THE UNIVERSITY OF CHICAGO

AUTOMATED PROVENANCE CAPTURE IN ARRAY-PROGRAMMING FRAMEWORKS

A DISSERTATION SUBMITTED TO
THE FACULTY OF THE DIVISION OF THE PHYSICAL SCIENCE
IN CANDIDACY FOR THE DEGREE OF
MASTER OF SCIENCE

DEPARTMENT OF COMPUTER SCIENCE

BY
JINJIN ZHAO

CHICAGO, ILLINOIS
MAY 25 2022

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

This paper presents DSLog, a system that efficiently capture and represent fine-grained data provenance in array-programming frameworks for black box functions. It uses a technique called annotated execution to capture "physical" provenance, automatically without user specification. We describe a low-level implementation that make this work for arrays up to 100 million (and more) cells, and improve capture performance up to 3400% over an high level baseline. We contribute a new compression algorithm, named `ProvRC`, that compresses such relations. We show that the `ProvRC` results in a significant storage reduction over functions with simple spatial regularity, beating alternative baselines by many orders of magnitude. Finally, we present the concepts of dimensional and generalized views over these compressed relational representation, which allows DSLog to recognize previously seen function (with only input array dimension information, and no input array information respectively), and re-use pre-existing materialized provenance views. We demonstrate that these views cover 92% and 73% respectively of 136 tested `numpy` functions, and preliminary results show that using the views have a marked improvement over pure naive annotated execution.

CHAPTER 1

INTRODUCTION

Manipulating multi-dimensional numerical arrays, or array programming, is a fundamental part of data science Soroush et al. [2011], Papadopoulos et al. [2016], Boehm et al. [2016], Stonebraker et al. [2011]. These arrays represent the vectors, matrices, and tensors used in machine learning, science, and geospatial applications. Since many of the core operations resemble relational algebra, and not surprisingly, there have been numerous efforts to bridge the worlds of "array programming" and relational database systems Boehm et al. [2016], Wang et al. [2020], Daniel et al. [2018], Nikolic et al. [2014]. Such connections allow for more effective code optimization Wang et al. [2020], result maintenance Nikolic et al. [2014], and data provenance Phani et al. [2021] in array programming frameworks. As computing moves towards machine learning and artificial intelligence as commoditized components, understanding the principles of data management for numerical arrays will be increasingly relevant Kumar et al. [2017].

This paper studies how ideas from relational data modeling can address *fine-grained data provenance* problems in array programming. Data provenance is the documentation of where a piece of data comes from and the computational operations by which it was produced. Provenance is valuable for debugging and auditing data analytics workflows. Collecting, storing, and managing provenance is well-studied in relational data management Cheney et al. [2009], Glavic [2021], and initial work has sought to extend these ideas to the array programming setting Phani et al. [2021], Wu et al. [2013]. While there is over a decade of research into this topic, provenance in multi-dimensional array workflows has recently had a surge of interest due to applications in machine learning debugging Vartak et al. [2018], Xin et al. [2018], Sellam et al. [2019], Rezig et al. [2020], Shang et al. [2019], Zhang et al. [2017]. These works, however, focus on narrow problem settings that restrict or make strong assumptions about user workflows.

We can characterize this lack of generality in two ways: (1) *operation constraints* and (2) *operation specification*. In (1), the system enforces that the user works within the boundaries of a known set of operations. For example, assume that the user applies neural network inference with a statically defined network architecture Vartak et al. [2018], Sellam et al. [2019], or only defines workflows that are compositions of operations in a domain-specific language (DSL) Phani et al. [2021]. Tracking provenance in such scenarios comes naturally since one can build a complete manifest of operations and make assumptions about how inputs to an operation relate to its outputs. Recognizing the limitations of (1), (2) allows users to specify custom operations with provenance descriptions. A good example of such a system is SubZero Wu et al. [2013], which defines an API to allow users to specify custom operations. These systems are complementary to those approaches described in (1) and often also maintain knowledge bases of common linear algebra, relational algebra, and common machine learning functions to automatically capture provenance in common cases.

Both approaches above fundamentally require control over the array programming API, make strong assumptions of what operations a user might use, and/or the technical capabilities of the user. For example, between TensorFlow 1.x and 2.x nearly 1300 new "raw operations" were added to their computational graph API tfr, most of which do not align with traditional linear algebra operations. This statistic also ignores the vast repositories of external contributor code that any user might draw on for their applications. Custom operations are also highly prevalent in modern scientific and machine learning workflows. For such "true" black-box settings, provenance options are limited, especially when the user was not to define the operation specifications. However, such applications are exactly the kind that could benefit from fine-grained provenance. For example, a user might want to know why particular results are `NaN` after applying one of these non-standard operations, and which input elements contribute to that result.

In an ideal world, provenance capture would be deeply integrated with the execution

environment where as an array-program is evaluated, the input-to-output relationship is incrementally built without user specification. We call this alternative strategy "annotated execution", where individual array elements are annotated with metadata describing how they were derived. But, to the best of our knowledge such a framework does not exist for the Python data science stack. While straightforward in principle, annotated execution comes at a steep overhead of computation and storage. Without operation-level information, one loses the ability to logically capture the provenance of known operations (which is faster and more concise). Thus, the key technical challenge is building an annotated execution framework is managing the storage and the computational overhead of provenance capture.

We present a new prototype framework called DSLog that takes a step towards efficient annotated execution in the `Python` data science stack and tracks provenance in `numpy` arrays. Our framework captures fine-grained provenance in `numpy` by defining an extended numerical data type that log accesses when it is manipulated by any numerical primitive operations (Section 3.3). A key contribution of DSLog is to use a relational data model to represent the captured provenance relationships, i.e., how input cells contribute to output cells. DSLog incorporates a series of post-hoc optimizations that organize and compress the captured relations, and pre-emptive optimizations to generate "provenance views" that recognize previously captured sub-routines. 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 4). We define the dimensional and generalized provenance views which allow us to capture provenance for function signatures independent of array content and array dimension size respectively (Section 5).

`ProvRC` is contribution independent of DSLog and is applicable to other provenance frameworks and capture methodologies. Integer-indexed multi-dimensional arrays are a sort of universal interface in data science that can represent dataframes, documents, time series, and images. The provenance relations of common operations over these data types often 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 also find that there is a deep connection between such a compression scheme and relational normalization, where element-wise operations induce one-to-one functional dependencies, aggregations induce multivalued dependencies, and compositions of operations resemble relational joins.

CHAPTER 2

BACKGROUND

First, we motivate our problem setting and implementation. In the context of our system and all baselines, *provenance* refers to the contribution relationship between input and output elements (a cell) in any array operation, i.e., which elements in the input array determine the values of each output element. The precise semantics of what "contribution" means can differ between systems but those differences are unimportant for our motivating discussion.

2.1 Three Approaches to Provenance Capture

Provenance capture is the problem of enumerating all provenance relationships between input elements and output elements in a user-defined procedure. The overarching challenges of provenance in array workflows have been described in a number of surveys Davidson and Freire [2008], but we focus our effort on the capture problem. Let us assume that we are given some such procedure F that operates on an array X and returns a new array Y. There are three primary approaches to capture provenance relationships between elements in Y and X.

Approach 1. Operation Constraints. One approach is to have a closed-world of operations that an analys might use through a domain-specific language or pattern matching known operations. In other words, the function `f(X)` above is a composition of sub-functions where the provenance relationships are already known. This approach has had a considerable amount of research in machine learning applications. For example, the Lima system tracks provenance for machine learning workflows written in SystemDS Phani et al. [2021]. Similarly, tools like MLInspect Grafberger et al. [2021] use pattern matching to identify certain function structures in the Python AST. Several other systems employ a similar approach Nikolic et al. [2014], Boehm et al. [2016]. Other work limits the scope of the system

to particular types of workflows such as machine learning model training or inference Vartak et al. [2018], Xin et al. [2018], Sellam et al. [2019], Rezig et al. [2020], Shang et al. [2019], Zhang et al. [2017]. While effective, the problems with such an approach are obvious: the tasks at hand might not be machine learning related, researchers may not have the skill-set to port an existing codebase to a new DSL, or might use custom operations for functionality/performance.

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 Kuehn et al. [2008], Goecks et al. [2010]. 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 Wu et al. [2013]. 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. Other systems have taken a similar laissez-faire approach to provenance capture where they provide a scalable service that the user can populate with metadata Hellerstein et al. [2017], Zhang et al. [2017], Namaki et al. [2020].

Approach 3 (Ours). Annotated Execution. In contrast, DSLog automatically captures and logs data type-level accesses made by an array-programming framework by tracking the read, write, and copy operations made to individual array elements and correlating these accesses with the system call stack. This is possible because arrays have a fixed data type and consistent indexing so low-level accesses can be translated into individual element manipulations. The user does not need to do anything to use DSLog as all tracking happens at a low-level. This approach is analogous to physical logging in a database system Haerder

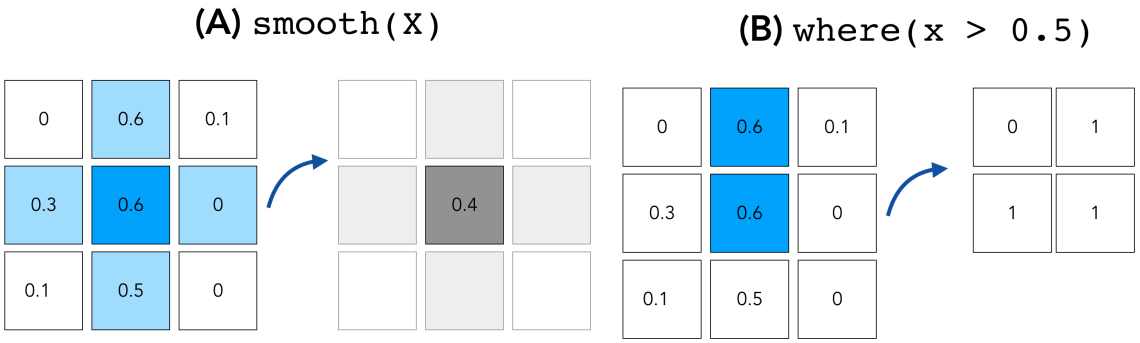

Figure 2.1: A visual description of two non-standard array operations that are common in imaging and scientific applications.

and Reuter [1983]. The key downsides are the computational overheads from such tracking and the storage overhead from low-level physical tracking. We show that with a combination of systems tricks the capture and storage of such logs can be accelerated to mitigate these overheads.

