SubZero: A Fine-Grained Lineage System for Scientific Databases

Eugene Wu, Samuel Madden, Michael Stonebraker

CSAIL, MIT
32 Vassar St, Cambridge, MA, USA 02139
{sirrice, madden, stonebraker}@csail.mit.edu

Abstract— **Data lineage is a key component of provenance that helps scientists track and query relationships between input and output data. While current systems readily support lineage relationships at the file or data array level, finer-grained support at an array-cell level is impractical due to the lack of support for user defined operators and the high runtime and storage overhead to store such lineage.**

We interviewed scientists in several domains to identify a set of common semantics that can be leveraged to efficiently store fine-grained lineage. We use the insights to define lineage representations that efficiently capture common locality properties in the lineage data, and a set of APIs so operator developers can easily export lineage information from user defined operators. Finally, we introduce two benchmarks derived from astronomy and genomics, and show that our techniques can reduce lineage query costs by up to 10× while incuring substantially less impact on workflow runtime and storage.

I. Introduction

Many scientific applications are naturally expressed as a workflow that comprises a sequence of operations applied to raw input data to produce an output dataset or visualization. Like database queries, such workflows can be quite complex, consisting up to hundreds of operations [1] whose parameters or inputs vary from one run to another.

Scientists record and query provenance – metadata that describes the processes, environment and relationships between input and output data arrays – to ascertain data quality, audit and debug workflows, and more generally understand how the output data came to be. A key component of provenance, *data lineage*, identifies how input data elements are related to output data elements and is integral to debugging workflows. For example, scientists need to be able to work backward from the output to identify the sources of an error given erroneous or suspicious output results. Once the source of the error is identified, the scientist will then often want to identify derived downstream data elements that depend on the erroneous value so he can inspect and possibly correct those outputs.

In this paper, we describe the design of a fine-grained lineage tracking and querying system for array-oriented scientific workflows. We assume a data and execution model similar to SciDB [2]. We chose this because it provides a closed execution environment that can capture all of the lineage information, and because it is specifically designed for scientific data processing (scientists typically use RDBMSes to manage metadata and do data processing outside of the database). The system allows scientists to perform exploratory workflow debugging by executing a series of *data lineage queries* that walk backward to identify the specific cells in the input arrays on which a given output cell depends and that walk forward to find the output cells that a particular input cell influenced. Such a system must manage input to output relationships at a *fine-grained* array-cell level.

Prior work in data lineage tracking systems has largely been limited to coarse-grained metadata tracking [3], [4], which stores relationships at the file or relational table level. *Fine-grained lineage* tracks relationships at the array cell or tuple level. The typical approach, popularized by Trio [5], which we call *cell-level lineage*, eagerly materializes the identifiers of the input data records (e.g., tuples or array cells) that each output record depends on, and uses it to directly answer backward lineage queries. An alternative, which we call *black-box lineage*, simply records the input and output datasets and runtime parameters of each operator as it is executed, and materializes the lineage at lineage query time by re-running relevant operators in a tracing mode.

Unfortunately, both techniques are insufficient in scientific applications for two reasons. First, scientific applications make heavy use of user defined functions (UDFs), whose semantics are opaque to the lineage system. Existing approaches conservatively assume that every output cell of a UDF depends on every input cell, which limits the utility of a fine-grained lineage system because it tracks a large amount of information without providing any insight into which inputs actually contributed to a given output. This necessitates proper APIs so that UDF designers can expose fine-grained lineage information and operator semantics to the lineage system.

Second, neither black-box only nor cell-level only techniques are sufficient for many applications. Scientific workflows consume data arrays that regularly contain millions of cells, while generating complex relationships between groups of input and output cells. Storing cell-level lineage can avoid re-running some computationally intensive operators (e.g., an image processing operator that detects a small number of stars in telescope imagery), but needs enormous amounts of storage if every output depends on every input (e.g., a matrix sum operation) – it may be preferable to recompute the lineage at query time. In addition, applications such as LSST[1] are often subject to limitations that only allow them to dedicate

[1]http://lsst.org

a small percentage of storage to lineage operations. Ideally, lineage systems would support a hybrid of the two approaches and take user constraints into account when deciding which operators to store lineage for.

This paper seeks to address both challenges. We interviewed scientists from several domains to understand their data processing workflows and lineage needs and used the results to design a science-oriented data lineage system. We introduce *Region Lineage*, which exploits locality properties prevalent in the scientific operators we encountered. It addresses common relationships between regions of input and output cells by storing grouped or summary information rather than individual pairs of input and output cells. We developed a lineage API that supports black-box lineage as well as *Region Lineage*, which subsumes cell-level lineage. Programmers can also specify forward/backward *Mapping Functions* for an operator to directly compute the forward/backward lineage solely from input/output cell coordinates and operator arguments; we implemented these for many common matrix and statistical functions. We also developed a hybrid lineage storage system that allows users to explicitly trade-off storage space for lineage query performance using an optimization framework. Finally, we introduce two end-to-end scientific lineage benchmarks.

