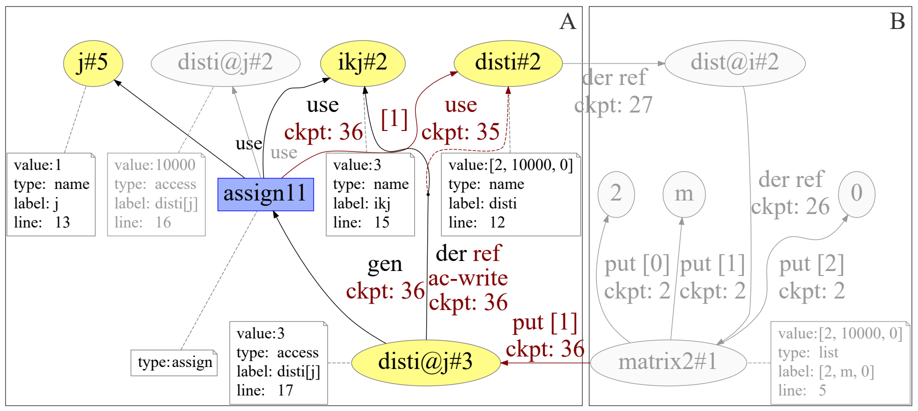


Fig. 4. Versioned-PROV mapping of `disti[j] = ikj`.

collection with N members (3 in this case). We represent these relationships by edges without labels in Fig. 2. PROV-Dictionary, on the other hand, uses a single *derivedByInsertionFrom* (`der-ins` edges in Fig. 3) to indicate that a collection was updated by the insertion of a member at a position (`disti#3` derived from `disti#2` by the insertion of `disti@j#3` from region A at position 1).

As stated before, `disti#2` represents the same value as `matrix2#1`. Thus, we would have to update `matrix2#1` to reflect the change. This does not occur because `matrix2#1` is out of the scope of the execution at this point and cannot be directly used without an access to `dist#1` or `result#1`. Due to P2, plain PROV and PROV-Dictionary update `dist#1` and `result#1` by generating `dist#2` and `result#2` in region C and replacing `matrix2#1`, in the second position, by `disti#3`.

In addition to this overhead in PROV and PROV-Dictionary, we use two extra *wasDerivedFrom* edges for every new collection entity to indicate that they derive both from the collection before the update and from the inserted value (`ikj#2`). Thus, in PROV, this operation has an overhead of 3 *entities*, 6 *wasDerivedFrom*, and 9 *hadMember*, and in PROV-Dictionary, this operation has an overhead of 3 *entities*, 6 *wasDerivedFrom*, and 3 *derivedByInsertionFrom*. Moreover, these overheads depend on the number of elements in the collections and the number of references to them.

Versioned-PROV does not suffer from these problems. It uses checkpoints to indicate multiple versions of a collection, and *derivations by reference* to indicate that two or more entities represent the same collection. In region C of Fig. 4, `matrix2#1` was defined at checkpoint `2` with the entities `2`, `m`, `0` as members. This changed at checkpoint `36` since this part assignment put `disti@j#3` in the first position. Thus, `matrix2#1` has a version with the members `2`, `m`, `0` between checkpoints `2` and `35`, and a version with the members `2`, `disti@j#3`, `0` after checkpoint `36`. Note that in Fig. 4 we show the first value representation of collections for easy reading, but other Versioned-PROV implementations are free to decide on having the *value* attribute or not.

The aforementioned versions are valid for all the entities that derive by reference from `matrix2#1`. In Fig. 4, `dist@i#2` derived by reference from `matrix2#1`, and `disti#2` derived by reference from `dist@i#2`. By transitivity, `disti#2` derived by reference from `matrix2#1`. This derivation avoids the creation of `disti#3` and all the other entities and relationships that exist in the other mappings.