Physical v.s. Logical: Point of Clarification. In these approaches, we use word *physical* carefully to denote the capturing methodology and not the storage model. One can capture logical provenance (the function descriptions) but store it physically (materialize each element-level relationship). A number of provenance systems rely on function descriptions to capture provenance but store it physically Widom [2004], Ikeda et al. [2011]. On the other hand, by "physical" we mean that the provenance is captured at the element and index level agnostic to the function that is calling it. The only thing that we use the function description for is to avoid recapturing previously captured provenance.

2.2 Motivating Application

Consider a sensing application where an imaging system returns a 2D array of pixels, where each value in a range 0 to 1 corresponds to an intensity. Such images very common in industrial part defect detection, thermal imaging, and astronomy. The user is interested in identifying pixel regions where the intensity is significantly higher than its neighbors. We

can express this logic in a simplified array programming pipeline:

```
def hotspot(X):
    X = smooth(X)
    Y = where(X > 0.5)
  return Y
```

`smooth(X)` is a hypothetical function that applies a smoothing filter over the image (e.g., a mean of window around the pixel), and `where(condition)` returns an array of index locations where the condition is true — if there are k such pixels, the array is of size $k \times 2$ where each row corresponds to a pixel location. These operations are illustrated in Figure 2.1. In terms of provenance, a user may want to look at a row in Y and know what input pixels in X may have contributed to it – which is simply a window around the pixel internally defined in `smooth`.

While conceptually simple, such a pipeline causes a lot of difficulty for existing array provenance systems. Firstly, the `where()` [1] is a non-standard function whose behavior is highly input data dependent. Unlike in linear algebra operations, there isn't a provenance relationship that is purely a function of input and output indexes. This makes it hard to apply the "operation constraint" approaches mentioned before, and would default to an uninformative all-to-all relationship in most frameworks. Second, the `smooth()` is actually what determines the key provenance relationships. An specification based approach would require a user to either provide specifications for every possible `smooth()` implementation, or change the specification for `hotspot()` each time. In these types of scenarios, we need an *annotated execution capture system*.

1. Inspired by `numpy.argwhere`

2.3 Related Work

Beyond the aforementioned systems, there are a number of other relevant research areas.

Provenance Compression. There is some prior work in provenance compression in its own right. Xie et al. studied compression provenance graphs for web search and semantic web applications Xie et al. [2011, 2012]. 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 Chapman et al. [2008]. 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 Olteanu [2011]. 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 Chen et al. [2017], Zhou et al. [2012]. Again, due to differences in the problem setting, they are not directly comparable to DSLog.

ML Workflow Tracking. There is also a tangentially related field of machine learning workflow management. Initial prototypes such as HELIX Xin et al. [2018], Alpine Meadow Shang et al., MLFlow Zaharia et al. [2018], and the Collaborative Optimizer Derakhshan et al. [2020] 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.

CHAPTER 3

DSLOG: SYSTEM DESIGN

We present DSLog, a system for low-level provenance capture. DSLog is designed to track fine-grained provenance for array workflows and is integrated with the Python `numpy` library. DSLog tracks automatic cell level provenance across all array operators, and defines a modular schema that supports storage and processing optimizations. This section overviews the system, its capabilities, and our key design choices.

3.1 Interface

Suppose, we have a function F that takes as input a set of multidimensional input arrays and outputs a multidimensional array. Users can register such a function to DSLog with function decorators:

```
@provenance_track
def F(X1,...,XN):
```

This means that DSLog will wrap around the function and re-write it in a way that we can track the contributions of each input element through the function until the output.

The final output the annotated execution is a bipartite provenance graph that relates output elements in Y to a set of contributing input elements in each X_i with edges of the form:

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

A forward query finds all output cells that a particular input contributed to. A reverse query finds all input cells that contributed to a particular output.

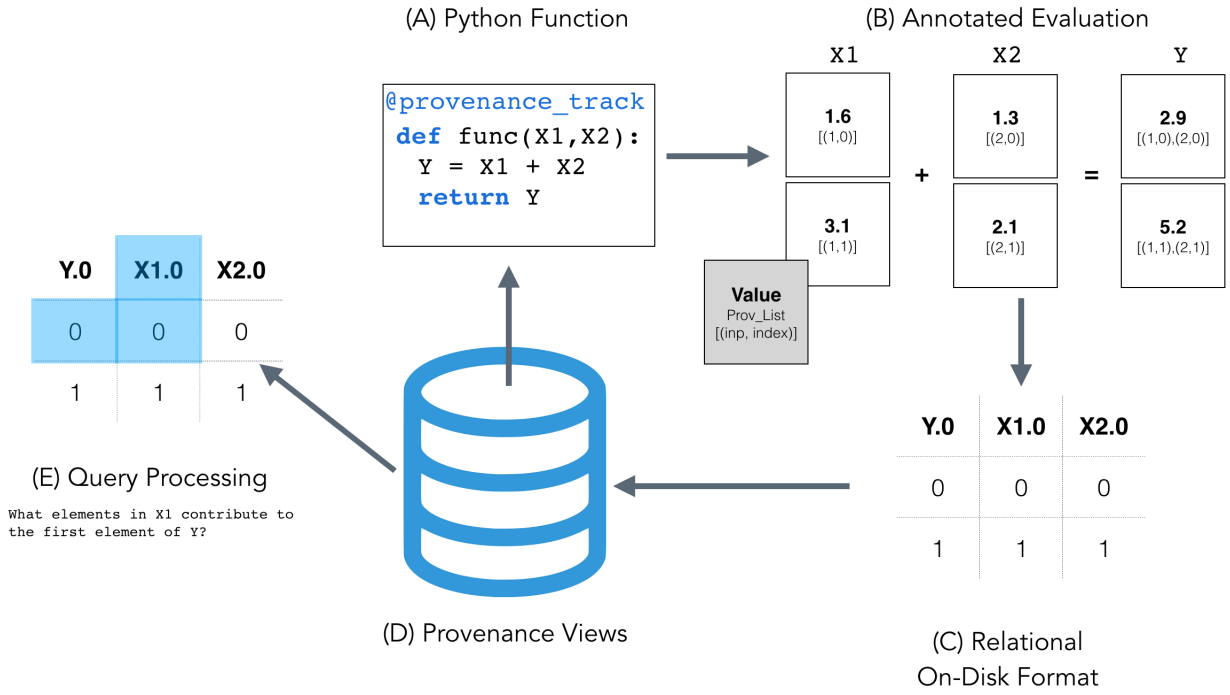

Figure 3.1: The system architecture of DSLog. The user defines Python functions composed of numpy operations (A), and these are evaluated in an annotated setting that aggregates contributing elements (B). Final list of provenance relationships are flushed to disk in a relational model (C), and the result is stored in an indexed database of "views" (D). The color coded number correspond to key optimizations in our system described in the text below.

3.2 Capture Workflow

Figure 3.1 illustrates the under-the-hood workflow that occurs in DSLog during a registered function call. This function call will be evaluated with annotated execution. As an easy example, we have a simple addition of two vectors (Figure 3.1A). The basic data structure in DSLog is a *provenance view* this is a relational table over the output and input cells with schema $R(b_1, ..., b_l, a_1, ..., a_k)$ where each row represents a contribution relationship captured through annotated executions. User's can query these views with standard SQL to answer forward and reverse queries. The capture workflow works as follows:

1. The function signature is checked against all the provenance views in the system. If the provenance is already known for the the function, capture is skipped and a duplicate of the view is made. Provenance views are discussed in Section 5, and there is subtlety in how views are generalized between functions of slightly different arguments.

2. If no view is returned, the function runs with annotated execution (Figure 3.1B) to generate the full provenance with low-level tracking. The details of annotated execution are described in Section 3.3.

3. The provenance resulted from annotated tracking can be compressed asynchronously in a relational representation (Figure 3.1C). In Section 4, we present the compression algorithm, `ProvRC` that takes advantage of the spatial regularities in array operations. This compressed provenance can be used to compute future provenance views.

To summarize, there are three architectural components that to the best of our knowledge have not been applied in array provenance problems: (1) a relational representation for storing the final input-to-output relationships on disk, (2) a compression algorithm that exploits common structures in numerical arrays, and (3) a framework for generalizing physical captures to different function arguments.

3.3 Annotated Execution

We will describe the main mechanism we use to capture provenance during annotated execution. The basic idea of annotation-based capture is to embed tracking information in every element so that when it is manipulated the tracking information is propagated through to the final result. We focus on arrays with `double` data types. We assume that every user-defined function takes as input a set of multidimensional `double` arrays and returns a set of multidimensional `double` array s(or cast appropriately to make this so).

