Fast Capture of Cell-Level Provenance in Numpy

Jinjin Zhao [1] Sanjay Krishnan [1]

[1]University of Chicago

Introduction and Motivation

Capturing cell-level provenance from array operations can significantly improve the governance, reproducibility, and overall quality of data science pipelines. However, existing data systems still face significant challenges in automatically capturing cell-level provenance.

1. **Changing APIs.** The shift from pandas version 1.x to 2.x introduced significant changes, including the removal of the 'inplace=True' parameter in certain methods, affecting provenance tracking strategies. Any provenance capture system must be flexible enough to adapt seamlessly to evolving APIs.
2. **Diverse Operations.** Data pipelines frequently involve diverse operations such as complex reshaping (e.g., NumPy's 'reshape' and 'transpose' functions), convolution functions (e.g., pandas' 'apply' method), and transformations utilizing machine learning models (e.g., predictions from scikit-learn models). Capturing provenance across these varied operations requires a more generalized and flexible provenance model.
3. **Scale of Datasets.** Many modern datasets, i.e. large-scale climate sensor datasets, often contain extremely large arrays - on the order of millions of cells. Efficient cell-level provenance capture mechanisms must maintain performance at this scale.

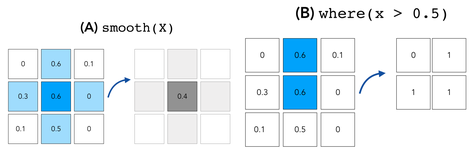

Figure 1. **Motivating Example**. Visual illustration of two non-standard array operations in imaging and scientific applications.

We present **a prototype cell-level provenance capture tool for the `numpy` library** to address these issues. It captures provenance by annotating over core low-level operations [addressing **(1)**, **(2)**], and scales well up to 100 million cells [addressing **(3)**].

Provenance Capture Workflow

We can show a typical example of how to capture provenance with this prototype.

```
(1) array = array.astype(CellProvenance)
(2) array.initialize()
(3) array = -array # negative unitary operation
(4) first_cell_provenance = array[0,0].parents
```

Figure 2. Example of provenance capture.

Explanation of Steps

1. Before an operation is performed, we first create an array using this data type.
2. Next, we initialize provenance tracking with the `initialize()` function.
3. We can perform typical `numpy` transformations on the array.
4. After the transformation, the `parents` property of every output cell contains the cells indices of the starting array that contributed to its value.

Abstract

Significant challenges arise in capturing cell provenance of array operations, including: (1) rapidly evolving APIs, (2) diverse operation types, and (3) large-scale datasets.

To address these challenges, our work presents a prototype annotation system designed for arrays, capturing cell-level provenance specifically within the `numpy` library.

With this prototype, we explore straightforward memory optimizations that substantially reduce annotation latency.

This tool is part of the larger DSLog system for array provenance management [ICDE'24].

Basic Provenance Capture Internals

`CellProvenance` contains typical scalar data and a list of cell indices indicating dependencies from previous cells

```
# annotated data type
class CellProvenance:
    data_value: Any
    parents: List[Indices]
```

The `parents` property is initialized with the original array indices; for instance, the top-left cell in the array is annotated with the index (0, 0).

```
# provenance initialization
c.parents = [(c.index1, c.index1)]
```

Each array operation in the `numpy` library is extended so that the `parents` property of the output value becomes the union of the `parents` properties from the input values. This effectively captures a type of provenance for all numpy operations.

```
# provenance operation
c.parents = Union(a.parents, b.parents)
```

Low-Level Memory Management

To overcome performance limitations, we introduce the `tracked_float` data type.

Each provenance annotation consists of three 32 bit integers: the first indicates the array ID, and the second and third represent array indices, uniquely identifying cells for arrays with up to two dimensions (though this strategy generalizes to higher dimensions).

These tuples are stored in a C array, with the first annotation directly embedded in the `annotated_cell`, and subsequent annotations stored as pointers to dynamically allocated memory buffers.

To implement provenance capture for all `numpy` operations using this data structure, we only need to adjust two primitive data operations. For unary operations, we copy the `prov_id` and pointer values to the output. For binary operations, we concatenate the `prov_id` arrays from both inputs.

Figure 3. Diagram of DSLog's tracked_float memory structure.

Microbenchmark Experiment Results

We compare DSLog against Python-only and C without buffers baselines to demonstrate the relative improvements provided by our performance optimizations. We measure the execution time cost of initializing a tracked array, and overhead of provenance capture over common data science transformation patterns. These experiments are performed on arrays up to **100 million cells**.

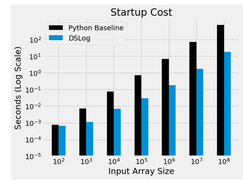

Figure 4. Startup cost of provenance annotation.

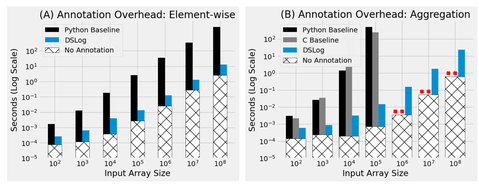

Figure 5. Execution overhead of provenance annotation on two different operation patterns.

Although capturing provenance at the cell level inherently carries performance overhead, we have shown that **careful memory management can significantly mitigate provenance capture costs compared to naive implementations**. DSLog adds less than 50 seconds of overhead on aggregate operations on 100 million array cells.

Vision and Future Work

[Vision] A Universal Provenance System. Within a single data science project, data frequently transitions between these structures. We can extend current ideas to capture provenance across different data structures for comprehensive end-to-end cell-level provenance tracking for all data transformations in a project. This raises broad research questions on:

- Privacy Risks of Centralized Provenance Governance
- Provenance as Proof-of-Work
- Provenance for Data Science Automation and Debugging

[Future Work] Optimizing Provenance Capture. Additional opportunities exist to enhance the performance of provenance capture in DSLog. Some techniques include:

- Parallelize Cell Capture
- Improved Predictive Memory Allocation

[Future Work] Provenance Capture with Uncertainty. Can DSLog capture incomplete or uncertain provenance, and how could that information be useful?