DSLog: A Provenance Storage Manager for Data Science

Anonymous

ABSTRACT

Tracking data provenance is important for ensuring data integrity, reproducibility, and debugging data science workflows. However, even for the smallest datasets, fine-grained provenance is difficult to capture and store. This paper presents DSLog, a storage system that efficiently stores, indexes and queries array data provenance that is agnostic to capture methodology. We contribute a new compression algorithm, named ProvRC, that compresses the captured relationships. We show that the ProvRC results in a significant storage reduction over functions with simple spatial regularity, beating alternative columnar-store baselines by up to 700x. We also show that ProvRC facilitates in-situ query processing that allows for forward and backward provenance queries without full decompression; in the optimal case, surpassing baselines by 200x in query latency. Finally, we introduce shape-based and generalized re-use of provenance over repeated calls of the same function, and demonstrate that these two re-use methods cover 92% and 73%, respectively, of tested numpy functions .

ACM Reference Format:
Anonymous. 2022. DSLog: A Provenance Storage Manager for Data Science. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Data provenance is the documentation of where a piece of data comes from and the computational operations by which it was produced. As data scientists work with ever more complex data workflows, data provenance is a crucial tool to ensure the reproducibility and reliability of results. The most popular frameworks today only support coarse-grained provenance (CGP); that is, they track input-to-output data relationships at the granularity of entire datasets. Examples of CGP systems include Vamsa [18], Juneau [42], and ReproZip [28].

Unfortunately, CGP systems hit a ceiling of utility. While such tools are efficient and easy to deploy, they are only useful for coarse tasks such as determining the datasets needed to reproduce a data product. In contrast, fine-grained provenance (FGP) systems track input-to-output relationships at a record, data-cell, or data subgroup level. FGP systems are additionally valuable for debugging faulty workflows, enforcing regulatory conditions like the right-to-be-forgotten, and comparing differing workflows over the same data. However, today, FGP systems are limited by scale. They incur significant provenance capture overheads, high costs of storing the fine-grained relationships, and significant retrieval costs when a user wishes to use the captured provenance. For example, a raw table storage of provenance for a matrix operation of 1000 by 1000 cells can be approximately 40GB (see Experiments).

As a step towards improving the scalability of FGP systems, this paper contributes a new service called a *data provenance storage manager* (DPSM). A DPSM is a centralized data repository that stores and indexes provenance relationships captured from an organization's data science programs. Each record in this repository is an input-to-output contribution relationship, i.e., a statement that some unit of input data contributes to a unit of output data. Over this repository, the storage manager must support two types of queries across multiple steps of the workflow: forward queries (i.e., find all output data contributed to by an input), and backward queries (i.e., find all input data contributing to an output).

This paper contributes a DPSM prototype called DSLog. DSLog provides an API to ingest captured data provenance from data analytics programs, and it stores this provenance in a central data repository. The API focuses on data science workflows and accordingly assumes data that are structured in multi-dimensional arrays (or data types the resemble arrays). Integer-indexed multi-dimensional arrays are a sort of universal interface in data science that can represent dataframes, documents, time series, and images. DSLog contributes a number of novel architectural insights: (1) it stores the captured provenance in a normalized relational format; (2) individual provenance relationships are encoded in a novel compressed representation; and (3) the storage format allows for fast in-situ query processing of forward and backward queries.

The compressed provenance encoding is the main algorithmic contribution of this work. Even relatively simple workflows can generate large amounts of provenance data. Luckily, we find that this provenance is highly compressible. The provenance relationships of common data science operations have a high degree of spatial coherence where nearby input indices often contribute to similar output indices, which can be exploited by a novel multi-dimensional range-encoding algorithm that efficiently represents long ranges of indices as intervals. We introduce the ProvRC compression algorithm which uses multi-attribute range encoding and relative indexing to leverage spatial regularities in a relational representation of provenance (Section 3.3).

In implementation we integrate DSLog with the Python data science stack. DSLog is agnostic to exact semantics of the provenance it stores, and is compatible with many different definitions. Accordingly, we take an inclusive approach in our prototype and integrate three different provenance capture methodologies. First, DSLog can capture cell-level provenance in numpy using a "taint tracking" approach that logs accesses when it is manipulated by any numerical primitive operations. It further integrates with explainable AI techniques that can identify feature-level importance in machine learning inference components (which we will show is just another provenance relationship). We finally implement function-specific provenance tracking methods as appropriate for relational operations. The prototype is able to scale to fine-grained provenance capture to arrays with 1 billion cells. Our research contribution describes that ProvRC results in very large storage reduction over functions with simple spatial regularity with an approximately 0.5% storage ratio over raw storage, and beating alternative columnar-store baselines by up to 700x. . We also demonstrate significant performance improvements in the downstream query processing,

Conference'17, July 2017, Washington, DC, USA
2022. ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

performing queries over 100 million input cells in <1 second in the optimal case, beating the same baselines by 2000x.

2 BACKGROUND

First, we motivate our problem setting and describe the related work.

2.1 Problem Setting and Scope

An *array* of data is a rectangular collection of data indexed by tuples of positive integers. Each value in an array is called a *cell*. We denote arrays with the square bracket notation, for example, the $(0, 1)$ cell of array X is denoted by $X[0, 1]$. The length of the tuples indexing an array define its *shape*. Arrays can represent any ordered data structure, and act as a useful universal data type for data science operations.

- A dataset of machine learning examples can be represented as a 2D array of feature vectors, with each cell represented by a floating point number.
- An image can be represented as a 3D array of R,G,B pixels each represented by a unsigned integer 0-255.
- A document can be represented as a 1D array of strings.
- A relational table can be represented as a 2D array where the rows and columns are organized in a canonical order.

An array program is simply a function that takes as input a set of input arrays and outputs another array. For example,

```
Y = foo(X1,...,XN)
```

the function `foo` applies to arrays `X1,...,XN`, and outputs another array `Y`. *Fine-grained data provenance* refers to contribution relationships between input and output cells in such operations: which elements in the input array contribute to the values of each output element. The precise semantics of what "contribution" means can differ between provenance frameworks, and the type of analysis that the user is doing. For example, in a pre-processing workflow that aggregates cells, we might care about strict definition of provenance that captures all the cells that are aggregated to produce an output. On the other hand, in a machine learning inference pipeline, we might care about softer definition of contribution that considers feature-importance. Regardless of the definition of a contribution relationship, we are interested in a storage interface that can store, index, and optimize queries over such relationships.

Mathematically, we think of such contribution relationships as forming a bipartite graph that relates output cells in Y to a set of contributing input cells in each X_i with edges of the form:

$$Y[b_1, ..., b_l] \leftarrow X_i[a_1, ..., a_k]$$

The set of all such edges for a given program, such as `foo` above, is called the *provenance graph*.

It is not enough to simply represent such graphs; we also need to be able to answer queries over them. A forward query finds all output cells that a particular input contributed to. Likewise, a backward query finds all input cells that contributed to a particular output.

Forward and backward queries can also span across compositions. For example, consider the array program:

```
Y = bar(foo(X1,...,XN))
```

One might have a provenance graph for `foo` and `bar`, and have to compose them to reason about the relationship between the input `X`'s and the outputs `Y`.

2.2 Capturing Provenance

There has been significant work in defining and discovering fine-grained provenance in different contexts. As new technologies emerge for data science and data management, new systems have been proposed for provenance capture. Therefore, we believe that a DPSM should allow the ingestion of cell-to-cell provenance independent of the capture methodology or the application-specific definitions of provenance. In this section, we briefly overview and classify existing work.

Capture Approach 1. Domain-Specific Languages. One approach is to have a closed-world of operations that an analyst might use through a domain-specific language or pattern matching known operations. For example, the Lima system tracks provenance for machine learning workflows written in SystemDS [23]. Similarly, tools like MLInspect [13] use pattern matching to identify certain function structures in the Python AST. Several other systems employ a similar approach [3, 19]. Other work limits the scope of the system to particular types of workflows such as machine learning model training or inference [29, 31, 32, 36, 40, 43].

Capture Approach 2. Operation Specification. Of course, one could always model unknown functions as all-to-all relationships where all input elements contribute to each output element [12, 17]. To capture finer-grained information, a complementary approach is to define specification API that allows a user to define their own provenance relationships to cover missing user-defined functions. For example, SubZero provides an extensive API that users can use to seamlessly capture provenance for new, user-defined functions [37]. While more flexible than an operation tracking approach, operation specification can be hard to deploy in certain settings. The provenance relationships of a particular function can be dependent on hyperparameter settings, and describing the specification might be as hard as developing the function itself.

Capture Approach 3. Annotated Execution/Information Flow Tracking. Other system automatically capture and logs data type-level accesses by tracking the read, write, and copy operations made to individual data elements and correlating these accesses with the system call stack [34, 35]. The user does not need to do anything to use such an approach as all tracking happens at a low-level. The key downsides are the computational overheads from such tracking and the storage overhead from low-level physical tracking.