Suppose, we extended the `double` type to be an `annotated_double`, which is a struct of a `double` data value and a set of annotations. Now, let `op` be an n-array operation over `double` values — it takes n `double` values as input and outputs a single `double`. The `annotated_double` data type has the following semantics for `op`:

```
def apply(op, others):
    v = [o.data for o in others]
    a = [o.annotations for o in others]
    return annotated_double(op(v), union(a))
```

where it applies the function to all of the data values and takes a union of all of their annotations. With this basic structure, we can simply override all of the primitive numerical operations used to manipulate `double` data types such as addition, subtraction, multiplication, division, etc. With the custom data type API in `numpy` [1], there are a relatively small number of operations that have to be overridden to make such an annotated double compatible with a vast majority of the library.

We use these data types track function provenance over arrays as follows. Given a function and set of input arrays, each array is converted into a `annotated_double` array. Each element is initialized to have its original value as its data value, and the annotation

1. https://numpy.org/doc/stable/reference/arrays.dtypes.html, https://numpy.org/doc/stable/user/basics.ufuncs.html

is the singleton set of its current position in the input array. We have to further tag these positions with a unique identifier of which input array. As `numpy` operations are applied to the annotated arrays, the provenance accumulates in the elements annotations. Therefore, at the output, each element contains a full history of all of the other elements that contributed to it.

This structure is particularly well-suited for array programming environments because almost all of the core operations are numerical, where if the input is a `double` array the output is also a `double` array or could be cast into one (e.g., a Boolean array). Of course, it is always possible to obfuscate the provenance with operations that change the data type or leverage structures unknown to `numpy`. For example, serializing the entire array into a string, manipulating the string, and the deserializing it. In our first implementation, we ignore such problems. We simply treat the capturing system as a best-effort provenance graph that is complete if a user stays within numerical and `numpy` array operations.

3.3.1 In-Memory Annotation Format

Our implementation written completely in C - for efficient computation and low-level memory management - and linked through Python's C interface. Each annotation tuple is represented with 3 32 bit integers. Its first value is the array id and its second and third values are array indices. This defines an unique cell for an array up-to 2-d dimensions (though the basic strategy extends to any dimensionality). These tuples are laid out into a C array, and copied during operations. The first value of the array is directly storied in the `annotated_double` data type, and the rest is stored as a pointer to a dynamically allocated memory buffer. For specific functions, our system supports pre-allocating a consistent memory buffer to avoid frequent memory requests.

CHAPTER 4

PROVENANCE DATA MODEL

Once the annotated execution finishes, we flush the annotation results to disk. At this point, there are opportunities for optimization because it can happen asynchronously.

4.1 Relational Representation of Array Provenance

Every provenance relationship can be thought of as an edge in a bipartite graph mapping input elements to output elements $Y[b_1, ..., b_l] \leftarrow X_i[a_1, ..., a_k]$. Such a graph is elegantly represented as a relation. Let X_i 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_i 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 $Y[b_1, b_2, ..., b_l] \leftarrow X_i[a_1, a_2, ..., a_m]$.

Table 4.1 shows the provenance relationship of `Y = np.sum(X, axis = 1)` over an array of size $(2, 2)$. 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 reverse queries can simply be cast as SQL queries over such a table.

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 these tables with a natural join operation, similar to that in Interlandi et al. [2015]. Given R_f is provenance of $X \rightarrow Y$ and R_g is the provenance of$Y \rightarrow Z$, we can find $X \rightarrow Z$ from $R_f \bowtie R_g$. Likewise, the reverse is also true. Any lossless join

```
Y = np.sum(X,axis=1)
```

b_1	a_1	a_2
0	0	0
0	0	1
1	1	0
1	1	1

Table 4.1: Example of relational provenance representation for a $2x2$ array.

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

decomposition over $R = S \bowtie T$ can still answer forward and reverse queries.

While the basic ideas of relational representations of provenance graphs have been studied before Chapman et al. [2008], Olteanu [2011], Interlandi et al. [2015], to the best of our knowledge such representations have not been used to support array programming. The relational model acts as a universal data format that abstracts the capture methodology away from the downstream query processing. The same relations could have been populated by operation constraint or operation specification frameworks as well. We see black-box capture as filling in the gap in cases where such information is not available.

4.2 Motivation: Compressed Provenance Representation

One of the challenges with the low-level relational representation is that the size of the captured provenance can be very large compared to frameworks that simply use operation-level provenance labels.

However, relational modeling gives us a set of well-understood tools to eliminate redundancy in these captures. To understand why relational modeling is so powerful, Table 4.2 illustrates a handful of simple provenance examples over a 3x2 array. All of these examples illustrate non-trivial redundancy patterns. Table 4.2A 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 4.2C) and slicing (Table 4.2B) show similar functional relationships between output and input axes. An observant reader will note that these examples are all Multi-Valued Dependencies.

In real data these relationships will not be as clean. The challenge in the compression algorithm is to exploit any such latent redundancy patterns in the final provenance relations even if they are only there in a subset of the relation. Even a black-box function likely have some spatially coherent structures that can be exploited. Interestingly enough, we find that after compression of simple operations, the final size of the relations is often close to what a logical capture framework would have produced.

4.2.1 Baseline Compression Algorithm

Given the patterns described above, columnar compression is a strong baseline algorithm. We first sort the data in R in lexicographical order from outputs to inputs $b_1, ..., b_l, a_1, ..., a_k$. Then, we partition each column of R. Within each column i, we delta encode the integers $R[i][j] = R[i][j] - R[i][j-1]$. We then compress the resulting column with a state of the art integer entropy encoder Hanzo et al. [2002]. If there are long ranges of indices, e.g., '1,2,3,4,5", these are converted to 1's that are effectively compressed.

```
1  Function RangeGroup(S, c, f):
2      S' ← Table(columns(S))
3      temp_row ← S[1]
4      for row ← S[2] to S[R_S] do
5          s ← f(row, t, c, columns(S))
6          if s ≠ () then
7              temp_row ← s                          /* Merge current row */
8          else
9              INSERT t IN S'
10         end
11     end
12     if t ∉ S' then
13         INSERT t IN S'
14     end
15     yield S'
```

We call this a strong baseline, because again, to the best of our knowledge prior work has not explored this in the context of array programming. In fact, in terms of compression ratio this approach is highly effective. However, it does have some issues downstream. Columnar compression is highly-effective for analytical queries that can entire columns. Unfortunately, both forward and reverse queries are row-oriented queries that are often very selective. The scheme above effectively requires one to decode the entire relation before answering a forward or reverse query. Thus, we explore alternatives that avoid this issue but come close to matching or exceeding the compression performance of columnar compression.

4.2.2 Row-Oriented Range Compression

DSLog introduces a new novel compression algorithm for dealing with relational array provenance named `ProvRC`. Our `ProvRC` algorithm uses two main concepts:

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.

Step 1. Multi-Attribute Range Encoding over Inputs

Now, we are ready to describe the compression algorithm. 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 Pibiri and Venturini [2020]. The basic idea is given a set of integers one represents the set as a union ranges, e.g.,

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

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. For example,

$$\mathsf{range}\left(\begin{array}{|ccc|}\hline b_1 & a_1 & a_2 \\ \hline 1 & 1 & 1 \\ 1 & 1 & 2 \\ 1 & 2 & 1 \\ 1 & 2 & 2 \\ \hline\end{array}\right) = \begin{array}{|ccc|}\hline b_1 & a_1 & a_2 \\ \hline 1 & [1,2] & [1,\,2] \\ \hline\end{array}$$

The decoding can be accomplished by expanding any ranges using a Cartesian product $\{0,1\} \times \{0\} \times \{0,1\}$ and taking the union of all such tuples (related to results in Ciucanu and Olteanu [2015]). We apply a simple heuristic to construct such a multi-attribute range encoding that segments each input 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 set of tuples S. Note that the order within subsequences $[a_1, a_2, ..., a_m]$ and $[b_1, b_2, ..., b_l]$ are arbitrary, but just must be consistent with how we iterate through the

Table 4.3: 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

$\rightarrow$

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

attributes in our algorithm.

Notationally, let s_i be the i^{th} row in S and $s_i[c]$ be the projection of the row onto a set of attributes c (we will use C to denote all attributes). 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 $C \setminus c$, 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.

For each input column, $a_i \in [a_m, \ldots a_1]$, we apply the `RangeGroup` function. 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. Table 4.3 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 as relative value transformation as shown with `AddRelative` in Algorithm 1. For every input attribute, $a_i \in S_{\text{input}}$, and output attribute, $b_j \in column(S)_{\text{output}}$, we append a new column to S named $a_i b_j$ that contains the value $b_j - a_i$.

This transformation is performed, because we found that, there is often a functional relationship in provenance relations where, for a set of rows, X, and some input and output

attributes, a_m, b_n, we have:

$$X[a_m] = X[b_n] - \delta \qquad \delta = X[b_n] - X[a_m]$$

, 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 generate all possible relative columns between input and output columns.

The entire algorithm is summarized in Algorithm 1. We can think of it as running `RangeGroup` with two different helper routines `InputEqual` and `OutputEqual`. In the second phase, we run our `RangeGroup` with the `OutputEqual` function over $b_i \in [b_l, \dots b_1]$. `OutputEqual` allow for matching on relative columns as well as the absolute columns. Our insight is that, for input array dimension, a_i, only one of the columns of $a_i, a_i b_1, \dots a_i b_l$ is needed to determine the dimension index. Therefore, given two rows, r_1, r_2, we can merge the rows, given that $\forall a_i \in C$, a non-empty subset of $a_i, a_i b_1, \dots a_i b_l$ 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 4.4, 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 `RangeGroup` algorithm that didn't exist before. We can compress down to an even more concise representation. Let us give a rough depiction of what `ProvRC` is doing. We are compressing rectangular input blocks generated in Step 1 into higher-level rectangles also include output attributes - on the condition that there are multiple fixed coordinates that can 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 4.4: 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_1b_1	a_2	a_2b_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_1b_1	a_2	a_2b_1
[1,3]	-	0	[1, 2]	-

4.3 Full Examples and Discussion

In Table 4.5, 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.