Since an entity can represent multiple versions of a collection in Versioned-PROV, we also use the *checkpoint* attribute in the use of `disti#2` to indicate the used version. Note in region A of Fig. 4 that this operation is using `disti#2` at checkpoint `35` to generate `disti@j#3` at checkpoint `36`.

Every entity can only be derived by a single reference: if the algorithm assigns a new value to the variable `disti` (in line 12 of Fig. 1), we must create a new entity (e.g., `disti#3`) as a placeholder for the new value. That is, the checkpoint attribute does not apply for reusing an entity with different values. A variable entity in Versioned-PROV represents not just the variable name, but a pair consisting of the variable name and its value (memory area). Note that we do not need a new entity for `disti#2` in the part assignment as it still references the same memory area after the operation.

Finally, `disti@j#3` derived by reference from `ikj#2` in region A of Fig. 4. Since these entities are not collections, the derivation by reference has no impact on them we use it just for consistency among all derivations. However, this specific derivation has other attributes in addition to *type* and *checkpoint*. We also indicate that it is a write *access* that puts the derived entity in the *key* position `1` of the collection `disti#2`. This information is required to answer the provenance query of *Floyd-Warshall* without encoding matrix positions into entities. Note that the members of `matrix2#1` in region B of Fig. 4 are the actual entities that exist in line 5 of Fig. 1, while the members of `matrix2#1` in Fig. 2 and Fig. 3 are dummy entities that encode the matrix position.

4 Evaluation

We evaluate the space overhead of Versioned-PROV in comparison to plain PROV and PROV-Dictionary by measuring the number or PROV-N statements each approach requires in similar situations. We analyze both the running example and the general case.

Space overhead analysis of the running example. For most operations, the storage requirements are the same in all three approaches. The only differences were observed in data structures definitions (lines 2-5 of Fig. 1), reference assignments or accesses (lines 2-5, 7, 9, 12, 18), and data structure updates (line 17).

In [13] we present the complete provenance graph of *Floyd-Warshall* in these three mappings, coloring only nodes and edges related to the list definitions, reference derivations, and part assignments, since these differ in the mappings. All nodes and edges that are common to all mappings are in light gray. PROV has many colored edges all over the graph due to the aforementioned problems P1 and P2. PROV-Dictionary has fewer scattered edges in the graph, but it has a huge concentration of Dictionary entities that derive from a single *EmptyDictionary* entity due to problem P2. Finally, Versioned-PROV has fewer colored nodes and edges since it does not suffer from these issues.

In Fig. 5(A) we count how many nodes are specific to each approach. Note that PROV and PROV-Dictionary use respectively 7.52 and 4.14 times the number of specific PROV-N statements used by Versioned-PROV to represent the same data structures. Additionally, Versioned-PROV does not impose any node overhead. All of its

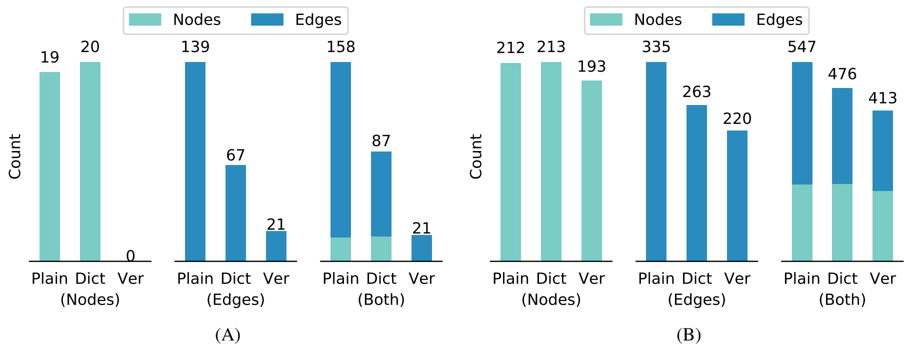


Fig. 5. Number of PROV, PROV-Dictionary, and Versioned-PROV PROV-N statements for list definitions, reference derivations, and part assignments (A) and total number of statements (B).

overhead occurs in edges that specify the membership of collections. On the other hand, PROV and PROV-Dictionary impose node overhead to indicate the position of elements in data structures and to derive immutable entities from existing ones. Moreover, by comparing Fig. 5(A) with Fig. 5(B), which shows the total number of statements, we can see that 29% of PROV statements, 18% of PROV-Dictionary statements, and 5% of Versioned-PROV statements are the overhead caused by collection operations.