Capture Approach 4. Explainable AI. We can even consider model explanation algorithms as type of provenance. These algorithms often define a contribution relationship between input features in an example and an output prediction [4]. Such relationships can often be expressed as (weighted) provenance graph edges. Converting such data into provenance information allows a DPSM to store and index this information.

DSLog: Missing Piece in Provenance. A single repository that can capture and store different types of provenance allows for new types of introspection into data manipulation operations. For example, we might be interested in determining how records that

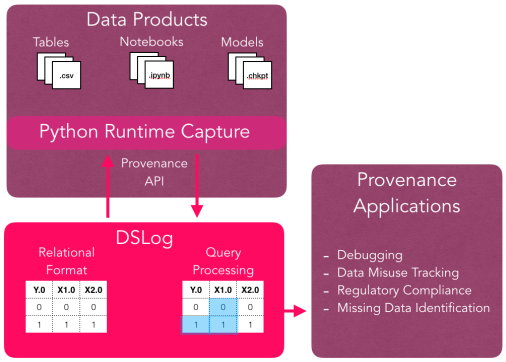

Figure 1: DSLog contributes a new service called a data provenance storage manager, which stores and indexes provenance captures from across a data science workflow. It acts as an intermediary between the data science applications (those that generate provenance) and the provenance applications (those that use provenance).

have be clipped with an outlier removal algorithm contribute downstream to a machine learning inference task. Such a query requires integrating different definitions of provenance. DSLog provides an interoperable interface that links these graphs together to answer forward and backward queries (Figure 1).

From a technical perspective, the graph storage and serving is an under-appreciated part of the design of provenance systems. For even simple programs, the size of a fine-grained provenance capture can be very large. A DPSM is a service that stores provenance graphs, serves backward and forward queries, and has domain-specific optimizations to make such queries efficient. We find that these optimizations are not present in off-the-shelf graph and relational data models, which have been employed in prior work [15].

As for a basic intuition, we note that array operations often have spatial locality where nearby indices have similar contribution relationships. A typical graph database would not be able to exploit such spatial correlations. Additionally, users often repeatedly execute the same or similar workflows, there will be a large number of nearly identical provenance graphs. DSLog is able to use these observations to optimize the storage size and query performance in a manner that a standard graph model of provenance is incapable of.

2.3 Related Work

To the best of our knowledge, a systematic exploration of DPSM design is absent from the literature. Existing system work takes a pragmatic to provenance storage. For example, the Ground [15] system uses a Postgres SQL engine with a row-oriented data organization. Similarly, Smoke uses an in-memory data structure to store and index provenance for visualization applications [26]. We show that the typical structures in array-provenance can be stored and indexed in a compressed relational format as opposed to a raw relational format. This relational format allows us to leverage any database backend, and scale up provenance capture beyond the scope of a single program.

There is some prior work in provenance compression. Xie et al. studied compression provenance graphs for web search and semantic web applications [38, 39]. While theoretically applicable in our scenario, such approaches do not exploit the numerical regularity of array indices in typical array-programming applications. Thus, the types of encodings used are ill-suited for our scenario. For hierarchical data, Chapman et al. proposed a "factorized" provenance approach [5]. This approach identifies common nodes in graph and combines relationships when possible. Mathematically, this can be interpreted as a polynomial factorization of the provenance polynomial [21]. It turns to that this approach has a deep connection to ours, and many of our optimizations can be interpreted as relational factorization. However, the focus on array programming gives us specific spatial patterns that we can exploit beyond simple factorizations. Compression has also been studied in network provenance problems [7, 44]. Again, due to differences in the problem setting, they are not directly comparable to DSLog. We leverage existing ideas from columnar compression [2] and in-situ query processing [25] to build our query processing framework.

Since our focus is on array programs, there is also a tangentially related field of machine learning workflow management. Initial prototypes such as HELIX [40], Alpine Meadow [33], MLFlow [41], and the Collaborative Optimizer [10] rely on coarse-grained lineage tracing at the level of entire machine learning programs. Such systems are useful for versioning and dataset discovery, but fail to give fine-grained information about how individual data values, i.e., individual array elements, contribute final results.

3 DSLOG: SYSTEM ARCHITECTURE

DSLog provides an API to ingest captured data provenance from data analytics programs, and it stores this provenance in a central data repository. The main architecture of DSLog is independent of the capture methodology and precise definition of provenance, but enforces arrays as the capture data type.

3.1 Core DSLog API

First, we present the core API that allows the ingestion and query of provenance data.

Defining Tracked Arrays. DSLog allows users to define named arrays with a specified shape (range of values that the index is allowed to take). These are the arrays that are tracked through a data science workflow.

```
Array(name: String, shape: Long [])
//defines an array with the specified shape
```

To record provenance between input and output arrays in our system, we require that the user must specify a capture object. A capture object is an iterator where for each cell in the output array, we can retrieve the spatial indices (of the cells) in the input array that have a relationship.

```
Provenance(arr1: String, arr2: String, capture: Object)
//defines the provenance relationship between two arrays
//capture(i: Long []) returns cells in arr2 that are
     linked to the cell in arr1 at index i
```

Users specify an input cell, an output cell, and the capture object. The user's desired capture object will be called internally to populate the storage manager.

```
X = numpy.array([[0,3],[1,5], [2,1]])
Y = X[1:3,1] #(A)
Y = X[:,0] #(B)
Y = np.sum(X,axis=0) #(C)
```

(B) Slicing

b_1	a_1	a_2
1	2	2
2	3	2

(B) Projection

b_1	a_1	a_2
1	1	1
2	2	1
3	3	1

(C) Sum Along Axis

b_1	a_1	a_2
1	1	1
1	1	2
2	2	1
2	2	2
3	3	1
3	3	2

Table 1: For three example operations (A,B,C), we present the provenance relations.

Queries Over Provenance. The user can issue any forward and backward queries to the storage manager. Forward and backward queries can be expressed in a single query interface.

```
prov_query(X: String [], query_cells: Long [[]])
//retrieves provenance along path of X
//query_cells is a list of cell ranges in X[1] to
     retrieve provenance from
```

The user first specifies some query path, which is a sequence of arrays. For example, consider the functions `Z = bar(Y)` and `Y = foo(X)`. One possible query path is $X \rightarrow Y \rightarrow Z$, which defines a forward query from the cells of X to that of Z. An example backward query is $Z \rightarrow Y$, which defines a query from the cells in Z to that of Y. This sequence must be connected by some edges in DSLog. Thus, the function `prov_query` retrieves the cells in the paths of $X : [X_1, \ldots X_n]$ that are linked to specified cells (`query_cells`).

3.2 Prototype Capture Algorithms

In DSLog, we include a number of useful baseline provenance capture algorithms for the Python data science ecosystem. Figure 2 illustrates how supporting multiple definitions of provenance can help data scientists.

3.2.1 Cell-Level Tracking in `numpy`. `numpy` is a popular array manipulation library in Python. We use track how cells of data move through `numpy` operations. We have implemented a data type `tracked_cell` that associates provenance with every cell.. Under the hood, this capture technique is similar in semantics to taint tracking [45]. With efficient memory management, `tracked_cell` performs **TODO**x over the equivalent Python implementation, and scales to (and over) arrays with 1 billion cells. The output track of this datatype can be inputted into as the `capture` argument in a `Provenance` object. Detailed capture performance experiments and further descriptions are included in the Appendix of the full paper [1].

3.2.2 Explainable AI Tracking. DSLog also can populate the DPSM with edges derived from model explanation algorithms. DSLog assumes that models take the form $y = \texttt{model}(x)$, where x is an input example a (vector, 1-d array) and y is the predicted label (a vector, 1-d array). DSLog applies explainable AI techniques - LIME or D-RISE - over the model [22, 30]. Both model generate a weighted significance graph between the cells in x and the cells in y. Edges can be filtered from this graph based on a significance threshold.

3.2.3 Provenance Re-use. DSLog also enables the option to capture provenance of multiple instances of the same operation. Instead of registering provenance pairs, there is a function registration API that generates provenance based on the data science operation executed.

```
register_function(function_name: String, input_arrays:
     String [], output_arrays: String [], capture: Object
      [], func_arguments: Any[], re-use: Optional String)
//registers an operation where provenance is captured
     between input and output arrays
```

This function internally generates a Provenance object between linked input and output pairs, and registers a record between those provenance objects and the function metadata. Now, if the `re-use` argument is set, in future registrations of the function, the system tries to generate the new capture objects based on previous calls, without requiring the default capture method. A full description how this is calculated internally is described in Section 6.