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) Eppstein [2009]. Instead, the complexity of `ProvRC` is $\mathcal{O}(ND + N \log N))$, 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 $\mathcal{O}(ND)$.

1. To avoid redundancy we only show the provenance of X and not Y

Algorithm 1: Full ProvRC Compression

Function `ProvRC`(S)**:**
  for $a_i \leftarrow [a_m \dots a_1]$ **do**
    `/* Input Range Compression (Step 1) */`
    $S \leftarrow$ `RangeGroup` $(S, a_i,$ `InputEqual`$)$
  end
  `/* Relative Transformation (Step 2) */`
  $S \leftarrow$ `AddRelative` (S)
  `/* Output Range Compression (Step 2) */`
  for $b_j \leftarrow [b_l \dots b_1]$ **do**
    $S \leftarrow$ `RangeGroup` $(S, b_j,$ `OutputEqual`$)$
  end
  yield S
Function `InputEqual`(r_1, r_2, a_i, C)**:**
  $is\_eq \leftarrow \bigwedge_{c \in C, c \neq a_i} r_1[c_i] = r_2[c_i]$
  if is_eq **then**
    $r \leftarrow r_1$
    $r[c] \leftarrow r_1[c] \cup r_2[c]$
  else
    $r \leftarrow ()$
  end
  yield r
Function `OutputEqual`(r_1, r_2, b_i, C)**:**
  $is\_eq1 \leftarrow \bigwedge_{b_j \in C_{\text{output}}, b_j \neq b_i} r_1[b_j] = r_2[b_j]$
  $is\_eq2 \leftarrow \bigwedge_{a_k \in C\text{input}} r_1[a_k] = r_2[a_k] \vee (\bigvee_{b_j \in C_{\text{output}}} r_1[a_k b_j] = r_2[a_k b_j])$
  if $is\_eq1 \wedge is\_eq\_12$ **then**
    $r \leftarrow Row(C)$
    for $c \in C$ **do**
      if $c = b_i$ **then**
        $r[c] \leftarrow r_1[c] \cup r_2[c]$
      else
        $r[c] \leftarrow r_1[c] \cap r_2[c]$
      end
    end
  else
    $r \leftarrow ()$
  end
  yield r
Function `AddRelative`(S)**:**
  $S' \leftarrow$ COPY $(S$
  for $a_i \in column(S)_{input}$ **do**
    for $b_j \in column(S)_{output}$ **do**
      INSERT COLUMN $a_i b_j : S[b_j] - S[a_i]$ IN S'
    end
  end
  yield S'

Function	**Input Array Dimensions**	ProvRC				Generalized Provenance View			
		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)[1]	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)[1]	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 4.5: Examples of compression results on common operations

CHAPTER 5

PROVENANCE VIEWS

The previous section describes a relational data model for representing provenance collected from an array programming framework. This section describes how we can efficiently re-use previously captured provenance in DSLog.

5.1 Exact Provenance Re-Use

Since the capturing process is so expensive, we would like to utilize prior knowledge whenever possible to avoid re-capturing the same function. Typical workloads in data science are highly repetitive with the same function repeatedly called. A researcher training a machine learning model is likely to test the preprocessing pipeline multiple times during debugging; hence running many of the same functions on the same data.

In our relational model, we can think of each capture as a "view definition" that represents a provenance relation. The view definition is parametrized by $\{f, \{X\}, \{x\}\}$, where f is the function, $\{X\}$ is the set of input arrays, and $\{x\}$ is the set of other arguments.

$$\mathsf{view}(f, \{X\}, \{x\}) := R(b_1, ..., b_l, a_1, ..., a_m)$$

Note that the same decorator might generate multiple different views based on the inputs to the function.

In the simplest form of result re-use, the parameters $\{f, \{X\}, \{x\}\}$ can be used as a signature to determine whether the exact same function call has been previously issued. The signatures can be hashed to give us a constant lookup-time cache to avoid recomputation. This basic strategy has been employed in systems like Lima Phani et al. [2021]. As we will see in the following, our system goes one step further in term of re-use, by utilizing more general signatures.

5.2 Dimensional and Generalized Views

5.2.1 Dimensional Views.

In a number of important scenarios, the provenance relation depends on only the dimensions of the data and not the actual data itself. For example, the provenance for all linear algebra functions fits this criteria. To make our result re-use component more effective, we support the case where the provenance is only dependent on the dimensions of the array and not the actual values:

$$\mathsf{dim\_view}(f, \{dim(X)\}, \{x\})$$

where $dim(X)$ is a function that outputs the dimensions of X. Now, the view signature can match previously computed captures even if the data was slightly different. Imagine, a forward evaluation of a neural network on a different mini-batch of data.

5.2.2 Generalized Views.

Data generalization is not enough to capture provenance views across programs; we have to be able to create signatures that are independent of array dimensions. We denote provenance that support these signatures as a generalized view:

$$\mathsf{gen\_view}(f, \{x\})$$

To create these views without human intervention, we can leverage our compressed representation as a sort of "model" for how this function behaves. Suppose, that we have already captured the dimensional view, Y, for $(f, \{dim(X)\}, \{x\})$. To create the generalized view, for $d_i \in dim(X)$, we identify all intervals $[1, d_i]$ in Y, and replace them with the generalized interval $[1, D_i]$, where D_i is an indicator of that dimension. If there multiple intervals, we scale the boundaries by the change in dimension.

5.2.3 Automatic Materialization.

Now, DSLog automatically attempts to recognize when it is correct to materialize dimension and generalized views. It is not possible to perfectly calculate this for every case without full logical analysis of functions. However, we can make a best guess by using heuristics to extrapolating based on past behavior. To extrapolate, DSLog generates a candidate dimension and generalized view, whenever the full annotated execution of a function is ran. If m instances of the same view with the same signature is observed, it is added to our cache of materialized views. In our current implementation, we set $m = 2$. With generalized views we set the condition that those views are re

5.2.4 Irrelevant Provenance Arguments.

There may be scalar arguments in $\{x\}$ that do not affect the provenance. For example, in `numpy's sort` function, the parameter, `kind` determines sorting algorithm, but has no effect on the final sorted array. Irrelevant provenance arguments, such as `kind`, leads to multiple provenance signatures that map to duplicate provenance views within the same function. This is space inefficient, especially when storing generalized views that persistent indefinitely across programs.

DSLog automatically finds these arguments as follows. Given a scalar argument $x \in \{x\}$, DSLog define x as an irrelevant provenance argument if all function signatures that only differ in x have the same materialized provenance view, and there exists n function signatures that differ in x and have the same materialized provenance view. Irrelevant provenance arguments are checked for after every view materialization, and removed from function signatures if found.

5.2.5 Examples of Generalized Views.

Table 4.5 shows a selection of generalized views for different functions in the "Generalized Provenance View" column. From the table, we observe that generalized views create compact representations for one-to-one element-wise functions (1) (2), axis aggregates (3) (4), and axis pivots (5). I.e. the generalized view provide exactly defines these patterns of provenance in an universal and parseable format.

	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	MatVecMul	dot	X: (1000, 1000), Y: (1000,)	Z: (1000)	linear algebra matrix-vector product
7	VecMul	dot	X: (1000,), Y: (1000,)	Z: (1,)	linear algebra vector dot product
8	RandFilter	-	X: (1000000, 1)	Z: (1000000, 1)	filter elements greater than average on a random array
9	CorrFilter	-	X: (1000000, 1)	Z: (1000000, 1)	filter elements greater than average on a correlated array
10	ImgFilter	-	X: (1000, 1000)	Z: (1000, 1000)	filter non-zero elements on upscaled MNIST image
11	RandBin4	-	X: (1000000, 1)	Z: (4,)	divide a randomly generated array into 4 bins
12	SortBin4	-	X: (1000000, 1)	Z: (4,)	divide a sorted array into 4 bins

Table 5.1: Description of Operations Evaluated in DSLog

CHAPTER 6

EXPERIMENTS

Our experiments were performed on Chameleon, a large scale computer science research platform Cloud [2022], with a Intel Xeon Gold 6126 Processor and 192 GiB of RAM. All our experiments, including baselines, are single-threaded. We performed detailed experiments on the three main components of DSLog: annotated execution, provenance compression, and materialized views.

6.1 Feasibility of Annotated Execution

The objective of the first experiment is to demonstrate that annotated execution is a feasible technique for capturing provenance. We illustrate the overheads (additional cost over normal execution) of capturing this provenance.

6.1.1 Experiment Design

We simulate the example in the introduction where a user wants to understand the provenance relationships in contributor code in an array programming library for these array operations. We select 12 `numpy` operations that range from typical in prior work (linear algebra) to less well-studied (histogram construction). DSLog does not know how these operations behave beforehand. Seven of the operations are data independent (meaning the provenance is independent of the values of the array) and five are data-dependent. In Table 5.1, we these operations are summarized in detail. For the data-dependent operations, we will briefly describe the content of their array to give an intuition to what each operation's provenance relation is. $X_{\text{RandFilter}}$ and X_{RandBin4} are drawn randomly from $[0, 1]$. For every cell, $c[i]$, in $X_{\text{CorrFilter}}$, we have $c[i] = \mathcal{N}(i, m/10)$, where $m = 100000000$ is the first dimension size. We also draw $X_{\text{SortedBin4}}$ from $[0, 1]$, but then sort the elements physically.

$X_{\text{ImgFilter}}$ is an up-scaled MNIST image of a handwritten digit that is 0 for all elements in the background.

The primary metric is overhead over standard `numpy` execution. For additional microbenchmark performance tests, we found consider two function types: a one-to-one element-wise operation, and one axis aggregation functions. These show extremes in terms of how many input cells correlated to output cells - only a single input in the first case, and along a whole dimension in the second case.