As mentioned earlier, the system prototype, SubZero, is implemented in the context of the SciDB model. SciDB stores multi-dimensional arrays and executes database queries composed of built-in and user-defined operators (UDFs) that are compiled into workflows. Given a set of user-specified storage constraints, SubZero uses an optimization framework to choose the optimal type of lineage (black box, or one of several new types we propose) for each SciDB operator that minimizes lineage query costs while respecting user storage constraints.

A summary of our contributions include:

1) The notion of *region lineage*, which SubZero uses to efficiently store and query lineage data from scientific applications. We also introduce several efficient representations and encoding schemes that each have different overhead and query performance trade offs.
2) A *lineage API* that operator developers can use to expose lineage from user defined operators, including the specification of *mapping functions* for many of the built in SciDB operators.
3) A *unified storage model* for mapping functions, region and cell-level lineage, and black-box lineage.
4) An *optimization framework* which picks an optimal mixture of black-box and region lineage to maximize query performance within user defined constraints.
5) A performance evaluation of our approach on end-to-end astronomy and genomics benchmarks. The astronomy benchmark, which is computationally intensive but exhibits high locality, benefits from efficient representations. Compared to cell-level and black-box lineage, SubZero reduces storage overhead by nearly 70× and speeds query performance by almost 255×. The genomics benchmark highlights the need for, and benefits of, using an optimizer to pick the storage layout, which improves query performance by 2–3× while staying within user constraints.

The next section describes our motivating use cases in more detail. It is followed by a high level system architecture and details of the rest of the system.

II. Use Cases

We developed two benchmark applications after discussions with environmental scientists, astronomists, and geneticists. The first is an image processing benchmark developed with scientists at the Large Synoptic Survey Telescope (LSST) project. It is very similar to environmental science requirements, so they are combined together. The second was developed with geneticists at the Broad Institute[2]. Each benchmark consists of a workflow description, a dataset, and lineage queries. We used the benchmarks to design the optimizations described in the paper. This section will briefly describe each benchmark's scientific application, the types of desired lineage queries, and application-specific insights.

A. Astronomy

The Large Synaptic Survey Telescope (LSST) is a wide angle telescope slated to begin operation in Fall 2015. A key challenge in processing telescope images is filtering out high energy particles (cosmic rays) that create abnormally bright pixels in the resulting image, which can be mistaken for stars. The telescope compensates by taking two consecutive pictures of the same piece of the sky and removing the cosmic rays in software. The LSST image processing workflow (Figure 1) takes two images as input and outputs an annotated image that labels each pixel with the celestial body it belongs to. It first cleans and detects cosmic rays in each image separately, then creates a single composite, cosmic-ray-free, image that is used to detect celestial bodies. There are 22 SciDB built-in operators (blue solid boxes) that perform common matrix operations, such as convolution, and four UDFs (red dotted boxes labeled A-D). The UDFs A and B output cosmic-ray masks for each of the images. After the images are subsequently merged, C removes cosmic-rays from the composite image, and D detects stars from the cleaned image.

The LSST scientists are interested in three types of queries. The first picks a star in the output image and traces the lineage back to the initial input image to detect bad input pixels. The latter two queries select a region of output (or input) pixels and trace the pixels backward (or forward) through a subset of the workflow to identify a single faulty operator. As an example, suppose the operator that computes the mean brightness of the image generated an anomalously high value due to a few bad pixel, which led to further mis-calculations. The astronomer might work backward from those calculations, identify the input pixels that contributed to them, and filter out those pixels that appear excessively bright.

Both the LSST and environmental scientists described workloads where the majority of the data processing code computes

[2]http://www.broadinstitute.org/

output pixels using input pixels within a small distance from the corresponding coordinate of the output pixel. These regions may be constant, pre-defined values, or easily computed from a small amount of additional metadata. For example, a pixel in the mask produced by cosmic ray detection (CRD) is set if the related input pixel is a cosmic ray, and depends on neighboring input cells within 3 pixels. Otherwise, it only depends on the related input pixel. They also felt that it is sufficient for lineage queries to return a superset of the exact lineage. Although we do not take advantage of this insight, this suggests future work in lossy compression techniques.

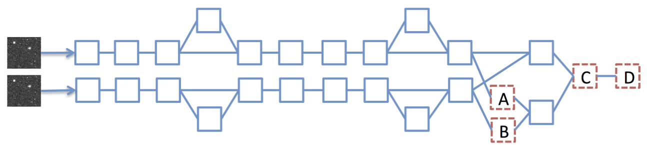


Fig. 1. Summary diagram of LSST workflow. Each solid rectangle is a SciDB native operator while the red dotted rectangles are UDFs.

B. Genomics Prediction

We have also been working with researchers at the Broad Institute on a genomics benchmark related to predicting recurrences of medulloblastoma in patients. Medulloblastoma is a form of cancer that spawns brain tumors that spread through the cerebrospinal fluid. Pablo et. al [6] have identified a set of patient features that help predict relapse in medulloblastoma patients that have been treated. The features include histology, gene expression levels, and the existence of genetic abnormalities. The workflow (Figure 2) is a two-step process that first takes a training patient-feature matrix and outputs a Bayesian model. Then it uses the model to predict relapse in a test patient-feature matrix. The model computes how much each feature value contributes to the likelihood of patient relapse. The ten built-in operators (solid blue boxes) are simple matrix transformations. The remaining UDFs extract a subset of the input arrays (E,G), compute the model (F), and predict the relapse probability (H).