3.3 Relational Data Model

DSLog provides a unified framework for representing graphs captured from these different methods. Every provenance relationship can be thought of as an edge in a bipartite graph mapping input elements to output elements $Y[b_1, ..., b_l] \leftarrow X[a_1, ..., a_k]$. Such a graph is elegantly represented as a relation. Let X be an m-dimensional input array whose axes are denoted with $a_1, a_2, ..., a_m$. Similarly, the output array, Y, is an l-dimensional array whose axes are denoted with $b_1, b_2, ..., b_l$. The dependency relationships between X and Y can be represented in a relation over the dimensions:

$$R(b_1, b_2, ..., b_l, a_1, a_2, ..., a_m)$$

Each row in the relation corresponds to a single output dependence $Y[b_1, b_2, ..., b_l] \leftarrow X_i[a_1, a_2, ..., a_m]$. In the API above, each distinct `Provenance` object defines a new relational table. Records in these tables are generated during initialization from internal capture calls.

As a concrete example, let us consider a simple array operation over a 3x2 array `Y = np.sum(X, axis = 1)`. This function uses the `numpy` syntax and sums over one of the axes of the 3 x 2 array. Table 1 shows a table representation of the provenance of `Y = np.sum(X, axis = 1)` **(C)**. The way to interpret this table is that each row represents a single contribution relationship, e.g., element at index $(0, 1)$ in the input contributes to element at index 0 in the output.

Forward and backwards queries can simply be cast as SQL predicates over such a table. For example, to find all output cells produced by (0,0), one simply evaluates the predicate $(a_1 = 1) \wedge (a_2 = 0)$. Likewise, to find all input cells that contribute to the output cell 0, one simply evaluates the predicate $(b_1 = 0)$.

More importantly, this format allows for compositional forward and reverse queries that span multiple operations. The provenance relationship over multiple operations can be reconstructed from

[1] Available to reviewers upon request

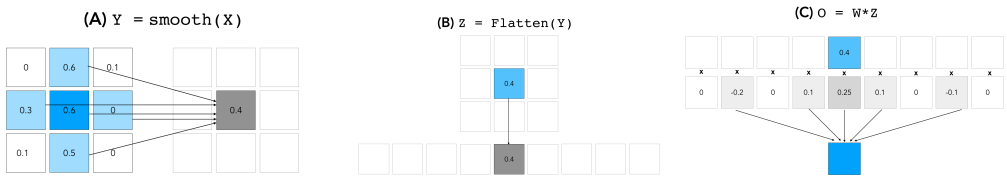

Figure 2: DSLog provides a unified interface to log provenance across a data science workflow. Consider a simple 3x3 imaging problem. (A) smooths the image with a mean filter, (B) flattens the smooth image, and (C) evaluates a linear model on the flattened image. DSLog allows for queries across the operations even if the definitions of provenance differ (for example, "Does the middle pixel contribute to the final result?").

these tables with a natural join operation, similar to that in [16]. Given R_f is provenance of $X \longrightarrow Y$ and R_g is the provenance of$Y \longrightarrow Z$, we can find $X \longrightarrow Z$ from $R_f \bowtie R_g$. While the basic ideas of relational representations of provenance graphs have been studied before [5, 16, 21], to the best of our knowledge such representations have not been used to support data science workflows.

3.3.1 The Power Of Data Independence. Relational modeling gives us a set of well-understood tools to optimize how the captured provenance is stored and indexed. We achieve a degree of data independence, where a user can still request forward and backwards provenance queries across multiple tables independent of exactly how the data are organized with lossless join decomposition.

Continuing with our `numpy` examples, Table 1 illustrates a handful of simple provenance examples over a 3x2 array. All of these examples illustrate non-trivial redundancy patterns. Table 1A shows that sums along axes create 1-to-1 relationships with the summation axis, and 1-to-all relationships with all other axes. Similarly, projection (Table 1C) and slicing (Table 1B) show similar functional relationships between output and input axes. An observant reader will note that these examples are all Multi-Valued Dependencies that can be decomposed using standard relational decompositions. In summary, a relational backend allows us to exploit the inherent redundancies in cell-to-cell provenance relationships. Relational databases give us a powerful set of modeling tools that to store such provenance captures in storage-efficient formats and lazily materialize the relations when the user requests them.

4 SEMANTIC COMPRESSION FOR PROVENANCE RELATIONS

In DSLog, the provenance between arrays is stored in a compressed range format that allows for in-situ forward and backward queries. Such compression has significant benefits in both storage and query performance In our experience, almost all data science operations exhibit spatial coherence, where nearby array indices are related to each other. This compression technique exploits such locality.

This section describe the `ProvRC` algorithm. `ProvRC` is optimized for answering backward queries: backward queries can be directly computed against the output of this compression algorithm. To enable forward queries, we additionally convert and material the output table in an alternative representation as needed. This design decision is not fundamental and is a workload bet that backward queries are more frequent than forward ones due to their prevalence in debugging tasks. To illustrate the key concepts, we will be using the running example from Table (Table 1C): a sum along axis.

4.1 Table Compression Algorithm

Our `ProvRC` algorithm has two main subroutines:

(1) Multi-Attribute Range Encoding. Express the relation as a union of multidimensional "ranges".
(2) Relative Value Transformation. Express input axes relative to a output axis to improve compression.

`ProvRC` is applied to provenance table as follows.

Step 1. Multi-Attribute Range Encoding over Inputs. The first steps is a generalization range encoding over input columns. This has been used before in many integer compression tasks from time-series compression to information retrieval [24]. The basic idea is given a set of integers one represents the set as a union ranges, e.g.,

$$\texttt{range}(\{1, 2, 3, 4, 9, 12, 13, 14, 15\}) = \{[1, 4], [9], [12, 15]\}.$$

We extend this basic principle to the multi-attribute setting. In the multi-attribute setting, this encoding can be thought of as decomposing a table into a union of Cartesian products of attribute ranges.

We apply a simple heuristic to construct such a multi-attribute range encoding that segments each column into ranges one at a time. We show the algorithm as operated over a relational table of provenance in the form of:

$$R(b_1, b_2, ..., b_l, a_1, a_2, ..., a_m)$$

R is first sorted in lexicographical order over the attributes $b_1, b_2, ..., b_l, a_1, a_2, ..., a_m$; we call the resulting ordered relational table S. Note that the order within subsequences $[a_1, a_2, ..., a_m]$ and $[b_1, b_2, ..., b_l]$ are arbitrary, but must be consistent with how we iterate through the those subsets in our algorithm.

Notationally, let s be a row in S and $s[c]$ be the projection of the row onto a set of attributes c (we will use C_R to denote all attributes in R). The basic primitive that we implement is a function which iterates through S and merges "contiguous" rows. A set of rows form a contiguous range if they match on all other attributes $C_R \setminus c$, and there exists at least one row for every value in the range at attribute c. Once constructed one can combine the rows together and replace them with a single row with attributes with the same value for the $C \setminus c$ attributes and a range for the c.

In the first step, for each input column, $a_i \in [a_m, \ldots a_1]$, we perform the algorithm over all rows $s \in S$ in order. In essence, this step compresses cases where a single output is dependent on rectangular blocks in the input array. The exact rectangular blocks formed depends on the original sort order of input attributes. Because S is sorted, we guarantee that we discover the minimal

Table 2: An example output of multi-attribute range compression for the provenance of (C)

b_1	a_1	a_2
1	1	1
1	1	2
2	2	1
2	2	2
3	3	1
3	3	2

→

b_1	a_1	a_2
1	1	[1, 2]
2	2	[1, 2]
3	3	[1, 2]

possible set of blocks. Table 2 shows an example output of this algorithm over the example C relational provenance representation.

Step 2. Relative Value Transformation and Output Range Encoding. In the second step, we apply a relative value transformation between input and output attributes. For every input attribute, $a_i, i \in [1, m]$, and output attribute, $b_j, j \in [1, l]$, we append a new column to S named $a_i b_j$ that contains the value $b_j - a_i$.

This transformation exploits the cases in provenance relations where, for a set of rows, X, and some input and output attributes, a_i, b_j, we have:

$$X[a_i] = X[b_j] - \delta$$

$$\delta = X[b_j] - X[a_i]$$

, for some constant δ. Now, there is a clear benefit of using a constant to represent a_i when performing RangeGroup- it possibly allows us to consider more rows to merge. However, since we do not know where this functional relationship exists in our given table, and we inclusively generate all possible relative columns between input and output columns. This is reasonable since the number of dimensions of captured arrays is generally small.

In the second step, we apply the multi-attribute range encoding algorithm over the output attributes $b_i \in [b_l, \ldots b_1]$. However, we add the modification that allow for matching on relative columns as well as the absolute columns. This generates the final table S'.

Our insight is that, for input array dimension, a_i, only one of the columns of $a_i, a_i b_1, \ldots a_i b_l$ is needed to determine the dimension index. Therefore, given two rows, r_1, r_2, we can merge the rows, given that $\forall a_i \in C_R$, a non-empty subset of $a_i, a_i b_1, \ldots a_i b_l$ are equal. If a functional relationship exists between one of the input and one of the output columns, that will be contained in the subset. Because of the subset step, we do not guarantee that we compress the ranges down a single output column optimally (unlike in Step 1).