6.1.2 Capture Time

In our first experiment, we evaluate our annotated execution environment on our workload. Note, we did not modify the `numpy` functions in any way to capture this information other than registering them to DSLog. Figure 6.1 shows the results of our workload executed over 1M element input arrays. We find that the cost of annotated execution is steep in relative terms (sometimes 40x slower than the standard execution). However, in absolute terms, the differences are relatively small considering the debugging (less than 100ms for every operation other than o5) and introspection benefits provenance provides.

6.1.3 Micro-Benchmark Baselines

Secondly, we compared the performance of DSLog on the annotated execution to two baseline systems. To the best of our knowledge, there doesn't currently exist another system with this type of annotation for direct comparison so this set of experiments serves more as an ablation study.

- **Python Baseline.** A Python-implemented baseline and that supports the exact behavior of element annotation as our DSLog implementation. It tracks all provenance with Python bindings and function overrides in numpy.

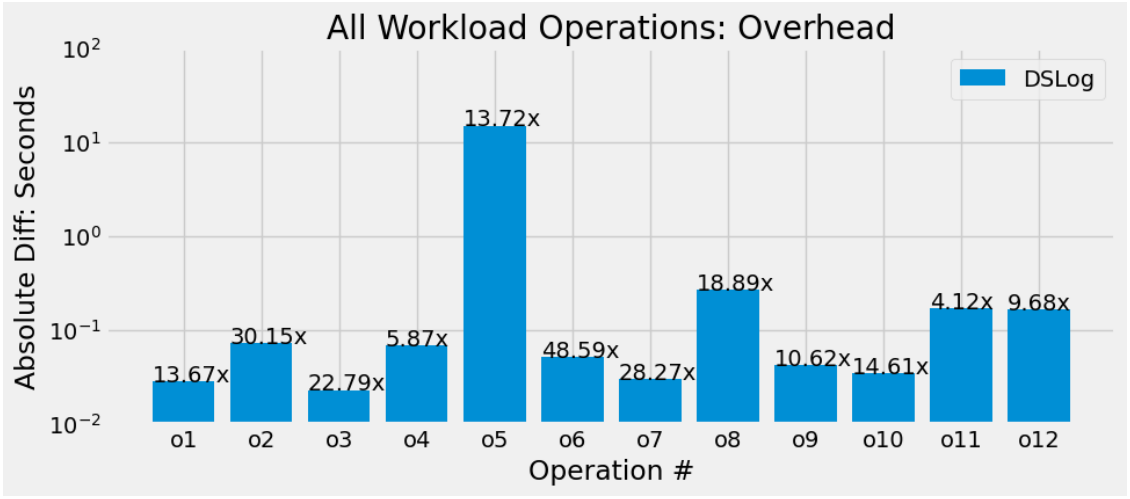

Figure 6.1: The overhead of annotated execution on 1M cell inputs. Annotated execution adds a significant overhead in relative terms to fast operations, but is still small in absolute terms.

- **C Baseline.** A baseline that extends the Python baseline but uses C data types to track the provenance. It supports the exact behavior of DSLog but without our optimization of a pre-allocated buffer (which guesses at how big the provenance of an element will be).

6.1.4 Deep Diving Performance

We show that baseline approaches cannot even match the high overheads seen in the last experiment – they are simply infeasible at any realistic scale. Figure 6.3 shows the cost of execution for baselines of over an element-wise function, and an aggregation function. Figure 6.3(A) shows the "easy case" of this problem, where the amount of annotations for each output element is minimal (only a single input). In this case, the C baseline and DSLog have equivalent performance and thus we omit the C baseline. On the element-wise function, DSLog performs up 275x faster than the Python baseline, but is still up to 5x slower than the bare function.

Figure 6.3(B) is more interesting and shows the other extreme of annotated execution,

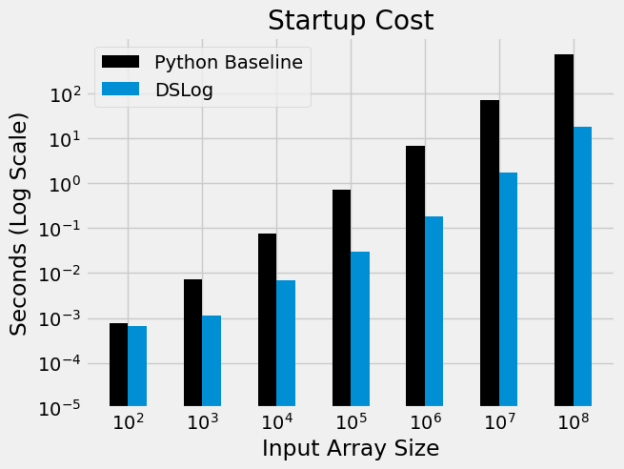

Figure 6.2: To use annotated execution, input arrays need to be converted to annotated arrays. Without optimization, this can be a significant overhead.

where each output element contains annotations from input cells along a whole axis. Figure 6.3(B) shows that without optimizations the baselines simply fail at large scales, we even see this for our C baseline without our optimization of a pre-allocated buffer. This is because they re-allocate new memory for every binary operation that occurs within reduce, leading to memory space overuse.

On the aggregate function, DSLog performs up to 34000x faster than the Python baseline, but is up to 44x slower than running without annotation. We have demonstrated that detailed low-level memory management can lead to a marked improvement over baseline, but annotations on every cell is inherently costly. We also note that DSLog is able to scale to large array sizes more efficiently than the baselines. It is able to maintain a consistent relative overhead on element-wise functions, while relative overhead in the Python implementation increases dramatically.

6.1.5 Function Registration Cost

A hidden cost in annotated execution is initialization time, or the time needed to convert the input arrays from their numerical data types into their annotated data types (one can

actually think of this as a running an additional element-wise function!). This was included in our workflow experiment but not captured in the micro-benchmarks. Figure 6.2 shows cost of annotation initialization for both the Python baseline and DSLog's implementation (the annotated initialization cost for the C-baseline would be same as DSLog). This includes the time to convert the array's data type from `double` to `annotated_double` and the time to fill the initial annotation with each cell's current index. We can see from the figure that, DSLog performs an order of magnitude better than the Python baseline. The Python baseline requires initialization of a full Python object for each cell. In contrast, DSLog directly implements a `numpy` data type that is stored within the array – this is interfaced to Python with a C API. Results suggest that the overhead of dealing with the Python object is responsible for most of this performance difference. Both implementations have a performance that is roughly linear with respect to the number cells, which is expected since filling the initial annotation requires iteration over all cells.

6.1.6 Summary

To conclude, annotated execution is relative expensive but it might be worth it as the only solution to debugging black-box functions. Typical functions will not be as bad as the aggregate shown in our experiments (having a smaller "blast radius") of provenance. Additionally, relative performance might not be significant on smaller arrays (¡1M) because the operation themselves are still performed quickly. Note, that by using provenance views as described in the paper, one only has to incur this cost once per function call with the same or similar arguments.

6.2 Compression

Our compression algorithm, `ProvRC`, is evaluated alongside two axes: compression ratio and compression latency. We deemed that the compression ratio of `ProvRC` is more important,

since the compression latency is asynchronous. However, we wish to show that this latency is within range of baseline compression algorithms used in practise.

6.2.1 Experiment Design

To evaluate storage, our compression algorithm was ran over the raw provenance result from the annotated results of the 12 operations summarized in Table 5.1 and Section 6.1.1. To evaluate performance, we measured the latency of compressing the two extremes of function types: one-to-one element-wise operation, and one axis aggregation functions.

6.2.2 Baselines

We experimentally compare `ProvRC` against other compression algorithms.

- **gzip** We use gzip as a generic compression algorithm that is unaware of the typical structures in provenance relations.
- **SubZero** SubZero introduced the concept of *Region Lineage*, which are pairs of sets that contain all-to-all relationships. This is subtly different from our range encoding, which describes one-to-all relations over a range of cells. Since SubZero is implemented over SciDB, we present our best effort reproduction of their paper for `numpy` ndarrays.
- **Columnar** We use the columnar compression described in Section 4.2.1 that uses integer entropy coding on each column Hanzo et al. [2002]
- `ProvRC- No Rel`. We also compare compression ratio against a version of `ProvRC` with only multi-attribute encoding and no relative compression.

In terms of metrics, we consider the compression ratio that measures the compressed size of the provenance versus the raw size of the provenance generated from an annotated execution. We also consider the compression latency as the time needed to fully compress

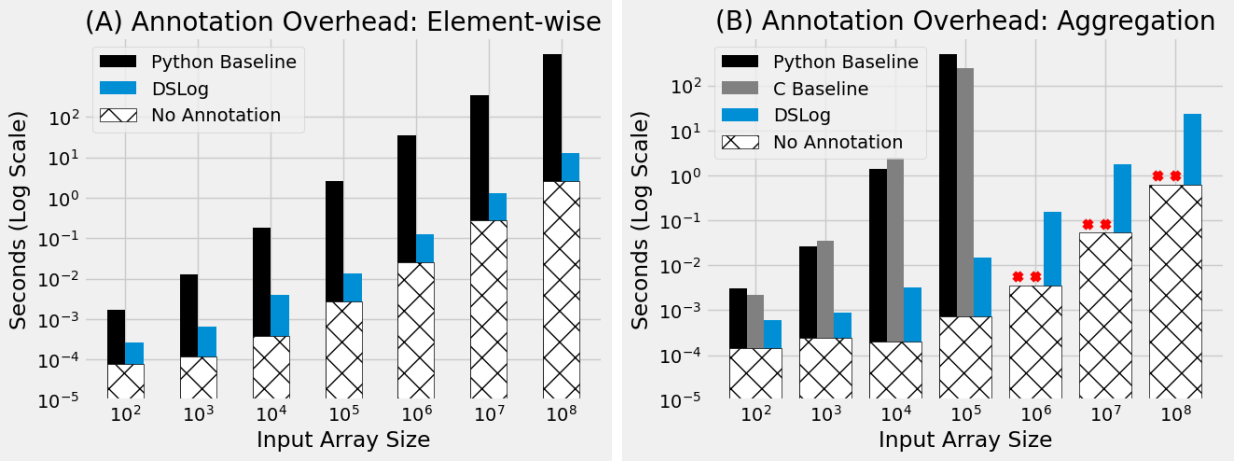

Figure 6.3: (A) The annotation overhead of element-wise operations, (B) the annotation overhead of aggregation operations.

that raw provenance. We exclude SubZero from the performance comparison, since its *Region Lineage* pairs are generated through user-defined functions, and does not perform actually perform compression over the raw provenance (and is more analogous to fetching materialized views in DSLog).