The model is designed to be used by clinicians through a visualization that generates lineage queries. The first query picks a relapse prediction and traces its lineage back to the training matrix to find supporting input data. The second query picks a feature from the model and traces it back to the training matrix to find the contributing input values. The third query points at a set of training values and traces them forward to the model, while the last query traces them to the end of the workflow to find the predictions they affected.

The genomics benchmark can devote up-front storage and runtime overhead to ensure fast query execution because it is an interactive visualization. Although this is application specific, it suggests that scientific applications have a wide range of storage and runtime overhead constraints.

III. Architecture

SubZero records and stores lineage data at workflow runtime and uses it to efficiently execute lineage queries. The input to

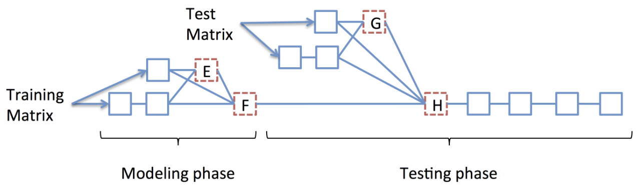


Fig. 2. Simplified diagram of genomics workflow. Each solid rectangle is a SciDB native operator while the red dotted rectangles are UDFs.

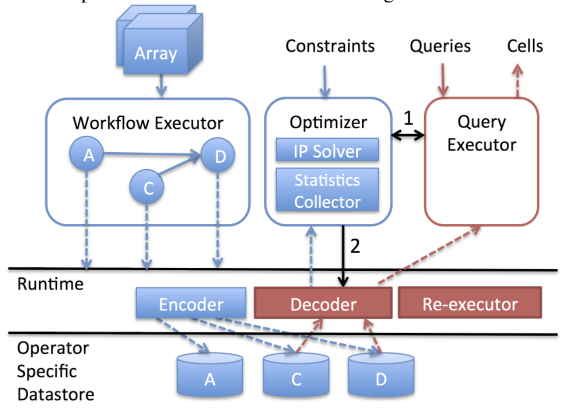


Fig. 3. The SubZero architecture.

SubZero is a workflow specification (the graph in *Workflow Executor*), constraints on the amount of storage that can be devoted to lineage tracking, and a sample lineage query workload that the user expects to run. SubZero optimally decides the type of lineage that each operator in the workflow will generate (the *lineage strategy*) in order to maximize the performance of the query workload performance.

Figure 3 shows the system architecture. The solid and dashed arrows indicate the control and data flow, respectively. Users interact with SubZero by defining and executing workflows (*Workflow Executor*), specifying constraints to the *Optimizer*, and running lineage queries (*Query Executor*). The operators in the workflow specify a list of the types of lineage (described in Section V) that each operator can generate, which defines the set of optimization possibilities.

Each operator initially generates black-box lineage (i.e., just records the names of the inputs it processes) but over time changes its strategy through optimization. As operators process data, they send lineage to the *Runtime*, which uses the *Encoder* to serialize the lineage before writing it to *Operator Specific Datastores*. The *Runtime* may also send lineage and other statistics to the *Optimizer*, which calculates statistics such as the amount of lineage that each operator generates. SubZero periodically runs the *Optimizer*, which uses an *Integer Programming Solver* to compute the new lineage strategy. On the right side, the *Query Executor* compiles lineage queries into query plans that join the query with lineage data. The *Executor* requests lineage from the *Runtime*, which reads and decodes stored lineage, uses the *Re-executor* to re-run the operators, and sends statistics (e.g., query fanout and fanin) to the optimizer to refine future optimizations.

Given this overview, we now describe the data model and structure of lineage queries (Section IV), the different types of lineage the system can record (Section V), the functionality of the *Runtime*, *Encoder*, and *Query Executor* (Section VI), and finally the optimizer in Section VII.

IV. Data, Lineage and Query Model

In this section, we describe the representation and notation of lineage data and queries in SubZero.

SubZero is designed to work with a workflow executor system that applies a fixed sequence of operators to some set of inputs. Each operator operates on one or more input objects (e.g., tables or arrays), and produces a single output object. Formally, we say an operator P takes as input n objects, $I_P^1, ..., I_P^n$, and outputs a single object, O_P.

Multiple operators are composed together to form a workflow, described by a workflow specification, which is a directed acyclic graph $W = (N, E)$, where N is the set of operators, and $e = (O^P, I_i^{P'}) \in E$ specifies that the output of P forms the *i'th* input to the operator P'. An instance of W, W_j, executes the workflow on a specific dataset. Each operator runs when all of its inputs are available.