These results refer to a small *Floyd-Warshall* execution, with a 3×3 matrix representing the input graph. Since the overheads of PROV and PROV-Dictionary grow in terms of the number of collection elements and the number of shared references, more complex input graphs and algorithms can cause a much larger overhead.

Space overhead analysis of the general case. In Section 3, we describe the part assignment of PROV, PROV-Dictionary, and Versioned-PROV. Fig. 6 presents the growth of statements in the three approaches for part assignments. Versioned-PROV has an overhead of **2 PROV-N statements**: the *hadMember* that puts the member in the collection, and the *used* that indicates the changed collection. Plain PROV has an overhead of $\boldsymbol{(3 + N) \times R}$ **statements** for collections with N members and R references: it creates R *entities*, each of them with 2 *wasDerivedFrom* and N *hadMember*. Finally, PROV-Dictionary has an overhead of $\boldsymbol{4 \times R}$ **statements**: it creates R *entities*, each with 2 *wasDerivedFrom* and 1 *derivedByInsertionFrom*. Note that both plain PROV and PROV-Dictionary also use the changed collection, but this *used* relationship can be inferred from one of the additional *wasDerivedFrom* statements. Hence, we count it only as an overhead for Versioned-PROV. The number of statements for PROV and PROV-Dictionary are lower bounds. If we update a collection `x` that is also a member of another collection `y`, we must also update all the references of `y` and apply this same rule with respect to references and number of elements. This occurs in our example of Section 3.2: the update of `disti#2` with $R = 1$ and $N = 3$ motivates the update of `dist#1` with $R = 2$ and $N = 3$.

Besides part assignments, the approaches also differ in list definitions and derivations by reference. Fig. 7(A) shows the overhead of defining a list in each approach.

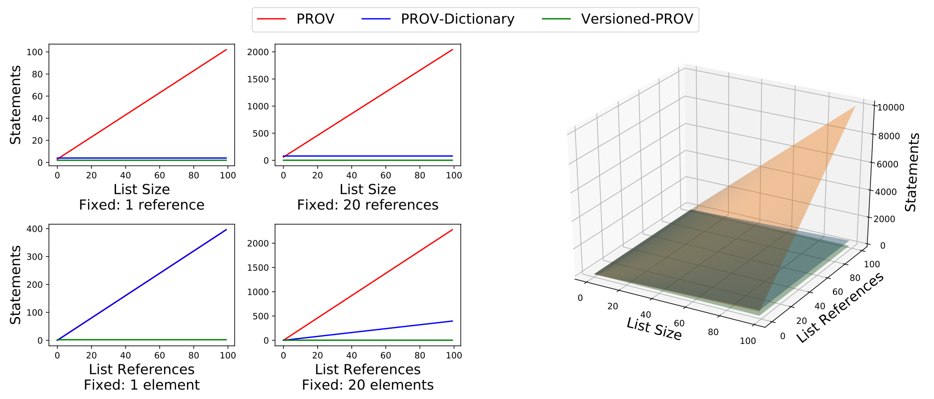


Fig. 6. Overhead functions of part assignments.

Versioned-PROV has an overhead of only N ***hadMember* statements** to define a list with N elements since they indicate the members with their positions in the list and we reference these positions in accesses. Thus, the provenance of *Floyd-Warshall* in Versioned-PROV includes the accessed positions, allowing us to use these positions to reconstruct the paths of the graph.