6.2.3 Compression Ratio

In Table 6.1, we show the size of the provenance file on disk for each compression algorithm. Interestingly, each algorithm presented a different pattern on the compression over provenance data. GZIP consistently compressed the file to about 20%-30% of the original size, which is in line with its compression ratio for other files of integers, and a good baseline for a data-agnostic approach. Subzero showed some successful compression on functions with aggregations, but does not show improvements in the other cases. This suggests that the all-to-all model is not a good general representation of provenance for many array functions. Note that Subzero does not prescribe a method for generating provenance pairs, and the compression ratio is dependent on how the user chooses to use this representation. For example, in MatMul, we paired slices of each input array with slices of each output array. Another

valid representation that would lead to much poorer compression would be to compress slices of both input arrays with the one corresponding output cell.

Columnar compression with had consistent good performance over every function, and was the best algorithm for array provenances with low spatial correlation. Turbo's performance indicates that the relational model of provenance leads to column regularity that arithmetic coders can recognize, and is a useful model for efficient compression. `ProvRC` is still able to beat its compression ratio on provenance with higher spatial correlation. Another major downside compared to `ProvRC` is that its output is not parse-able for downstream tasks.

We 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. In cases without spatial regularity that matches those detected in `ProvRC`, the processed file can be relatively worse than the original file, due the additional scaffolding used in `ProvRC` to describe relative indices and ranges. However, the benefit of `ProvRC` is that it doesn't necessarily scale with array size; for example, the provenance used in Table 4.5 for an array of size 1000 to describe the `negative` function is the exact same size as the one generated in our results for an array of size 1 million. `ProvRC` is notably the optimal choice for array provenance with simple spatial regularity, we see up to approximately 10^9x in reduction of storage size with the tested common `numpy` functions.

Name	Raw	GZIP		SubZero		Columnar		ProvRC- No Rel		ProvRC	
	Absolute (MB)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)	Absolute (MB)	Relative (%)
Negative	20.69	5.921	28.62	22.45	108.5	0.1109	0.5361	37.75	182.5	**0.000071**	**0.0003432**
Addition	35.36	9.702	27.43	32.14	90.84	0.2220	0.6276	75.51	213.5	**0.000124**	**0.0003505**
Aggregate	13.50	3.871	28.68	10.76	79.71	0.009108	0.06748	0.03450	0.2593	**0.000085**	**0.0006298**
Repetition	82.75	23.68	28.62	57.14	69.05	0.1509	0.1823	156.27	188.9	**0.000243**	**0.0002936**
MatMul	32000.13 [1]	10960[1]	34.26[1]	39.04	0.1220	9.942[1]	0.03106[1]	0.07198	0.0002249	**0.000126**	**0.0000003937**
MatVecMul	26.24	8.673	33.05	10.59	40.35	0.01548	0.05901	0.03504	0.1335	**0.000162**	**0.0006173**
VecMul	29.74	6.603	22.20	21.74	73.01	0.3101	1.043	0.000094	0.0003161	**0.00023**	**0.0007733**
RandFilter	13.93	4.306	30.91	12.96	92.99	**2.364**	**16.97**	21.24	152.5	21.87	156.9
CorrFilter	13.87	3.791	27.33	12.80	92.32	**1.503**	**10.84**	20.98	151.2	4.940	35.75
ImgFilter	10.78	3.001	27.83	8.242	76.44	1.069	9.910	12.85	1.192	**0.06644**	**0.006162**
RandBin4	14.87	3.553	23.89	3.783	25.442	**0.8164**	**5.490**	13.49	90.71	43.32	291.3
SortedBin4	14.87	3.528	23.73	3.817	25.67	0.3092	2.080	**0.000166**	**0.001116**	0.000468	0.003147

Table 6.1: Comparison of Compression Ratio for Different Algorithms Against ProvRC

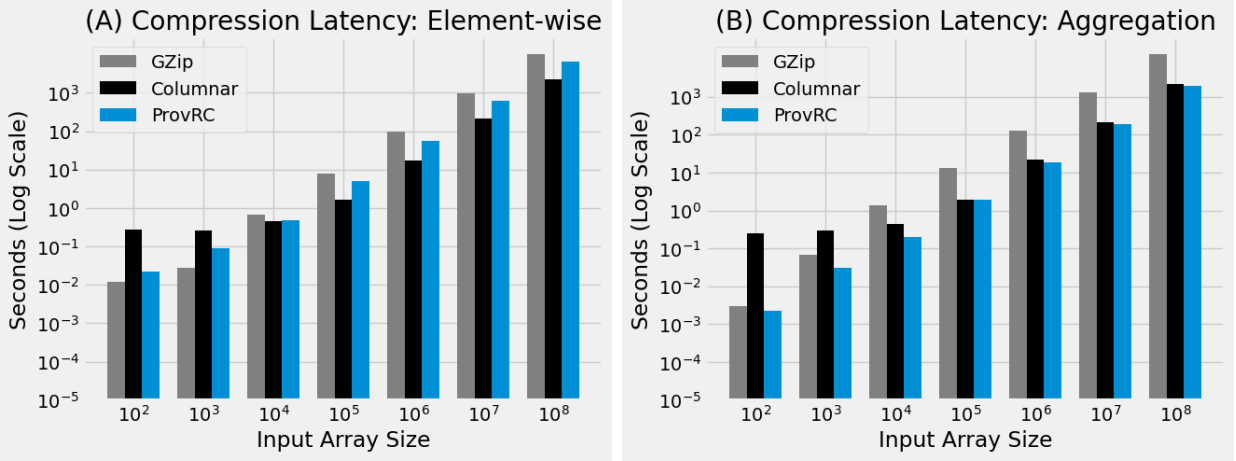

Figure 6.4: (A) and (B) benchmark the compression algorithms in terms of their latency as a function of input size. At large scales, ProvRC and Columnar are largely similar in terms of compression latency, but gzip is about an order of magnitude slower.

6.2.4 Latency

6.2.5 Compression Latency

Figure 6.4 shows the asynchronous latency of compression for GZIP, columnar compression with TurboRC, and `ProvRC` over different input array sizes for element-wise and aggregation functions. Our compression algorithm performed up to 6x better than GZIP in both element-wise and aggregation. Turbo is significantly slower on small arrays, but has similar latency on larger arrays with aggregation lineage, and is faster on element-wise lineage. Considering that `ProvRC` wasn't designed to optimize performance, we found that its latency is comparable to current state-of-the-art and popular solution.

6.3 Materialized Views

The views in DSLog are useful under two conditions: (1) they generate provenance with a lower overhead than annotated execution DSLog, and (2) they have significant coverage over

1. Memory usage for full provenance compression exceeded our machine's limit during compression, so row-level provenance were individually compressed

array-based function signature.

6.3.1 Overhead

The point that materialized views have lower overhead than annotated execution is obvious. For the sake of completion, we evaluated the overhead of finding pre-materialized provenance views compared to running the bare function for element-wise and aggregation functions - given that DSLog first loads a hash map of function signatures to views in memory. These results are not shown in the paper as, even smoothed over 100 runs, the overhead is undetectable over the natural variance in performance in running the functions.

We should aim to materialize and take advantage of repeated function signatures whenever possible, and reduce the number of annotated executions needed. In a sense, this is re-framing what past systems established with operation specifications and operation constraints. However, our contribution is that DSLog generates and uses unconstrained views without human intervention.

Now we discuss the more important consideration of materialized view: can we frequently generate these views for common array operations?

6.3.2 Coverage Experiment Design

We evaluate coverage of DSLog's view framework by assessing a full set of 136 different functions in `numpy` in a simulated user workflow. These functions consists of all the functions in `numpy`'s API that meet the criteria of current supported function signatures: they (1) can intake `annotated_double` arrays and output an `annotated_double`, and (2) only intakes scalar arguments outside of `annotated_double` arrays. For each function, a list of possible scalar arguments are are hand-generated and we decorate each function to run within DSLog's framework. In our simulated workflow, DSLog populates the materialized views over 100 runs. For each run, scalar arguments are randomly selected from our list, and array

Function	Total	`dim_view`	`global_view`	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 6.2: A table showing how many `numpy` API functions are covered in our view generation and accurately characterized.

sizes random picked for each input array. We then perform the annotated execution twice with the selected arguments and array sizes, randomly re-populating the content of the array before each executions. DSLog automatically materializes views as described in Section 5. Each run represents an individual program where the same dimensional function signature might be repeated multiple times. Multiple runs represent use across multiple programs, where the function signature varies with each program.

We evaluate functions with this workflow as follows. We consider that the function is covered by dimensional views, if, after each run, a dimensional view is created for that dimensional signature. We consider that the function is covered by generalized views, if a set of generalized views are created after all 100 runs. A detailed list of all 136 functions has been uploaded onto Github[2]. For each individual function, this includes a list of tested arguments, and whether DSLog was successful in generating the dimensional and generalized views.