The data follows the SciDB data model, which processes multi-dimensional arrays. A combination of values along each dimension, termed a *coordinate*, uniquely identifies a cell. Each cell in an array has the same schema, and consists of one or more named, typed fields. SciDB is "no overwrite," meaning that intermediate results produced as the output of an operator are always stored persistently, and each update to an object creates a new, persistent version. SubZero stores lineage information with each version to speed up lineage queries.

Our notion of backward lineage is defined as a subset of the inputs that will reproduce the same output value if the operator is re-run on its lineage. For example, the lineage of an output cell of Matrix Multiply are all cells of the corresponding row and column in the input arrays – even if some are empty. Forward lineage is defined as a subset, C, of the outputs such that the backward lineage of C contains the input cells. The exact semantics for UDFs are ulitmately controlled by the developer.

SubZero supports three types of lineage: *black box*, *cell-level*, and *region* lineage. As a workflow executes, lineage is generated on an operator-by-operator basis, depending on the types of lineage that each operator is instrumented to support and the materialization decisions made by the optimizer. We have instrumented SciDB's built-in operators to generate lineage mappings from inputs to outputs and provide an API for UDF designers to expose these relationships. If the API is not used, then SubZero assumes an all-to-all relationship between the cells of the input arrays and cells of the output array.

a) Black-box lineage: SubZero does not require additional resources to store black-box lineage because, like SciDB, our workflow executor records intermediate results as well as input and output array versions as peristent, named objects. These are sufficient to re-run any previously executed operator from any point in the workflow.

b) Cell-level lineage: Cell-level lineage models the relationships between an output cell and each input cell that generated it [3] as a set of pairs of input and output cells:

$$\{(out, in) | out \in O_P \wedge in \in \cup_{i \in [1,n]} I_P^i\}$$

Here, $out \in O_P$ means that out is a single cell contained in the output array O_P. in refers to a single cell in one of the input arrays.

c) Region lineage: Region lineage models lineage as a set of *region pairs*. Each region pair describes an all-to-all lineage relationship between a set of output cells, $outcells$, and a set of input cells, $incells_i$, in each input array, I_P^i:

$$\{(outcells, incells_1, ..., incells_n) | outcells \subseteq O_P \wedge incells_i \subseteq I_P^i\}$$

Region lineage is more than a short hand; scientific applications often exhibit locality and generate multiple output cells from the same set of input cells, which can be represented by a single region pair. For example, the LSST star detection operator finds clusters of adjacent bright pixels and generates an array that labels each pixel with the star that it belongs to. Every output pixel labeled *Star X* depends on all of the input pixels in the *Star X* region. Automatically tracking such relationships at the cell level is particularly expensive, so region lineage is a generalization of cell-level lineage that makes this relationship explicit. For this reason, later sections will exclusively discuss region pairs.

Users execute a lineage query by specifying the coordinates of an initial set of query cells, C, in a starting array, and a path of operators $(P_1 \ldots P_m)$ to trace through the workflow:

$$R = execute\_query(C, ((P_1, idx_1), ..., (P_m, idx_m)))$$

Here, the indexes $(idx_1 \ldots idx_m)$ are used to disambiguate which input of a multi-input operator that the query path traverses.

Depending on the order of operators in the query path, SubZero recognizes the query as a *forward lineage query* or *backward lineage query*. A *forward lineage query* defines a path from some ancestor operator P_1 to some descendent operator P_m. The output of an operator P_{i-1} is the idx_i'th input of the next operator, P_i. The query cells C are a subset of P_1's idx_1'th input array, $C \subseteq I_{P_1}^{idx_1}$.

A *backward lineage query* reverses this process, defining a path from some descendent operator, P_1 that terminates at some ancestor operator, P_m. The output of an operator, P_{i+1} is the idx_i'th input of the previous operator, P_i, and the query cells C are a subset of P_1's output array, $C \subseteq O_{P_1}$. The query results are the coordinates of the cells $R \subseteq O_{P_m}$ or $R \subseteq I_{P_m}^{idx_m}$, for forward and backward queries, respectively.

V. Lineage API and Storage Model

SubZero allows developers to write operators that efficiently represent and store lineage. This section describes several modes of region lineage, and an API that UDF developers can use to generate lineage from within the operators. We also introduce a mechanism to control the modes of lineage

[3] Although we model and refer to lineage as a mapping between input and output cells, in the SubZero implementation we store these mappings as references to physical cell coordinates.

TABLE I
RUNTIME AND OPERATOR METHODS

API Method	Description
System API Calls	
lwrite(outcells, $\mathrm{incells_1}$, ...,$\mathrm{incells_n}$)	API to store lineage relationship.
lwrite(outcells, payload)	API to store small binary payload instead of input cells. Called by payload operators.
Operator Methods	
run(input-1,...,input-n,cur_modes)	Execute the operator, generating lineage types in cur_modes $\subseteq \{Full, Map, Pay, Comp, Blackbox\}$
$\mathrm{map_b}$(outcell, i)	Computes the input cells in $\mathrm{input_i}$ that contribute to outcell.
$\mathrm{map_f}$(incell, i)	Computes the output cells that depend on incell $\in \mathrm{input_i}$.
$\mathrm{map_p}$(outcell, payload, i)	Computes the input cells in $\mathrm{input_i}$ that contribute to outcell, has access to payload.
supported_modes()	Returns the lineage modes $C \subseteq \{Full, Map, Pay, Comp, Blackbox\}$ that the operator can generate.

that an operator generates. Finally, we describe how SubZero re-executes black-box operators during a lineage query. Table I summarizes the API calls and operator methods that are introduced in this section.