Let's see an example of how this works. In Table 3, the values in a_1 is the same as b_1 for each row. Adding the column $a_1 b_1$ introduces an exact match in a_1 for the multi-attribute range encoding algorithm that didn't exist before. We can compress down the previous results to an even more concise representation.

We conclude with a brief visualization of Step 2. We are compressing rectangular input blocks generated in Step 1 into higher-level rectangles also include output attributes - with the addition that allow for relative coordinates to define each input block. Similar to Step 1, the exact rectangular blocks formed depends on the original sort order of output attributes, and now, they are also dependent on how the algorithm iterates through the rows of S (which is fixed in ProvRC).

Table 3: Output Relative Indexing Can Improve Opportunities for Multi-Attribute Range Encoding

b_1	a_1	a_2
1	1	[1, 2]
2	2	[1, 2]
3	3	[1, 2]

→

b_1	a_1	$a_1 b_1$	a_2	$a_2 b_1$
1	1	0	[1, 2]	[0, 1]
2	2	0	[1, 2]	[-1, 0]
3	3	0	[1, 2]	[-2, -1]

→

b_1	a_1	$a_1 b_1$	a_2	$a_2 b_1$
[1,3]	-	0	[1, 2]	-

4.2 Full Examples and Discussion

In Table 4, we can seen multiple examples of the compression over different numpy functions. In the input columns, the label in parenthesis indicates the relative column name used for representation (if relevant). We observe that the full ProvRC algorithm captures concise ranges between the input and output arrays, while preserving the input and output relational model. Not only is this an efficient compression schema, the parse-able format makes it suitable for downstream tasks such as pre-capture and queries. There are admittedly smaller benefits to the compression algorithm when the function do not exhibit regular patterns.

Notice that the ranges of each row represent a rectangle over the set of indices. As a matter of fact, we can view our provenance compression algorithm as an approximate solution to the "grid covering" problem. Finding the minimum covering is known to be hard in the number of dimensions (axes) [11]. Instead, the complexity of ProvRC is $O(ND + N \log N))$, where N is the number of rows and D is the number of dimensions. In most cases, we found that the provenance given by our annotation to be already sorted and the average-case complexity can be given as $O(ND)$.

4.3 Asymmetry in Representation

There is a little bit of an asymmetry where certain query paths are easier to answer depending on how the algorithm is run. We give a brief insight as to why backward queries can be directly made from the compressed representation without much work, but a slight tweak is needed to optimize forward queries. In practice, we can materialize two versions of the compressed tables, ones that optimize forward queries and ones that optimize backward ones, as needed.

In Table 3, the single row in the final table has the cell index range $[1, 3]$ for the attribute b_1. If we are interested in a backward query for the cell index range range $b_1 = [2, 4]$, we can simply check for existence of provenance by taking the intersection between the values of b_1 in the query and in the table. Then we can perform appropriate arithmetic to find the values of the input array that corresponded to that intersection.

We can now observe for forwards queries the same query processing algorithm does not work (due to the relative indexing of input cells to output ones). Suppose, that we were interested in a forward query over a_1. We would like to a table T that has absolute cell indexes on attributes $a_1, \ldots, a_m$ to calculate that query result. Our forward query processing algorithm has to materalizes a table T with de-relativized attributes.

Thus, we construct a forward query representation that relative indexes output cells to input ones. This can be formed directly from

[2]To avoid redundancy we only show the provenance of X and not Y

Function	Input Array Dimensions	ProvRC				Generalized Provenance Representation			
		b_1	b_2	a_1	a_2	b_1	b_2	a_1	a_2
np.negative(X)	X: (1000,)	[1, 1000]	-	0 (a_1b_1)	-	$[1, B_1]$	-	0 (a_1b_1) -	
np.add(X, Y)^2	X: (1000,), Y: (1000,)	[1, 1000]	-	0 (a_1b_1)	-	$[1, B_1]$	-	0 (a_1b_1) -	
np.sum(X, axis = 0)	X: (1000,)	1	-	[1,1000]	-	1	-	[1,B_1] -	
np.tile(X, (2, 1))	X: (1000,)	[1, 1000]	-	0 (a_1b_1)	-			n/a	
		[1001, 2000]	-	-1000 (a_1b_1)	-				
np.dot(X, Y)^2	X: (1000,1000), Y: (1000,1000)	[1, 1000]	[1, 1000]	0 (a_1b_1)	0 (a_2b_2)	$[1, B_1]$	$[1, B_2]$	0 (a_1b_1)	0 (a_2b_2)
np.transpose(X)	X: (1000,1000), Y: (1000,1000)	[1, 1000]	[1, 1000]	0 (a_1b_2)	0 (a_2b_1)	$[1, B_1]$	$[1, B_2]$	0 (a_1b_2)	0 (a_2b_1)

Table 4: Examples of compression results on common operations

Table 5: Alternative Compression Representation to Enable Forward Queries

a_1	a_2	b_1	b_1a_1	b_1a_2
[1,3]	[1,2]	-	0	-

the original compressed representation without having to compress the data twice. For every row, $s \in S'$, in most cases, we can generate a row, t in T as followed:

(1) For $i \in [1, m]$,
- If $s.a_i$ is not empty, set $t.a_i := s.a_i$
- Else, find a non-empty $s.a_ib_j$ for some $j \in [1, l]$ and set $t.a_i := s.b_j + s.a_ib_j$

(2) For $j \in [1, l]$,
- if $s.a_ib_j$ is not empty for some $i \in [1, m]$, set $t.b_ja_i := s.a_ib_j$
- Else, $t.b_j := s.b_j$

There is an edge condition where, for a value of $j \in [1, l]$, there are multiple (>1) values of i where $s.a_ib_j$ is non-empty. In this case, the absolute cells in array X that correspond to s spatially forms a set of rectangles along a diagonal axis relative to $s.b_j$. Under this condition, there isn't a way to represent those cells as one row in T when $s.b_j$ is an interval. Instead, we expand out $s.b_j$ from its interval representation to form multiple rows. Given $s.b_j = [x, y]$, for all $p \in [x, y]$, we have the row, s_p, such that $s_p.b_j = p$ and $s_p.c = s_i.c$ for $c \in C, \neq b_j$. The rows in $\{s_p\}$ now can follow the conversion algorithm to rows in T detailed above. In our experience, such cases are rare, and the forward query representation table generally take up the same size as the original compression.

In Table 5, we show the resulting forward query table for our running example. We can observe that this table encodes the same information as the previous representation, but have absolute values on attributes $a_1, \ldots, a_m$.

5 IN-SITU QUERY PROCESSING

In this section, we describe the internal algorithm by which backward and forward queries are performed in-situ over the two compressed tables. As previously mentioned, the function `prov_query` is called by the user to query provenance between two Array objects. We further specify that, `query_cells` is a list of coordinates in the form of $Q : (q.a_1, \ldots q.a_m)$ that describe rectangle spatial indices of the cells in X_1. Specifically, $q.c$ is an interval that denote the cell indices on a attribute c.

5.1 Baseline: No Compression

Let us first see how query processing works without compression. Let $X_1 \rightarrow X_2 \ldots \rightarrow X_n$ be the path of a query made to `query_cells`, Q. In the baseline case without ProvRC compression, the query is performed as a predicate over joins provenance tables between adjacent arrays on the path. For every cell, q, within the coordinates of Q, we make the following query:

```
SELECT (c_1, ..., c_k)
    FROM R1(X1, X2) NATURAL JOIN R2(X2, X3)
         R2(X2, X3) NATURAL JOIN R3(X3, X4)
         ...
    WHERE a_1 = q_a_1 AND
        ... AND
        a_m = q_a_m
```

In this SQL query, each table R_i is the provenance relation between arrays X_i and X_j, and $a_1, \ldots, a_m, c_1, \ldots, c_k$ are the attributes of X_1 and X_n respectively. When this data is not compressed, such a SQL query is sufficient to produce the result of forward or backward queries.

However, we see that this query is highly structured with a chain-structured join over integer relations. Since these integers further have spatial locality, this join can be expressed as joins over intervals. When this query is performed over compressed tables, DSLog uses a custom query processing algorithm to exploit the interval representation of those tables.

5.2 DSLog Query Processing

Let's suppose that instead of working with the original relations R_i, we work with compressed relations W_i. We assume that each W_i is appropriately constructed using the ProvRC compression algorithm.

The uncompressed R tables describe edges in a graph. Likewise, we can think of each row of W tables as describing "bundles" of edges, namely vertex intervals that have an all-to-all relationship between them. For example, a record in the table $W(X, Y)$ that is $([1, 2], [1, 3])$ represents the edges between arrays X and Y with indices $(1, 1), (1, 2), (1, 3), (2, 1), (2, 2), (2, 3)$. These types of all-to-all relationships commute nicely with joins. Let's consider another table $W(Y, Z)$ that has a record $([1, 4], [1, 1])$. Due to the all-to-all relationship, the join $W_1(X, Y) \bowtie W_2(Y, Z) = W_3(X, Y, Z)$ can simply be constructed by taking an intersection of the intervals: $([1, 2], [1, 3], [1, 1])$.