As automated view capture 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 dimensional and generalized views.

6.3.3 Coverage Over `numpy` Library

Table 6.2 summarizes the results. 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 dimensional and generalized views. For complex patterns, we are able to generate dimensional and generalization views 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 views to be a significant part of our performance optimization strategy.

The coverage of generalize views is particular interesting outside of optimization. As discussed in Section 5, these views are extrapolated based on our compression framework, and actually suggest that our compression framework is capable of some logical inference. Combined with our experimental results, we show that DSLog is implicitly capable of predicting the underlying logical provenance pattern of the majority of the tested `numpy` functions.

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.

2. https://github.com/j2zhao/DSLog-Coverage-Functions

CHAPTER 7

DISCUSSION AND CONCLUSION

This paper explores techniques to efficiently capture and represent fine-grained data provenance in array-programming frameworks. Our system, DSLog, uses a technique called annotated execution to capture provenance automatically without user specification. The design decisions of DSLog draws from a set of core concepts:

1. User effort should be minimized. Outside of decorating the function, we do not require user any user interaction to generate function provenance.

2. Low-level array annotations are important and can be optimized. We've established that low-level array annotations address the issue of black-box provenance. Unfortunately, these annotations naturally have a significant impact on performance. We believe that, while this impact cannot be mitigated, it can be reduced. These leads to our low-overhead implementation of the `annotated_double` data type.

3. Physical provenance over the entire array should be accessible to the user. This means that the user should be able to easily query our system to know the provenance of every cell in the output array. We currently guarantee this by tracking every cell during annotated execution, and storing the full provenance in a lossless manner with `ProvRC`.

4. **There are similarities in provenance across functions with different provenance patterns.** This is the most important idea in this paper, and leads to our `ProvRC` compression algorithm and generalized provenance views. For example, element-wise functions, and tiling functions both benefit from relative indexing during compression (Table 4.5). Similarly, transpose and matrix multiplication functions can have their provenance be generalized by simply scaling $[1, D_i]$.

As we noted before, we use word *physical* carefully to denote the capturing methodology and not the storage model. One can capture logical provenance (the function descriptions) but store it physically (materialize each element-level relationship). In a sense, the workflow in DSLog does the reverse - creating generalized arrays from array annotations is a specific case of translating physical provenance to logical provenance.

There is an interesting equivalence between operation specification (a form of logical capture) and generalized views. Generalized views are defined on function signatures that are independent of array specifications. Hence, an materialized generalized view is functionally equivalent to the output of an user-defined operation specification. This leads us to conclude that our process in creating generalized views is actually making and encoding a predication on the underlying logic of physically-gathered provenance. `ProvRC` and generalized views provide a method where this encoding could be automatically generated from low-level physical provenance. This contribution is independent of the system metrics of the performance and space efficiency of DSLog. Future work can expand on this concept, and extend the encoding for more functions.

REFERENCES

Tensorflow raw operations.

Sameer Agarwal, Barzan Mozafari, Aurojit Panda, Henry Milner, Samuel Madden, and Ion Stoica. Blinkdb: queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems*, pages 29–42. ACM, 2013.

Sameer Agarwal, Henry Milner, Ariel Kleiner, Ameet Talwalkar, Michael Jordan, Samuel Madden, Barzan Mozafari, and Ion Stoica. Knowing when you're wrong: building fast and reliable approximate query processing systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 481–492. ACM, 2014.

Airbnb. Airbnb new york city open data 2019. `https://www.kaggle.com/dgomonov/new-york-city-airbnb-open-data`, 2019.

Matthias Boehm, Michael W Dusenberry, Deron Eriksson, Alexandre V Evfimievski, Faraz Makari Manshadi, Niketan Pansare, Berthold Reinwald, Frederick R Reiss, Prithviraj Sen, Arvind C Surve, et al. Systemml: Declarative machine learning on spark. *Proceedings of the VLDB Endowment*, 9(13):1425–1436, 2016.

Jaqueline Brito, Korhan Demirkaya, Boursier Etienne, Yannis Katsis, Chunbin Lin, and Yannis Papakonstantinou. Efficient approximate query answering over sensor data with deterministic error guarantees. *arXiv preprint arXiv:1707.01414*, 2017.

Peter Buneman, Sanjeev Khanna, and Tan Wang-Chiew. Why and where: A characterization of data provenance. In *International Conference on Database Theory*, pages 316–330. Springer, 2001.

Doug Burdick, Prasad M Deshpande, TS Jayram, Raghu Ramakrishnan, and Shivakumar Vaithyanathan. Efficient allocation algorithms for olap over imprecise data. In *Proceedings of the 32nd international conference on Very large data bases*, pages 391–402. VLDB Endowment, 2006.

Walter Cai, Magdalena Balazinska, and Dan Suciu. Pessimistic cardinality estimation: Tighter upper bounds for intermediate join cardinalities. In *Proceedings of the 2019 ACM SIGMOD International Conference on Management of Data*, pages 18–35. ACM, 2019.

José Cambronero, John K Feser, Micah J Smith, and Samuel Madden. Query optimization for dynamic imputation. *Proceedings of the VLDB Endowment*, 10(11):1310–1321, 2017.

T. M. Chan. Klee's measure problem made easy. In *2013 IEEE 54th Annual Symposium on Foundations of Computer Science*, pages 410–419, Oct 2013. doi: 10.1109/FOCS.2013.51.

Adriane P Chapman, Hosagrahar V Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 993–1006, 2008.

Chen Chen, Harshal Tushar Lehri, Lay Kuan Loh, Anupam Alur, Limin Jia, Boon Thau Loo, and Wenchao Zhou. Distributed provenance compression. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 203–218, 2017.

James Cheney, Laura Chiticariu, and Wang-Chiew Tan. *Provenance in databases: Why, how, and where*. Now Publishers Inc, 2009.

Xu Chu, Ihab F Ilyas, Sanjay Krishnan, and Jiannan Wang. Data cleaning: Overview and emerging challenges. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 2201–2206. ACM, 2016.

Radu Ciucanu and Dan Olteanu. Worst-case optimal join at a time. Technical report, Technical report, Technical report, Oxford, 2015.

Chameleon Cloud. A configurable experimental environment for large-scale cloud research. chameleoncloud. org. retrieved, 2022.

Graham Cormode, Minos Garofalakis, Peter J Haas, Chris Jermaine, et al. Synopses for massive data: Samples, histograms, wavelets, sketches. *Foundations and Trends® in Databases*, 4(1–3):1–294, 2011.

G Daniel, Johnnie Gray, et al. Opt_einsum-a python package for optimizing contraction order for einsum-like expressions. *Journal of Open Source Software*, 3(26):753, 2018.

Susan B Davidson and Juliana Freire. Provenance and scientific workflows: challenges and opportunities. In *Proceedings of the 2008 ACM SIGMOD international conference on Management of data*, pages 1345–1350, 2008.

Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg. ISBN 978-3-540-78800-3.

Behrouz Derakhshan, Alireza Rezaei Mahdiraji, Ziawasch Abedjan, Tilmann Rabl, and Volker 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.

Daniel Deutch, Zachary G Ives, Tova Milo, and Val Tannen. Caravan: Provisioning for what-if analysis. In *CIDR*, 2013.

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

Ehud Friedgut. Hypergraphs, entropy, and inequalities. *The American Mathematical Monthly*, 111(9):749–760, 2004. ISSN 00029890, 19300972. URL `http://www.jstor.org/stable/4145187`.

Minos N Garofalakis and Phillip B Gibbons. Approximate query processing: Taming the terabytes. In *VLDB*, pages 343–352, 2001.

Lise Getoor and Ashwin Machanavajjhala. Entity resolution: theory, practice & open challenges. *Proceedings of the VLDB Endowment*, 5(12):2018–2019, 2012.

Boris Glavic. Data provenance. *Foundations and Trends® in Databases*, 9(3-4), 2021.

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

Stefan Grafberger, Shubha Guha, Julia Stoyanovich, and Sebastian 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.

Todd J Green, Grigoris Karvounarakis, and Val Tannen. Provenance semirings. In *Proceedings of the twenty-sixth ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 31–40, 2007.

Theo Haerder and Andreas Reuter. Principles of transaction-oriented database recovery. *ACM computing surveys (CSUR)*, 15(4):287–317, 1983.

Alon Halevy, Anand Rajaraman, and Joann Ordille. Data integration: The teenage years. In *Proceedings of the 32nd international conference on Very large data bases*, pages 9–16, 2006.

Lajos Hanzo, Tong Hooi Liew, and Bee Leong Yeap. *Turbo coding, turbo equalisation and space-time coding*. John Wiley & Sons, 2002.

Joseph M Hellerstein. Quantitative data cleaning for large databases. *United Nations Economic Commission for Europe (UNECE)*, 2008.

Joseph M Hellerstein, Peter J Haas, and Helen J Wang. Online aggregation. In *Proceedings of the 1997 ACM SIGMOD International Conference on Management of Data*, pages 171–182, 1997.

Joseph M Hellerstein, Vikram Sreekanti, Joseph E Gonzalez, James Dalton, Akon Dey, Sreyashi Nag, Krishna Ramachandran, Sudhanshu Arora, Arka Bhattacharyya, Shirshanka Das, et al. Ground: A data context service. In *CIDR*. Citeseer, 2017.

Robert Ikeda, Hyunjung Park, and Jennifer Widom. Provenance for generalized map and reduce workflows. 2011.

Tomasz Imieliński and Witold Lipski Jr. Incomplete information in relational databases. In *Readings in Artificial Intelligence and Databases*, pages 342–360. Elsevier, 1989.