Before describing the different lineage storage methods, we illustrate the basic structure of an operator:

```
class OpName:
   def run(input-1,...,input-n,cur_modes):
      /* Process the inputs, emit the output */
      /* Record lineage modes specified
         in cur_modes */
    def supported_modes():
      /* Return the lineage modes the
         operator supports */
```

Each operator implements a *run()* method, which is called when inputs are available to be processed. It is passed a list of lineage modes it should output in the *cur_modes* argument; it writes out lineage data using the *lwrite()* method described below. The developer specifies the modes that the operator supports (and that the runtime will consider) by overriding the *supported_modes()* method. If the developer does not override *supported_modes()*, SubZero assumes an all-to-all relationship between the inputs and outputs. Otherwise, the operator automatically supports black-box lineage.

For ease of explanation, this section is described in the context of the LSST operator CRD (cosmic ray detection, depicted as A and B in Figure 1) that finds pixels containing cosmic rays in a single image, and outputs an array of the same size. If a pixel contains a cosmic ray, the corresponding cell in the output is set to 1, and the output cell depends on the 49 neighboring pixels within a 3 pixel radius. Otherwise the output cell is set to 0, and only depends on the corresponding input pixel. A region pair is denoted ($outcells$, $incells$).

A. Lineage Modes

SubZero supports four modes of region lineage (*Full, Map, Pay, Comp*), and one mode of black-box lineage (*Blackbox*). *cur_modes* is set to *Blackbox* when the operator does not need to generate any pairs (because black box lineage is always in use). *Full* lineage explicitly stores all region pairs, and the other lineage modes reduce the amount of lineage that is stored by partially computing lineage at query time using developer defined mapping functions. The following sections describe the modes in more detail.

1) Full Lineage: Full lineage (*Full*) explicitly represents and stores all region pairs. It is straightforward to instrument any operator to generate full lineage. The developer simply writes code that generates region pairs and uses $lwrite()$ to store the pairs. For example, in the following CRD pseudocode, if cur_modes contains *Full*, the code iterates through each cell in the output, calculates the lineage, and calls $lwrite()$ with lists of cell coordinates. Note that if *Full* is not specified, the operator can avoid running the lineage related code.

```
def run(image, cur_modes):
   ...
   if Full ∈ cur_modes:
     for each cell in output:
       if cell == 1:
         neighs = get_neighbor_coords(cell)
         lwrite([cell.coord], neighs)
      else:
         lwrite([cell.coord], [cell.coord])
```

Although this lineage mode accurately records the lineage data, it is potentially very expensive to both generate and store. We have identified several widely applicable operator properties that allow the operators to generate more efficient modes of lineage, which we describe next.

2) Mapping Lineage: Mapping lineage (*Map*) compactly represents an operator's lineage using a pair of mapping functions. Many operators such as matrix transpose exhibit a fixed execution structure that does not depend on the input cell values. These operators, called *mapping operators*, can compute forward and backward lineage from a cell's coordinates and metadata (e.g., input and output array sizes) and do not need to access array data values. This is a valuable property because mapping operators do not incur runtime and storage overhead. For example, one-to-one operators, such as matrix addition, are mapping operators because an output cell only depends on the input cell at the same coordinate, regardless of the value. Developers implement a pair of mapping functions, $map_f(cell, i)/map_b(cell, i)$, that calculate the forward/backward lineage of an input/output cell's coordinates, with respect to the i'th input array. For example, a 2D transpose operator would implement the following functions:

```
def map_b((x,y), i):   def map_f((x,y), i):
   return [(y,x)]         return [(y,x)]
```

Most SciDB operators (e.g., matrix multiply, join, transpose, convolution) are mapping operators, and we have implemented their forward and backward mapping functions. Mapping operators in the astronomy and genomics benchmarks are depicted as solid boxes (Figures 1 and 2).

3) Payload Lineage: Rather than storing the input cells in each region pair, payload lineage (*Pay*) stores a small amount of data (*a payload*), and recomputes the lineage using a payload-aware mapping function ($map_p()$). Unlike mapping lineage, the mapping function has access to the user-stored binary payload. This mode is particularly useful when the operator has high fanin and the payload is very small. For example, suppose that the radius of neighboring pixels that a cosmic ray pixel depends on increases with brightness, then payload lineage only stores the brightness insteall of the input cell coordinates. (*Payload operators*) call $lwrite(outcells, payload)$ to pass in a list of output cell coordinates and a binary blob, and define a *payload function*, $map_p(outcell, payload, i)$, that directly computes the backward lineage of $outcell \in outcells$ from the $outcell$ coordinate and the payload. The result are input cells in the i'th input array. As with mapping functions, payload lineage does not need to access array data values. The following pseudocode stores radius values instead of input cells:

```
def run(image,cur_modes):
   ...
   if $PAY$ ∈ cur_modes:
     for each cell in output:
       if cell == 1:
         lwrite([cell.coord], '3')
       else:
         lwrite([cell.coord], '0')

def map_p((x,y), payload, i):
   return get_neighbors((x,y), int(payload))
```