This is the key performance insight behind the DSLog query processing algorithm, namely, joins can happen directly over the compressed intervals. Note in the example above how much we can reduce the I/O cost by performing the join over the intervals rather than the expanded records. In the following text, we walk through the query processing algorithm.

Step 1. Query Encoding. We will use the running example operation of `Y = np.sum(X, axis = 1)`. For this operation, we query over the path $X \rightarrow Y$ (i.e., a forward query). The corresponding compressed provenance table is shown in Table 5. The query specification is the in the form of a set of input cells $(a_1, a_2) = (1, 1), (1, 2), (2, 1), (2, 2)$.

The first step of the query processing algorithm is to encode the query in the same format as the data. This is typical with other in-situ query processing algorithms, e.g., dictionary coding [2]. This means that we generate a table Q (a query table) that contains intervals of interest along the relevant cells. For the query in our running example:

a_1	a_2
[1,2]	[1,2]

The same logic would hold for a backward query, just that Q would be specified over the "b" attribute in the example instead.

Step 2. Interval-Join Expansion. We expand the query into a particular type of θ-join that finds pairs with intersecting intervals along shared attributes:

$$R(A,B) \bowtie_\theta S(B,C) = \{(r,s) : r.B \cup s.B \neq \emptyset\}$$

This θ-join over all of the relevant provenance relations over a query path:

$$O = Q(X_1) \bowtie_\theta W_1(X_1,X_2) \bowtie_\theta \ldots \bowtie_\theta W_{n-1}(X_{n-1},X_n)$$

Note that the query table is included in this join expansion.

We process these joins in a left-to-right query plan, which processes the joins in the order they show up in the path specified by the user.

$$O = ((((Q(X_1) \bowtie_\theta W1(X_1,X_2)) \bowtie_\theta W_2(X_2,X_3))\ldots)$$

From a query optimization perspective, if the Q table is relatively small then the intermediate join results that propagate left-to-right are also similarly small. Note the query will always be on the left side of this join because it will always define intervals over the first array in the query path. With our running example, the output of this join expansion would be:

a_1	a_2	b_1	b_1a_1	b_1a_2	q_1	q_2
[1,3]	[1,2]	-	0	-	[1, 2]	[1,2]

Step 3. Converting Relative Indexes. This left-to-right join plan has a key subtlety because of the relative indexing. The algorithm cannot join on a shared attribute that has been relatively indexed. After each join is executed, the intermediate result must turn every attribute back into an absolute index. This can be calculated by reversing the provenance relationships captured in the compression algorithm to compute absolute coordinates for attributes. Otherwise, we can compute the attribute values from the backward and forward tables using the functions rel_{for} and rel_{back} respectively.

$$rel\_back(q,\delta) := [q^1+\delta^1, q^2+\delta^2]$$

$$rel\_for(q,a,\delta) :=$$

$$[\max(q^1-\delta^2, a^1-\delta^2), \min(q^2-\delta^1, a^1-\delta^2)]$$

For any interval x, we represent the endpoints as $[x^1,x^2]$. Scalar points can be represented by intervals with coordinates $x^1 = x^2$.For row $w \in W''$, we define q_w such that $q_w.a_i := w.a_i \cap w.q_i$. A rectangular interval, r, over X_2 is computed as follows. For $j \in [1,l]$,

- If $w.b_j$ is not empty, set $r.b_j := w.b_j$
- Else, find a non-empty $w.a_ib_j$ for some $i \in [1,m]$,
  - If the table is a backward representation, set $w.b_j := rel\_back(q_w, w.a_ib_j)$
  - If the table is a forward representation, set $w.b_j := rel\_for(q_w, w.a_i, w.a_ib_j)$

We define the set of all output intervals, $\{r\}$, as Q_{out}. This is the set of cells in X_2 that are covered by provenance relationships from Q.

For the row in our example Table 3, we calculate:

$$w.b_1 = rel\_for([1,2],[1,3],0)$$

$$w.b_1^1 := \max(1-0, 1-0) = 1$$

$$w.b_1^2 := \min(2-0, 3-0) = 2$$

$$w.b_1 = [1,2]$$

As noted before, we perform the previous steps alternatively in a left-to-right join evaluation. `query_cells` return the final set of cells from this algorithm. Since the example query only has one step in the provenance path, `query_cells` returns $(b_1 : [1,2])$.

When multiple intervals are present, DSLog performs an optimization step to merge intervals in Q_{out}. In the previous step, the algorithm can retrieve adjacent intervals that can be merged into a larger rectangle. We discover and merge those intervals using a method similar to the multi-attribute range encoding step in `ProvRC`. This isn't shown in our example as we only retrieve one interval.

6 PROVENANCE RE-USE

In many data science workflows, the same provenance graphs are captured repeatedly. This section describes how previously captured provenance graphs can be re-used in DSLog when `register_function` is called. There is an analogy here to query rewriting with materialized views [6], namely, if a workflow contains a sub-routine that has already been captured DSLog can short-circuit that sub-routine.

6.1 Basic Design

A function signature can be associated with each call of `register_function`. The function signature is defined by the name of the array program, `function_name`, the set of input arrays, $\{X\}$, and other input arguments. The list of full relevant input arguments is given by the user with `func_arguments`.

When full re-use (`re-use='full'`) is enabled, a key-value relation between the function signature and generated provenance tables is saved. In subsequent calls to `register_function` with the same signature, the previous provenance tables are used, bypassing calls to the capture object. This basic strategy has been employed in systems like Lima [23]. As we will see in the following, our system goes one step further in term of re-use, by defining more general signatures.

6.2 Provenance Extrapolation

Beyond exact matches, we can sometimes enable matches on partial function signatures. Firstly, in a number of important scenarios, the provenance relation depends on only the shape of the input arrays and not the actual input arrays themselves. For example, the provenance for all linear algebra functions or a forward evaluation of a neural network on a different mini-batch of data fits this criteria.

Now, when `register_function` is called with the argument `re-use= 'shape'`, DSLog internally stores the key-value relation:

$$\text{dim\_sig}(\text{function\_name}, \{shape(X)\}, \text{other\_arguments}) : \{R\}$$

where $shape(X)$ gives the shape of array X, and $\{R\}$ is set of the provenance tables associated with `function_name`. DSLog automatically fetches the values for $shape(X)$ from the Array objects named in `input_arrays`. Similar to the general case, in subsequent calls to `register_function`, if there is a match on the "shape" function signature, we can directly re-use the provenance registered above.

It is also often possible to extrapolate provenance in some scenarios without matching exactly on shape. For example, consider an image smoothing task with a convolutional filter. The user runs this program twice, and for the second time, she down-samples the image by a constant factor. In this case, the provenance relationship from the second run is simply a scaled version of the first run.

We leverage the intervals in our compressed representation to do the extrapolation. Given a compressed relational table, S, to create the generalized representation, for $d_i \in dim(X)$, we identify all intervals equal to $[1, d_i]$ in S, and replace them with the generalized interval $[1, D_i]$, where D_i is an indicator of that attribute. If those intervals are the only ones that are dependent on the shape of the input arrays, and there is no data dependence, then a generalize representation is formed that is independent of input array shape.

When `register_function` is called with the argument `re-use= 'gen'`, DSLog internally stores the key-value relation:

$$\text{gen\_sig}(\text{function\_name}, \text{other\_arguments}) : \{R'\}$$

where $\{R'\}$ is the set of generalized provenance tables associated with `function_name`. In subsequent calls, if there is a match in `gen_sig` with the function signature, the generalized provenance is used, where $\{D_i\}$ is replaced with the corresponding new input array shapes. Table 4 shows a selection of generalized provenance for different functions in the "Generalized Provenance Representation" column. From the table, we observe that there exists compact representations for one-to-one element-wise functions (1) (2), axis aggregates (3) (4), and axis pivots (5).

6.3 Automatic Materialization.

Understanding when re-use is possible requires knowledge of the exact provenance pattern of an operation, and how that provenance pattern is stored. This is time-intensive and error-prone for users of DSLog. Therefore, we also offer the option to automatically predict when functions can be re-used using heuristics to extrapolating based on past behavior.

When `register_function` is called with the argument `re-use= 'predict'`, DSLog internally stores *temporary* `dim_sig` and `gen_sig` key-value relations. In the next m calls to `register_function` that match on either partial function signature, we check if the provenance tables stored match the new provenance tables. If there is a match after m calls, we move the key-value relations to the permanent tables. For the `gen_sig` matches, we also require that the m calls have different array shape values. In our current implementation, we set $m = 1$.