Matteo Interlandi, Kshitij Shah, Sai Deep Tetali, Muhammad Ali Gulzar, Seunghyun Yoo, Miryung Kim, Todd Millstein, and Tyson 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.

Noah Johnson, Joseph P Near, and Dawn Song. Towards practical differential privacy for sql queries. *Proceedings of the VLDB Endowment*, 11(5):526–539, 2018.

Srikanth Kandula, Anil Shanbhag, Aleksandar Vitorovic, Matthaios Olma, Robert Grandl, Surajit Chaudhuri, and Bolin Ding. Quickr: Lazily approximating complex adhoc queries in bigdata clusters. In *Proceedings of the 2016 ACM SIGMOD International Conference on Management of Data*, pages 631–646. ACM, 2016.

Robert L Kaufman. *Heteroskedasticity in regression: Detection and correction*. Sage Publications, 2013.

Oliver Kennedy and Boris Glavic. Analyzing uncertain tabular data. In *Information Quality in Information Fusion and Decision Making*, pages 243–277. Springer, 2019.

Tim Kraska. Northstar: An interactive data science system. *Proceedings of the VLDB Endowment*, 11(12):2150–2164, 2018.

Sanjay Krishnan, Jiannan Wang, Michael J Franklin, Ken Goldberg, and Tim Kraska. Stale view cleaning: Getting fresh answers from stale materialized views. *Proceedings of the VLDB Endowment*, 8(12):1370–1381, 2015a.

Sanjay Krishnan, Jiannan Wang, Michael J Franklin, Ken Goldberg, Tim Kraska, Tova Milo, and Eugene Wu. Sampleclean: Fast and reliable analytics on dirty data. *IEEE Data Eng. Bull.*, 38(3):59–75, 2015b.

Heidi Kuehn, Arthur Liberzon, Michael Reich, and Jill P Mesirov. Using genepattern for gene expression analysis. *Current protocols in bioinformatics*, 22(1):7–12, 2008.

Arun Kumar, Matthias Boehm, and Jun Yang. Data management in machine learning: Challenges, techniques, and systems. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 1717–1722, 2017.

Xi Liang, Zechao Shang, Sanjay Krishnan, Aaron J Elmore, and Michael J Franklin. Fast and reliable missing data contingency analysis with predicate-constraints. In *Proceedings of the 2020 ACM SIGMOD International Conference on Management of Data*, pages 285–295, 2020.

Matteo Magnani and Danilo Montesi. A survey on uncertainty management in data integration. *Journal of Data and Information Quality (JDIQ)*, 2(1):1–33, 2010.

Alexandra Meliou and Dan Suciu. Tiresias: the database oracle for how-to queries. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*, pages 337–348. ACM, 2012.

Alexandra Meliou, Wolfgang Gatterbauer, and Dan Suciu. Reverse data management. *Proceedings of the VLDB Endowment*, 4(12), 2011.

Barzan Mozafari. Approximate query engines. 2017.

Mohammad Hossein Namaki, Avrilia Floratou, Fotis Psallidas, Subru Krishnan, Ashvin Agrawal, and Yinghui Wu. Vamsa: Tracking provenance in data science scripts. *CoRR*, abs/2001.01861, 2020. URL `http://arxiv.org/abs/2001.01861`.

Hung Q Ngo, Ely Porat, Christopher Ré, and Atri Rudra. Worst-case optimal join algorithms. *Journal of the ACM (JACM)*, 65(3):16, 2018.

Milos Nikolic, Mohammed Elseidy, and Christoph 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.

The Bureau of Transportation Statistics. Border crossing dataset. `https://www.kaggle.com/akhilv11/border-crossing-entry-data`, 2019.

Chris Olston, Boon Thau Loo, and Jennifer Widom. Adaptive precision setting for cached approximate values. In *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, pages 355–366, New York, NY, USA, 2001. ACM. ISBN 1-58113-332-4. doi: 10.1145/375663.375710. URL `http://doi.acm.org/10.1145/375663.375710`.

Dan Olteanu. On factorisation of provenance polynomials. In *In TaPP*. Citeseer, 2011.

Stavros Papadopoulos, Kushal Datta, Samuel Madden, and Timothy Mattson. The tiledb array data storage manager. *Proceedings of the VLDB Endowment*, 10(4):349–360, 2016.

Wei Hong Peter Bodik et al. Intel wireless dataset. `http://db.csail.mit.edu/labdata/labdata.html`, 2004.

Arnab Phani, Benjamin Rath, and Matthias 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.

Giulio Ermanno Pibiri and Rossano Venturini. Techniques for inverted index compression. *ACM Computing Surveys (CSUR)*, 53(6):1–36, 2020.

Viswanath Poosala, Peter J Haas, Yannis E Ioannidis, and Eugene J Shekita. Improved histograms for selectivity estimation of range predicates. *ACM SIGMOD Record*, 25(2): 294–305, 1996.

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

Pew Research. Why 2016 election polls missed their mark. https://www.pewresearch.org/fact-tank/2016/11/09/why-2016-election-polls-missed-their-mark/, 2004.

El Kindi Rezig, Lei Cao, Giovanni Simonini, Maxime Schoemans, Samuel Madden, Nan Tang, Mourad Ouzzani, and Michael 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.

Stefano Rizzi and Enrico Gallinucci. Cubeload: a parametric generator of realistic olap workloads. In *International Conference on Advanced Information Systems Engineering*, pages 610–624. Springer, 2014.

Thibault Sellam, Kevin Lin, Ian Huang, Michelle Yang, Carl Vondrick, and Eugene Wu. Deepbase: Deep inspection of neural networks. In *Proceedings of the 2019 International Conference on Management of Data*, pages 1117–1134, 2019.

Zeyuan Shang, Emanuel Zgraggen, and Tim Kraska. Alpine meadow: A system for interactive automl.

Zeyuan Shang, Emanuel Zgraggen, Benedetto Buratti, Ferdinand Kossmann, Philipp Eichmann, Yeounoh Chung, Carsten Binnig, Eli Upfal, and Tim 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.

Emad Soroush, Magdalena Balazinska, and Daniel Wang. Arraystore: a storage manager for complex parallel array processing. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of data*, pages 253–264, 2011.

R STEVEN, F SARAH, and G RONALD. Ipums usa: Version 9.0 [dataset], 2019.

Michael Stonebraker, Paul Brown, Alex Poliakov, and Suchi Raman. The architecture of scidb. In *International Conference on Scientific and Statistical Database Management*, pages 1–16. Springer, 2011.

Dan Suciu. *Probabilistic databases*. Springer, 2009.

Liwen Sun, Michael J Franklin, Sanjay Krishnan, and Reynold S Xin. Fine-grained partitioning for aggressive data skipping. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 1115–1126. ACM, 2014.

Bruhathi Sundarmurthy, Paraschos Koutris, Willis Lang, Jeffrey Naughton, and Val Tannen. m-tables: Representing missing data. In *20th International Conference on Database Theory (ICDT 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

Bruhathi Sundarmurthy, Paraschos Koutris, and Jeffrey Naughton. Exploiting data partitioning to provide approximate results. In *Proceedings of the 5th ACM SIGMOD Workshop on Algorithms and Systems for MapReduce and Beyond*, BeyondMR'18, pages 5:1–5:5, New

York, NY, USA, 2018. ACM. ISBN 978-1-4503-5703-6. doi: 10.1145/3206333.3206337. URL http://doi.acm.org/10.1145/3206333.3206337.

Manasi Vartak, Joana M F. da Trindade, Samuel Madden, and Matei 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.

Jiannan Wang, Sanjay Krishnan, Michael J Franklin, Ken Goldberg, Tim Kraska, and Tova Milo. A sample-and-clean framework for fast and accurate query processing on dirty data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 469–480. ACM, 2014.

Yisu Remy Wang, Shana Hutchison, Jonathan Leang, Bill Howe, and Dan Suciu. Spores: sum-product optimization via relational equality saturation for large scale linear algebra. *arXiv preprint arXiv:2002.07951*, 2020.

Jennifer Widom. Trio: A system for integrated management of data, accuracy, and lineage. Technical report, Stanford InfoLab, 2004.

Eugene Wu, Samuel Madden, and Michael 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.

Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Darrell DE Long, Ahmed Amer, Dan Feng, and Zhipeng Tan. Compressing provenance graphs. In *3rd USENIX Workshop on the Theory and Practice of Provenance (TaPP 11)*, 2011.

Yulai Xie, Dan Feng, Zhipeng Tan, Lei Chen, Kiran-Kumar Muniswamy-Reddy, Yan Li, and Darrell DE 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.

Doris Xin, Stephen Macke, Litian Ma, Jialin Liu, Shuchen Song, and Aditya Parameswaran. Helix: Holistic optimization for accelerating iterative machine learning. *arXiv preprint arXiv:1812.05762*, 2018.

Matei Zaharia, Andrew Chen, Aaron Davidson, Ali Ghodsi, Sue Ann Hong, Andy Konwinski, Siddharth Murching, Tomas Nykodym, Paul Ogilvie, Mani Parkhe, et al. Accelerating the machine learning lifecycle with mlflow. *IEEE Data Eng. Bull.*, 41(4):39–45, 2018.

Qing Zhang, Nick Koudas, Divesh Srivastava, and Ting Yu. Aggregate query answering on anonymized tables. In *2007 IEEE 23rd International Conference on Data Engineering*, pages 116–125. IEEE, 2007.

Zhao Zhang, Evan R Sparks, and Michael 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.

Wenchao Zhou, Suyog Mapara, Yiqing Ren, Yang Li, Andreas Haeberlen, Zachary Ives, Boon Thau Loo, and Micah Sherr. Distributed time-aware provenance. *Proceedings of the VLDB Endowment*, 6(2):49–60, 2012.