In the above implementation, each region pair stores the output cells and an additional argument that represents the radius, as opposed to the neighboring input cells. When a backward lineage query is executed, SubZero retrieves the (outcells, payload) pairs that intersect with the query and executes map_p on each pair. This approach is particularly powerful because the payload can store arbitrary data – anything from array data values to lineage predicates [7]. Operators D to G in the two benchmarks (Figures 1 and 2) are payload operators.

Note that payload functions are designed to optimize execution of backward lineage queries. While SubZero can index the input cells in full lineage, the payload is a binary blob that cannot be easily indexed. A forward query must iterate through each (outcells, payload) pair and compute the input cells using map_p before it can be compared to the query coordinates.

4) Composite Lineage: Composite lineage (*Comp*) combines mapping and payload lineage. The mapping function defines the default relationship between input and output cells, and results of the payload function *overwrite* the default lineage if specified. For example, CRD can represent the default relationship – each output cell depends on the corresponding input cell in the same coordinate – using a mapping function, and write payload lineage for the cosmic ray pixels:

```
def run(image,cur_modes):
  ...
  if $COMP$ ∈ cur_modes:
     for each cell in output:
       if cell == 1:
        lwrite([cell.coord], 3)
    // else map_b defines default behavior

def map_p((x,y), radius, i):
   return get_neighbors((x,y), radius)

def map_b((x,y), i):
   return [(x,y)]
```

Composite operators can avoid storing lineage for a significant fraction of the output cells. Although it is similar to payload lineage in that the payload cannot be indexed to optimize forward queries, the amount of payload lineage that is stored may be small enough that iterating through the small number of (outcells, payload) pairs is efficient. Operators A,B and C in the astronomy benchmark (Figure 1) are composite operators.

B. Supporting Operator Re-execution

An operator stores black-box lineage when cur_modes equals $Blackbox$. When SubZero executes a lineage query on an operator that stored black-box lineage, the operator is re-executed in tracing mode. When the operator is re-run at lineage query time, SubZero passes $cur\_modes = Full$, which causes the operator to perform $lwrite()$ calls. The arguments to these calls are sent to the query executor.

Rather than re-executing the operator on the full input arrays, SubZero could also reduce the size of the inputs by applying bounding box predicates prior to re-execution. The predicates would reduce both the amount of lineage that needs to be stored and the amount of data that the operator needs to re-process. Although we extended both mapping and full operators to compute and store bounding box predicates, we did not find it to be a widely useful optimization. During query execution, SubZero must retrieve the bounding boxes for every query cell, and either re-execute the operator for each box, or merge the bounding boxes and re-run the operator using the merged predicate. Unfortunately, the former approach incurs an overhead on each execution (to read the input arrays and apply the predicates) that quickly becomes a significant cost. In the latter approach, the merged bounding box quickly expands to encompass the full input array, which is equivalent to completely re-executing the operator, but incurs the additional cost to retrieve the predicates. For these reasons, we do not further consider them here.

VI. Implementation

This section describes the *Runtime*, *Encoder*, and *Query Executor* components in greater detail.

A. Runtime

In SciDB (and our prototype), we automatically store black-box lineage by using write-ahead logging, which guarantees that black-box lineage is written before the array data, and is "no overwrite" on updates. Region lineage is stored in a collection of BerkeleyDB hashtable instances. We use BerkeleyDB to store region lineage to avoid the client-server communication overhead of interacting with traditional DBMSes. We turn off fsync, logging and concurrency control to avoid recovery and locking overhead. This is safe because the region lineage is treated as a cache, and can always be recovered by re-running operators.

The runtime allocates a new BerkeleyDB database for each operator instance that stores region lineage. Blocks of region pairs are buffered in memory, and bulk encoded using the *Encoder*. The data in each region pair is stored as a unit (SubZero does not optimize across region pairs), and the output and input cells use separate encoding schemes. The layout can be optimized for backward or forward queries by respectively storing the output or input cells as the hash key. On a key collision, the runtime decodes, merges, and re-encodes the two hash values. The next subsection describes how the *Encoder* serializes the region pairs.

B. Encoder

While Section V presented efficient ways to represent region lineage, SubZero still needs to store cell coordinates, which can easily be larger than the original data arrays. The *Encoder* stores the input and output cells of a region pair (generated by calls to $lwrite()$) into one or more hash table entries, specified by an *encoding strategy*. We say the encoding strategy is *backward optimized* if the output cells are stored in the hash key, and *forward optimized* if the hash key contains input cells.

We found that four basic strategies work well for the operators we encountered. – $FullOne$ and $FullMany$ are the two strategies to encode full lineage, and $PayOne$ and $PayMany$ encode payload lineage[4].