On the other hand, plain PROV and PROV-Dictionary have overheads of $\mathbf{3 \times N + 2}$ **statements**, and **1 (global)** $\mathbf{+2 \times N + 3}$ **statements**, respectively. This occurs because these approaches do not indicate the access position and the access derivation directly from the member. Hence, we must encode the position information into entities. This encoding requires the creation of N dummy entities. Each one of these dummy entities derives from their respective entities (i.e., N *wasDerivedFrom*) by the application of a new *definelist* activity. The resulting list entity is also generated by this activity (i.e., 1 *wasGeneratedBy* and 1 list *entity* itself), and it has the dummy entities as members. PROV-Dictionary expresses the membership with a single *derivedByInsertionFrom* statement from a single global *EmptyDictionary*, while PROV additionally requires N *hadMember* statements to define the membership of all elements.

Fig. 7(B) compares the growth of overhead in derivations by reference. Versioned-PROV imposes **no statement overhead** since it uses attributes of *wasDerivedFrom* to indicate the derivation. On the other hand, PROV and PROV-Dictionary have to recreate the membership of this new entity. PROV requires N ***hadMember* statements**, and PROV-Dictionary requires a **single *derivedByInsertionFrom* statement**. Note that both PROV-Dictionary and Versioned-PROV do not grow in terms of the number of elements, but Versioned-PROV still performs better than PROV-Dictionary, since the former does not require any extra statement.

5 Related Work

Many approaches have been proposed to collect and represent provenance from scripts. Some tools export provenance from scripts to OPM [1, 16], which is easily convertible to PROV. However, these tools work at coarse-grain and do not take mutable data

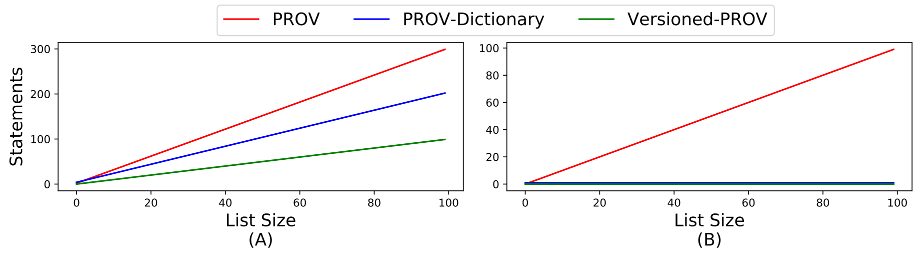


Fig. 7. Overhead functions for list definitions (A) and derivations by reference (B).

structures into account. Other tools work at fine-grain but use non-interoperable mechanisms for storage and distribution [5, 14, 15]. Moreover, these approaches work only at the variable and statement dependency level and do not provide support for tracking the provenance of changes on data structures referred by multiple variables.

Michaelides et al. [7] collect fine-grained provenance from Blockly variables and export it to plain PROV. Plain PROV assumes that entities are immutable and uses *hadMember* statements to describe structures, but its usage is too verbose and imposes a high overhead in the storage of mutable data structures, as we present in Section 4.

PROV has been extended in many different ways [2, 4, 9], but most extensions focus only on representing domain-specific provenance and do not improve the support for data structures. The PROV-Dictionary extension [8] improves the PROV support for data structures by adding insertion and removal derivations. Such derivations reduce the storage overhead in comparison to PROV, but still produces a high overhead in comparison to Versioned-PROV due to the assumption of immutability.

6 Final Remarks

In this paper, we propose Versioned-PROV, a PROV extension that supports mutable data structures. Tools that collect fine-grained provenance from scripts can use Versioned-PROV to support the collection of provenance from complex data structures and variables that are implicitly modified due to the existence of other variables pointing to the same mutable data. Nevertheless, our extension is not restricted to scripts.