7 EXPERIMENTS

We performed experiments to evaluate the compression, query and re-use components of DSLog. Our experiments were performed on Chameleon, a large scale computer science research platform [9], with a Intel Xeon Gold 6126 Processor and 192 GiB of RAM. Since real data science pipelines are difficult to benchmark, for each experiment, we designed custom workloads to the best of our ability that demonstrate performance on a variety of data science operations. Compression storage experiments are performed over individual operations across lineage algebra, relational, and interpretable machine learning domains. Query performance is evaluated through stimulated pipelines over random `numpy` operations, and two customized pipelines that integrate relational and explainable machine learning operations. We evaluate coverage of the re-use component over a large curated set of `numpy` functions.

7.1 DPSM Baselines and Implementation

To evaluate the compression and query components of DSLog, we have implemented alternative storage formats and physical designs. We will see that these approaches are unable to exploit all of the structures in provenance captures. The query interface is served by the DuckDB system [27]. We design all of the storage formats to be transparently queried from DuckDB; however, for some baselines this might require explicit decompression before querying. Note that most of our baselines are columnar storage formats since we found superior performance with such formats over row-oriented ones.

Raw. The captured provenance tuples are stored in a row-oriented format with no data compression. This format is similar to the design principles of Ground [15], but with an implementation in DuckDB so we could compare to baselines [3].

Parquet. The captured provenance tuples are stored in sorted order in Apache Parquet files, which is a columnar file format that can be parsed by different query processing engines. We use default encoding and row-group partitioning settings.

Parquet-GZip. The captured provenance tuples are stored in a GZIP compressed Parquet file, as suggested by industry practice [1].

Turbo-RC. We designed a custom columnar format that applies state-of-the-art integer compression over each column. This compression included run-length encoding combined with integer entropy coding [14]. This baseline is designed to optimize storage size over downstream query performance.

7.2 Provenance Capture Size

In this experiment, we compare the different storage formats in terms of size on disk. We applied each of the baselines and `ProvRC` to the operations described below. A strong baseline, a version of `ProvRC` with only multi-attribute encoding and no relative value transformation (`ProvRC-NoRel`), is also included in the results. Note that we only materialize the `ProvRC` representation for backward queries. Since the backward and forward queries store the same information, only one needs to be stored in long-term storage. For each of the baselines, we captured the same provenance, and applied the baseline method to represent that captured provenance. We measured the file size of the database files that were ultimately served to DuckDB.

[3] We referred to the table design in https://github.com/ground-context/ground

	Operation Name	numpy	Input Arrays Size	Output Arrays Size	Notes
1	Negative	negative	X: (10, 100000)	Z: (10, 100000)	element-wise
2	Addition	add	X: (10, 100000), Y:(10, 100000)	Z: (10, 100000)	element-wise over two inputs
3	Aggregate	sum	X: (1000, 1000),	Z: (1000, 1)	one axis aggregation
4	Repetition	tile(reps=(2,2))	X: (10, 100000)	Z: (20, 200000)	duplicate array 4 times
5	MatMul	dot	X: (1000, 1000), Y: (1000, 1000)	Z: (1000,1000)	linear algebra matrix product
6	RandFilter	-	X: (1000000, 1)	Z: (1000000, 1)	filter elements greater than average on a randomly generated array
7	ImgFilter	-	X: (1000, 1000)	Z: (1000, 1000)	filter non-zero elements on upscaled MNIST image
8	Lime	-	X: (4160, 4160)	Z: (4160, 4160)	Lime explanation for YOLOv4 on upscaled VIRAT dataset frame
9	DRISE	-	X: (4160, 4160)	Z: (4160, 4160)	DRISE explanation for YOLOv4 on upscaled VIRAT dataset frame
10	Group By	-	X: (1000000, 2)	Z: (137, 2)	Group by over IMDB dataset title.basics on 'startYear' (first billion rows), summing on 'isAdult'
12	Inner Join	-	X:(1000000, 9), Y: (1000000, 4)	Z: (466005, 12)	Inner Join over IMDB title.basics and title.episode tables on 'tconst' (first billion rows)

Table 6: Description of Operations Evaluated in Compression Storage Experiments

Name	Raw	Parquet		Parquet-GZIP		Turbo-RC		ProvRC- NoRel		ProvRC- GZIP	
	Absolute (MB)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)
Negative	22.66	5.05	22.31	4.33	19.10	0.116	0.515	8.66	38.20	**0.0106**	**0.0469**
Addition	45.33	10.11	22.31	8.66	19.10	0.233	0.515	17.32	38.20	**0.0212**	**0.0469**
Aggregate	20.55	0.131	0.639	0.0255	0.124	0.00910	0.0442	0.0173	0.0844	**0.0109**	**0.0531**
Repetition	98.22	25.20	25.65	14.52	14.78	0.153	0.156	29.04	29.56	**0.0106**	**0.0108**
MatMul	40127.85	254.87	0.635	48.06	0.119	15.47	0.0385	0.0347	8.65×10^{-5}	**0.0212**	$\mathbf{5.30} \times 10^{-5}$
RandFilter	12.26	4.56	37.22	**2.04**	**16.69**	2.36	19.24	4.09	33.34	4.16	33.97
ImgFilter	4.288	1.96	45.93	1.06	24.73	0.285	6.66	2.12	49.46	**0.0109**	**0.254**
Lime	24.09	0.529	2.19	8.242	0.513	0.0246	0.102	0.247	1.02	**0.0123**	**0.0514**
DRISE	8.82	0.0898	1.01	8.242	0.271	0.0132	0.149	0.0473	0.536	**0.0111**	**0.126**
Group By	21.11	4.41	20.93	**2.00**	**9.51**	15.88	75.25	2.87	13.61	9.10	43.11
Inner Join	142.42	14.35	10.07	8.242	3.70	89.47	62.82	10.54	7.401	**0.600**	**0.421**

Table 7: Comparison of Compression Ratio for Different Algorithms Against ProvRC

We constructed a workload of 12 individual data science operations across different applications and provenance capture techniques. The full summary of each operation can be found in Table 6. The workload can be broken down into the following categories:

(1) Seven of those operations are numpy functions: five of these are statistical operations where the provenance is independent of the values of the array, and two are conditional operations where the provenance is data-dependent. These operations are tracked with the fine-grained tracking described earlier.
(2) Two of these operations are over explainable AI. A frame was chosen from VIRAT [20], a surveillance camera dataset, and YOLOv4 object detection was applied over that frame to detect a 'car' object. On this model and data, DSLog captured provenance between the input cells and this car object with LIME and DRISE.
(3) The final two operations are relational operations and the stored provenance is a standard why-provenance [8]. These operations are applied to a subset of IMDB dataset tables. Note that the 'tconst' and 'startYear' columns are sorted in the original tables, but 'isAdult' is unsorted.

Table 7 shows the results size of the provenance file on disk for each compression algorithm, and relative compression ratios to the raw file. In general, the columnar compression baselines (Parquet, Parquet-Gzip and Turbo-RC) had consistently decent performance over the raw storage files, indicating that the provenance table structure leads to column regularity that generic coders can recognize. While Turbo-RC generally achieved the best compression ratio out of the baselines for the structured provenance tables, with a compression <1% in many cases, Parquet-Gzip improved the compression ratio for all tables over Parquet, and was best performing overall on the two functions with the most unstructured provenance on unsorted data (RandFilter and Group-By).

For all other operations, ProvRC was the most space efficient compression algorithm, with a compression ratio of <0.05% for most operations. The operation that it is most efficient on is MatMul, beating the next best baseline by 700x. We note that ProvRC has a slight edge in situations where other algorithms generated highly compressed files (e.g. Turbo-RC over the "Negative" function). Additionally, ProvRC is able to compress more complex multi-columnar patterns (e.g. MatMul) that is not fully captured by the baselines. Both of these points can be attributed to an algorithm specifically designed for structured provenance patterns. We also see that ProvRC without relative indexing performs poorly in many cases where the original ProvRC algorithm is able to optimally find a pattern. This suggests that relative indexing is an essential concept to capture the relationship between input and output array indices for many patterns.

To conclude, we observe that the columnar storage formats achieves great space improvements over the raw storage format for unstructured provenance tables, but ProvRC is more optimal for structured provenance tables. An optimization technique to evaluate in the future would be integrating multiple compression techniques into DSLog depending on table type.

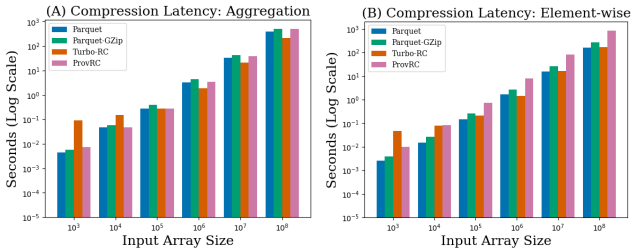

Figure 3: (A) and (B) benchmark the compression algorithms in terms of their latency as a function of input size

7.3 Compression Latency

Next, we evaluate the latency of the different baselines and ProvRC. We consider the full read, format-conversion, compression and flush latency time it takes to write the compression provenance tables to disk. We measured the latency of table compression for the two extremes of function types: one-to-one element-wise operation, and one-axis aggregation functions. This latency is measured over a range of array shapes.