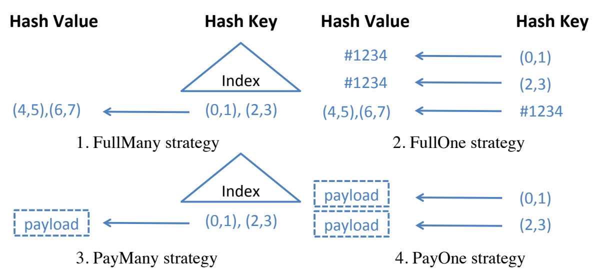


Fig. 4. Four examples of encoding strategies

Figure 4 depicts how the backward-optimied implementation of these strategies encode two output cells with coordinates $(0, 1)$ and $(2, 3)$ that depend on input cells with coordinates $(4, 5)$ and $(6, 7)$. $FullMany$ uses a single hash entry with the set of serialized output cells as the key and the set of input cells as the value (Figure 4.1). Each coordinate is bitpacked into a single integer if the array is small enough. We also create an R Tree on the cells in the hash key to quickly find the entries that intersect with the query. This index uses the dimensions of the array as its keys and identifies the hash table entries that contain cells in particular regions. The figure shows the unserialized versions of the cells for simplicity. $FullMany$ is most appropriate when the lineage has high fanout because it only needs to store the output cells once.

If the fanout is low, $FullOne$ more efficiently serializes and stores each output cell as the hash key of a separate hash entry. The hash value stores a reference to a single entry containing the input cells (Figure 4.2). This implementation doesn't need to compute and store bounding box information and doesn't need the spatial index because each input cell is stored separately, so queries execute using direct hash lookups.

For payload lineage, $PayMany$ stores the lineage in a similar manner as $FullMany$, but stores the payload as the hash value (Figure 4.3). $PayOne$ creates a hash entry for every output cell and stores a duplicate of the payload in each hash value (Figure 4.4).

The *Optimizer* picks a lineage strategy that spans the entire workflow. It picks one or more *storage strategies* for each operator. Each storage strategy is fully specified by a lineage mode (Full, Map, Payload, Composite, or Black-box), encoding strategy, and whether it is forward or backward optimized ($\rightarrow$ or $\leftarrow$). SubZero can use multiple storage strategies to optimize for different query types.

C. Query Execution

The *Query Executor* iteratively executes each step in the lineage query path by joining the lineage with the coordinates of the query cells, or the intermediate cells generated from the previous step. The output at each step is a set of cell coordinates that is compactly stored in an in-memory boolean array with the same dimensions as the input (backward query) or output (forward query) array. A bit is set if the intermediate result contains the corresponding cell. For example, suppose we have an operator P that takes as input a 1×4 array. Consider a backwards query asking for the lineage of some output cell C of P. If the result of the query is 1001, this means that C depends on the first and fourth cell in P's input.

We chose the in-memory array because many operators have large fanin or fanout, and can easily generate several times more results (due to duplicates) than are unique. Deduplication avoids wasting storage and saves work. Similarly, the executor can close an operator early if it detects that all of the possible cells have been generated.

We also implement an *entire array optimization* to speed up queries where all of the bits in the boolean array are set. For example, this can happen if a backward query traverses several high-fanin operators or an all-to-all operator such as matrix inversion. In these cases, calculating the lineage of every query cell is very expensive and often unnecessary. Many operators (e.g., matrix multiply or inverse) can safely assume that the forward (backward) lineage of an entire input (output) array is the entire output (input) array. This optimization is valuable when it can be applied – it improved the query performance of a forward query in the astronomy benchmark that traverses an all-to-all-operator by 83×.

In general, it is difficult to automatically identify when the optimization's assumptions hold. Consider a concatenate operator that takes two 2D arrays A, B with shapes (1, n) and (1, m), and produces an (1, n+m) output by concatenating B to A. The optimization would produce different results, because A's forward lineage is only a subset of the output. We currently

[4]We tried a large number of possible strategies and found that complex encodings (e.g., compute and store the bounding box of a set of cells, C, along with cells in the bounding box but not in C) incur high encoding costs without noticeably reduced storage costs. Many are also readily implemented as payload or composite lineage

rely on the programmer to manually annotate operators where the optimization can be applied.

VII. Lineage Strategy Optimizer

Having described the basic storage strategies implemented in SubZero, we now describe our lineage storage optimizer. The optimizer's objective is to choose a set of *storage strategies* that minimize the cost of executing the workflow while keeping storage overhead within user-defined constraints. We formulate the task as an integer programming problem, where the inputs are a list of operators, strategy pairs, disk overheads, query cost estimates, and a sample workload that is used to derive the frequency with which each operator is invoked in the lineage workload. Additionally, users can manually specify operator specific strategies prior to running the optimizer.