The proposed approach has some limitations. First, while our extension reduces the storage overhead for provenance collection from scripts, it introduces an extra overhead for querying due to the requirement of unfolding data structure versions based on checkpoints. Thus, users must consider this tradeoff according to their needs. Second, by using a dictionary-like structure to represent lists (i.e., indexes mapped to keys, and elements mapped to values), some operations still produce an overhead in the provenance storage. For instance, inserting an element at the beginning of a list will require updating all the other members of the list. Third, using an explicit checkpoint ordering imposes synchronization challenges for parallel provenance collection. Finally, the usage of optional attributes to extend PROV imposes a storage overhead in disk due to the attribute name repetition. However, this overhead may not occur depending on how it is stored. A normalized storage schema would remove the repetitions.

As future work, we intend to develop an efficient querying algorithm for Versioned-PROV. We also plan to adopt the proposed model in noWorkflow [12] to export its fine-grained provenance [14] and evaluate it in real scenarios. We foresee the elaboration of unfolding algorithms that converts Versioned-PROV into plain PROV to improve its interoperability and optimize analyses that require many queries. These algorithms could also run by demand, populating caches of unfolded data structures. Additionally, we plan to work on an extension of Versioned-PROV to improve the incremental membership definition of lists.

Finally, our companion website [13] contains all the source code used to generate images of this paper in addition to detailed descriptions of the mapping we applied in each approach, as well as a preliminary query implementation.

References

1. Angelino, E., et al.: StarFlow: A script-centric data analysis environment. In: IPAW. pp. 236–250. Springer Berlin Heudelberg, Troy, USA (2010)
2. Costa, F., et al.: Capturing and querying workflow runtime provenance with ProV: a practical approach. In: Joint EDBT/ICDT Workshops. ACM, Genoa, Italy (2013)
3. Floyd, R.W.: Algorithm 97: shortest path. Commun. ACM. **5**(6), 345 (1962)
4. Garijo, D., Gil, Y.: Augmenting PROV with Plans in P-PLAN: Scientific Processes as Linked Data. In: LISC. Boston, USA (2012)
5. Lerner, B., et al.: Using Introspection to Collect Provenance in R. Informatics **5**(1), 12 (2018)
6. Luc Moreau, Paolo Missier: PROV-DM: The PROV Data Model (2012), http://www.w3.org/TR/prov-dm/
7. Michaelides, D.T., et al.: Intermediate Notation for Provenance and Workflow Reproducibility. In: IPAW. pp. 83–94. Springer Cham, McLean, USA (2016)
8. Missier, P., et al.: PROV-Dictionary: Modeling Provenance for Dictionary Data Structures, https://www.w3.org/TR/prov-dictionary/
9. Missier, P., et al.: D-PROV: Extending the PROV Provenance Model with Workflow Structure. In: TaPP. USENIX, Lombard, USA (2013)
10. Moreau, L., et al.: The Open Provenance Model: An Overview. In: IPAW. vol. 5272, pp. 323–326. Springer Berlin Heidelberg, Salt Lake City, USA (2008)
11. Muniswamy-Reddy, K.K., et al.: Provenance-Aware Storage Systems. In: USENIX Annual Technical Conference. pp. 43–56. USENIX, Boston, USA (2006)
12. Murta, L.G.P., et al.: noWorkflow: Capturing and Analyzing Provenance of Scripts. In: IPAW. pp. 71–83. Springer Cham, Cologne, Germany (2014)
13. Pimentel, J.F., et al.: Versioned-PROV, https://dew-uff.github.io/versioned-prov/
14. Pimentel, J.F., et al.: Fine-grained Provenance Collection over Scripts Through Program Slicing. In: IPAW. Springer Cham, McLean, USA (2016)
15. Runnalls, A., Silles, C.: Provenance tracking in R. In: IPAW. pp. 237–239. Springer Berlin Heidelberg, Santa Barbara, USA (2012)
16. Tariq, D., et al.: Towards Automated Collection of Application-level Data Provenance. In: TaPP. USENIX, Boston, USA (2012)