The comparison of latency between the different compression algorithms can be observed in Figure 3. We note that the latency between different compression algorithms are within an order of magnitude of each other, and increase as input array shape increases; however, ProvRC is consistently one of the worse performers, especially on the larger element-wise provenance table. Given that the current implementation is written in Python for simplicity, and the baselines are written in C++ and from mature/public codebases, we find this performance reasonable. ProvRC is also highly parallelizable, so we expect vast performance gains from a multi-threaded implementation.

7.4 Query Processing

In this experiment, we evaluate the query latency of provenance queries over fixed workflows. All of these queries run over 5 function operations, meaning they have to join 5 different provenance graphs together. We fix the (forward) query path and generate queries of different selectivities. Specifically, each query is over a random rectangular range of a set number of cells, and the wall-clock latency to get the query results is measured. For brevity, backward query experiments are not shown in the paper - those results have similar trends as the forward query results. We will now outline the workloads evaluated.

Firstly, for twenty runs, we simulate a variety of data science workflows by randomly selecting 5 numpy functions to form a pipeline. Each pipeline is ran with an initial randomly generated array with size $(1000, 100)$ and the provenance is captured with the `tracked_cell` object. The five functions are chosen from an initial list of hand-selected numpy functions that can intake and output a single `float_64` arrays. The full list of viable functions is presented in the Appendix of the full paper. This set of workflows evaluates the performance of our query design over a diverse operations. Figure 4 shows the average wall-clock query latency times over these twenty workflows. In the figure, minimum and maximum latency times of the twenty pipelines are also shown with an interval bar. Additionally, in these experiments, we also perform an ablation study where the merge step is skipped in the query over ProvRC compression (ProvRC-NoMerge).

	Image Pipeline	Relational Pipeline
1	Resize to (416,416)	Inner Join on 'tconst'
2	Increase Pixel Luminosity	Filter Columns with Missing Value
3	Rotate 90°	Add Two Columns Together
4	Horizontal Flip	One-Hot Encode on Primary "genres"
5	Evaluate Lime on YOLOv4	Add Constant to One Column

Table 8: Summary of the image and relational workflows

There are multiple interesting points to note. Firstly, there is a large variation in query latency for all compression algorithms, with the minimum and maximum query times being up to 100x the average. This demonstrates that the contents of provenance table queried have a large impact on latency. Additionally, the columnar format tables generally has better or on par latency compared to the raw table, showing that there are query benefits to the columnar format, even though we are querying over multiple attributes. For extremely selective queries of under 100 cells, parquet formats achieved the lowest latency; however, all compression formats returned query results in <1 second, and hence probably have similar real-world usability. There is significant improvement of DSLog over baselines on query latency for queries with over 100 cells. For a query of 10^8 cells over the full input array, DSLog with ProvRC had about 100x latency reduction on average. This can be attributed to the range compression over the numpy operations, reducing the number of rows queried in the tables. Thus we can conclude that DSLog introduces significant improvements on queries over large cell ranges, while incurring slight overhead costs on extremely selective queries. We also not that ProvRC-NoMerge performed worse that ProvRC over all types of queries. Hence, the merge step is show to have a significant inpact on query performance.

For additional experiments, we evaluate the query latency over real data science workflows. Two simple workflows are implemented and tracked in DSLog: one for computer vision model debugging and another for relational data pre-processing. Each of these workflows have 5 steps that include a mix of data dependent and data independent operations, as summarized in Table 8. The image workflow is performed over a frame of the VIRAT dataset, and the relational workflow is performed over the first billion rows of the IMDB title.basics and title.episode tables. These are the same data used in the compression storage experiments.

Figure 5 shows the result of latency query experiments over these two workflows. We evaluate queries over a percentage of the input array. In these graphs, we omit Raw and ProvRC-NoMerge for clarity. In general, the trends from Figure 4 can be observed here. Even the more selective queries in these experiments, i.e. 0.1% and 1%, cover >100 cells, so it makes sense that ProvRC performs significantly better over all queries compared to baseline. With the relational pipeline, all operations are highly structured, lending itself well to our compression algorithm. Hence, this is an optimal case for queries as well. In Figure 5 (B), we observe that, while the increase in query response time matches the trend in other graphs as query selectivity decreases for the baselines, this is not true for ProvRC. Even querying for the full provenance table, ProvRC has a query reponse time of <1 second, a 2000x improvement over the other tables.

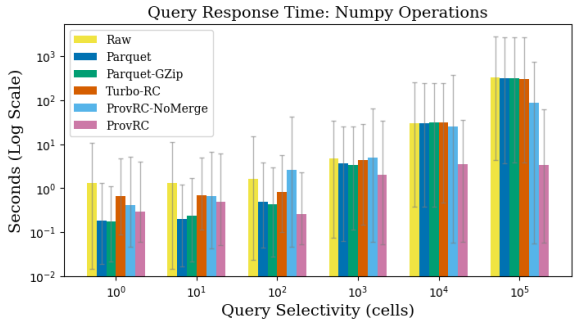

Figure 4: Evaluation of query latency on random `numpy` pipelines

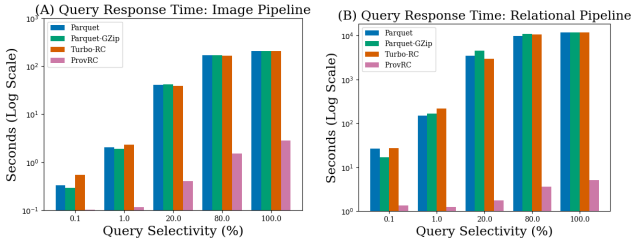

Figure 5: Evaluation of query latency on (A) image machine learning and (B) relational pipelines

7.5 Provenance Re-Use Coverage

We approximate operational coverage DSLog's re-use prediction algorithm in real-world data science pipelines by evaluating coverage over a full set of 136 different functions in `numpy`. These functions consists of the functions in `numpy`'s API that meet the criteria of current supported function signatures: they (1) can intake and output `float_64` arrays, and (2) can intakes scalar-only arguments outside of `float_64` arrays. This is a super-set of functions used in the `numpy` query experiments. A detailed list of all 136 functions is presented in the Appendix of the full paper. We use `tracked_cells` to annotate the provenance.

In our simulated workflow, DSLog populates the re-use tables over 200 runs. For every second run, array sizes, array values, and scalar arguments are randomly chosen. DSLog automatically tries to re-use operation provenance as described in Section 6. We evaluate the number of functions where DSLog successfully discovers and stores shape-based and generalized re-use signatures. As automated capture re-use is a novel task, there does not exist any baselines to compare our coverage against. We simply evaluate the percentage of all functions we are able to cover with both shape-based and generalized re-use.

Table 9 summarizes the results of that evaluation. We divide the `numpy` functions into two categories - element-wise function and other more complex patterns. As one can see from the table, element-wise functions make up over half of our results. DSLog is well suited to deal with these functions; they are easily identifiable after applying the `ProvRC` compression, and are completely captured with both shape-based and generalized re-used. For complex

Function	Total	`dim_sig`	`global_sig`	Error
element-wise	75	75	75	0
complex	61	51	24	1
total	136	126	99	1
total (%)	-	92.65	72.79	0.73

Table 9: A table showing how many `numpy` API functions are covered and accurately characterized by provenance signatures generated for re-use.

patterns, we are able to generate shape-based and generalization re-use for 86% and 39% of the functions respectively. In total, we cover 92% of functions with dimensional views and 72% with generalization views. If DSLog is deployed for full array-based programs in the future, we expect re-use to be a significant part of any provenance capture strategy.

There is one mis-prediction that occurred, where DSLog generated a wrong generalized view for the function `numpy.cross`. This is due to the fact that `numpy.cross` has different provenance patterns depending on the size of the second dimension. However, there are only two valid sizes of that dimension (2 or 3), so it is likely that we observe only one pattern after m runs is high. This is the downside of setting m to be a low value. Mis-predictions like this can be mitigated, by periodically re-evaluation the materialized view with new calls. Overall, we still found an error rate of less than 1% on our simulated workflow. Of course, workflows in the wild are more unpredictable, and the set of function signatures used not purely random, leading the higher likelihood of similar generalization errors. It is very important to note that, theoretically, **any** black-box provenance prediction algorithm that doesn't capture exact signature can make errors. Therefore, it is up to the user to verify results or use their judgement call in deploying re-use.

8 CONCLUSION

In this paper, we presented a fine-grained data provenance storage manager for data science, DSLog. DSLog introduces a novel compression provenance technique, `ProvRC`, an in-situ forward and backward query optimization over compressed provenance, and a capture re-use algorithm. We believe that DSLog is the first step towards a full provenance system for data science that can leverage fine-grained provenance for reproducibility and reliability.

REFERENCES

[1] Parquet compression.
[2] D. Abadi, S. Madden, and M. Ferreira. Integrating compression and execution in column-oriented database systems. In *Proceedings of the 2006 ACM SIGMOD international conference on Management of data*, pages 671–682, 2006.
[3] M. Boehm, M. W. Dusenberry, D. Eriksson, A. V. Evfimievski, F. M. Manshadi, N. Pansare, B. Reinwald, F. R. Reiss, P. Sen, A. C. Surve, et al. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.
[4] N. Burkart and M. F. Huber. A survey on the explainability of supervised machine learning. *Journal of Artificial Intelligence Research*, 70:245–317, 2021.
[5] A. P. Chapman, H. V. Jagadish, and P. Ramanan. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 993–1006, 2008.
[6] S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim. Optimizing queries with materialized views. In *Proceedings of the Eleventh International Conference on Data Engineering*, pages 190–200. IEEE, 1995.
[7] C. Chen, H. T. Lehri, L. Kuan Loh, A. Alur, L. Jia, B. T. Loo, and W. Zhou. Distributed provenance compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 203–218, 2017.
[8] J. Cheney, L. Chiticariu, and W.-C. Tan. *Provenance in databases: Why, how, and where.* Now Publishers Inc, 2009.

[9] C. Cloud. A configurable experimental environment for large-scale cloud research. chameleoncloud. org. retrieved, 2022.

[10] B. Derakhshan, A. Rezaei Mahdiraji, Z. Abedjan, T. Rabl, and V. Markl. Optimizing machine learning workloads in collaborative environments. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 1701–1716, 2020.

[11] D. Eppstein. Graph-theoretic solutions to computational geometry problems. In *International Workshop on Graph-Theoretic Concepts in Computer Science*, pages 1–16. Springer, 2009.

[12] J. Goecks, A. Nekrutenko, and J. Taylor. Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences. *Genome biology*, 11(8):1–13, 2010.

[13] S. Grafberger, S. Guha, J. Stoyanovich, and S. Schelter. Mlinspect: A data distribution debugger for machine learning pipelines. In *Proceedings of the 2021 International Conference on Management of Data*, pages 2736–2739, 2021.

[14] L. Hanzo, T. H. Liew, and B. L. Yeap. *Turbo coding, turbo equalisation and space-time coding*. John Wiley & Sons, 2002.

[15] J. M. Hellerstein, V. Sreekanti, J. E. Gonzalez, J. Dalton, A. Dey, S. Nag, K. Ramachandran, S. Arora, A. Bhattacharyya, S. Das, et al. Ground: A data context service. In *CIDR*. Citeseer, 2017.

[16] M. Interlandi, K. Shah, S. D. Tetali, M. A. Gulzar, S. Yoo, M. Kim, T. Millstein, and T. Condie. Titian: Data provenance support in spark. In *Proceedings of the VLDB Endowment International Conference on Very Large Data Bases*, volume 9, page 216. NIH Public Access, 2015.

[17] H. Kuehn, A. Liberzon, M. Reich, and J. P. Mesirov. Using genepattern for gene expression analysis. *Current protocols in bioinformatics*, 22(1):7–12, 2008.

[18] M. H. Namaki, A. Floratou, F. Psallidas, S. Krishnan, A. Agrawal, and Y. Wu. Vamsa: Tracking provenance in data science scripts. *CoRR*, abs/2001.01861, 2020.

[19] M. Nikolic, M. Elseidy, and C. Koch. Linview: incremental view maintenance for complex analytical queries. In *Proceedings of the 2014 ACM SIGMOD international conference on Management of data*, pages 253–264, 2014.

[20] S. Oh, A. Hoogs, A. Perera, N. Cuntoor, C.-C. Chen, J. T. Lee, S. Mukherjee, J. Aggarwal, H. Lee, L. Davis, et al. A large-scale benchmark dataset for event recognition in surveillance video. In *CVPR 2011*, pages 3153–3160. IEEE, 2011.

[21] D. Olteanu. On factorisation of provenance polynomials. In *In TaPP*. Citeseer, 2011.

[22] V. Petsiuk, R. Jain, V. Manjunatha, V. I. Morariu, A. Mehra, V. Ordonez, and K. Saenko. Black-box explanation of object detectors via saliency maps. In *Computer Vision and Pattern Recognition (CVPR)*, 2021.

[23] A. Phani, B. Rath, and M. Boehm. Lima: Fine-grained lineage tracing and reuse in machine learning systems. In *Proceedings of the 2021 International Conference on Management of Data*, pages 1426–1439, 2021.

[24] G. E. Pibiri and R. Venturini. Techniques for inverted index compression. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2020.

[25] N. Potti and J. M. Patel. Daq: a new paradigm for approximate query processing. *Proceedings of the VLDB Endowment*, 8(9):898–909, 2015.

[26] F. Psallidas and E. Wu. Smoke: Fine-grained lineage at interactive speed. *arXiv preprint arXiv:1801.07237*, 2018.

[27] M. Raasveldt and H. Mühleisen. Duckdb: an embeddable analytical database. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1981–1984, 2019.

[28] R. Rampin, F. Chirigati, D. Shasha, J. Freire, and V. Steeves. Reprozip: the reproducibility packer. *Journal of Open Source Software*, 1(8):107, 2016.

[29] E. K. Rezig, L. Cao, G. Simonini, M. Schoemans, S. Madden, N. Tang, M. Ouzzani, and M. Stonebraker. Dagger: a data (not code) debugger. In *CIDR 2020, 10th Conference on Innovative Data Systems Research, Amsterdam, The Netherlands, January 12-15, 2020, Online Proceedings*, 2020.

[30] M. T. Ribeiro, S. Singh, and C. Guestrin. " why should i trust you?" explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining*, pages 1135–1144, 2016.

[31] T. Sellam, K. Lin, I. Huang, M. Yang, C. Vondrick, and E. Wu. Deepbase: Deep inspection of neural networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1117–1134, 2019.

[32] Z. Shang, E. Zgraggen, B. Buratti, F. Kossmann, P. Eichmann, Y. Chung, C. Binnig, E. Upfal, and T. Kraska. Democratizing data science through interactive curation of ml pipelines. In *Proceedings of the 2019 international conference on management of data*, pages 1171–1188, 2019.

[33] Z. Shang, E. Zgraggen, and T. Kraska. Alpine meadow: A system for interactive automl.

[34] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas. Secure program execution via dynamic information flow tracking. *ACM Sigplan Notices*, 39(11):85–96, 2004.

[35] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood. Complete information flow tracking from the gates up. In *Proceedings of the 14th international conference on Architectural support for programming languages and operating systems*, pages 109–120, 2009.

[36] M. Vartak, J. M. F. da Trindade, S. Madden, and M. Zaharia. Mistique: A system to store and query model intermediates for model diagnosis. In *Proceedings of the 2018 International Conference on Management of Data*, pages 1285–1300, 2018.

[37] E. Wu, S. Madden, and M. Stonebraker. Subzero: a fine-grained lineage system for scientific databases. In *2013 IEEE 29th International Conference on Data Engineering (ICDE)*, pages 865–876. IEEE, 2013.

[38] Y. Xie, D. Feng, Z. Tan, L. Chen, K.-K. Muniswamy-Reddy, Y. Li, and D. D. Long. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM international conference on Information and knowledge management*, pages 1752–1756, 2012.

[39] Y. Xie, K.-K. Muniswamy-Reddy, D. D. Long, A. Amer, D. Feng, and Z. Tan. Compressing provenance graphs. In *3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP 11)*, 2011.

[40] D. Xin, S. Macke, L. Ma, J. Liu, S. Song, and A. Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *arXiv preprint arXiv:1812.05762*, 2018.

[41] M. Zaharia, A. Chen, A. Davidson, A. Ghodsi, S. A. Hong, A. Konwinski, S. Murching, T. Nykodym, P. Ogilvie, M. Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

[42] Y. Zhang and Z. G. Ives. Juneau: data lake management for jupyter. *Proceedings of the VLDB Endowment*, 12(12), 2019.

[43] Z. Zhang, E. R. Sparks, and M. J. Franklin. Diagnosing machine learning pipelines with fine-grained lineage. In *Proceedings of the 26th International Symposium on High-Performance Parallel and Distributed Computing*, pages 143–153, 2017.

[44] W. Zhou, S. Mapara, Y. Ren, Y. Li, A. Haeberlen, Z. Ives, B. T. Loo, and M. Sherr. Distributed time-aware provenance. *Proceedings of the VLDB Endowment*, 6(2):49–60, 2012.

[45] D. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. Tainteraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, 2011.