The formal problem description is stated as:

$$\begin{array}{lll}
\min_x & \sum_i p_i * \left(\min_{j|x_{ij}=1} q_{ij}\right) + & \epsilon * \sum_{ij}(disk_{ij} + \beta * run_{ij}) * x_{ij} \\
\text{s.t.} & \sum_{ij} disk_{ij} * x_{ij} & \leq MaxDISK \\
 & \sum_{ij} run_{ij} * x_{ij} & \leq MaxRUNTIME \\
 & \forall_i \left(\sum_{0\leq j<M} x_{ij}\right) & \geq 1 \\
 & \forall_{i,j} x_{ij} & \in \{0,1\} \\
 & & \\
\multicolumn{2}{l}{\text{user specified strategies}} & \\
 & x_{ij} = 1 & \forall_{i,j} x_{ij} \in U
\end{array}$$

Here, $x_{ij} = 1$ if operator i stores lineage using strategy j, and 0 otherwise. $MaxDISK$ is the maximum storage overhead specified by the user; q_{ij}, run_{ij}, and $disk_{ij}$, are the average query cost, runtime overhead, and storage overhead costs for operator i using strategy j as computed by the cost model. p_{ij} is the probability that a lineage query in the workload accesses operator i, and is computed from the sample workload. A single operator may store its lineage data using multiple strategies.

The goal of the objective function is to minimize the cost of executing the lineage workload, preferring strategies that use less storage. When an operator uses multiple strategies to store its lineage, the query processor picks the strategy that minimizes the query cost. The $\min$ statement in the left hand term picks the best query performance from the strategies that have been picked ($j|x_{ij} = 1$). The right hand term penalizes strategies that take excessive disk space or cause runtime slowdown. β weights runtime against disk overhead, and ϵ is set to a very small value to break ties. A large ϵ is similar to reducing $MaxDISK$ or $MaxRUNTIME$.

We heuristically remove configurations that are clearly non-optimal, such as strategies that exceed user constraints, or are not properly indexed for any of the queries in the workload (e.g., forward optimized when the workload only contains backward queries). The optimizer also picks mapping functions over all other classes of lineage.

We solve the ILP problem using the simplex method in GNU Linear Programming Kit. The solver takes about 1ms to solve the problem for the benchmarks.

TABLE II
Lineage Strategies for experiments.

Strategy	**Description**
	Astronomy Benchmark
BlackBox	All operators store black-box lineage
BlackBoxOpt	Like BlackBox, uses mapping lineage for built-in-operators.
FullOne	Like BlackBoxOpt, but uses FullOne for UDFs.
FullMany	Like FullOne, but uses FullMany for UDFs.
Subzero	Like FullOne, but stores composite lineage using PayOne for UDFs.
	Genomics Benchmark
BlackBox	UDFs store black-box lineage
FullOne	UDFs store backward optimized FullOne
FullMany	UDFs store backward optimized FullMany
FullForw	UDFs store forward optimized FullOne
FullBoth	UDFs store FullForw and FullOne
PayOne	UDFs store PayOne
PayMany	UDFs store PayMany
PayBoth	UDFs store PayOne and FullForw

A. *Query-time Optimizer*

While the lineage strategy optimizer picks the optimal lineage strategy, the executor must still pick between accessing the lineage stored by one of the lineage strategies, or re-running the operator. The query-time optimizer consults the cost model using statistics gathered during query execution and the size of the query result so far to pick the best execution method. In addition, the optimizer monitors the time to access the materialized lineage. If it exceeds the cost of re-executing the operator, SubZero dynamically switches to re-running the operator. This bounds the worst case performance to 2× the black-box approach.

VIII. Experiments

In the following subsections, we first describe how SubZero optimizes the storage strategies for the real-world benchmarks described in Section II, then compare several of our lineage storage techniques with black-box level only techniques. The astronomy benchmark shows how our region lineage techniques improve over cell-level and black-box strategies on an image processing workflow. The genomics benchmark illustrates the complexity in determining an optimal lineage strategy and that the the optimizer is able to choose an effective strategy within user constraints.

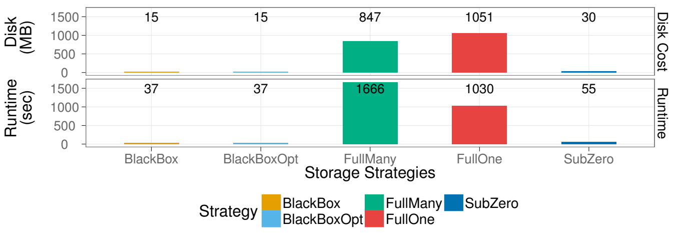
Overall, our findings are that:

- An optimal strategy heavily relies on operator properties such as fanin, and fanout, the specific lineage queries, and query execution-time optimizations. The difference between a sub-optimal and optimal strategy can be so large that an optimizer-based approach is crucial.
- Payload, composite, and mapping lineage are extremely effective and low overhead techniques that greatly improve query performance, and are applicable across a number of scientific domains.
- SubZero can improve the LSST benchmark queries by up to 10× compared to naively storing the region lineage (similar to what cell-level approaches would do) and up to 255× faster than black-box lineage. The runtime and storage overhead of the optimal scheme is up to 30 and

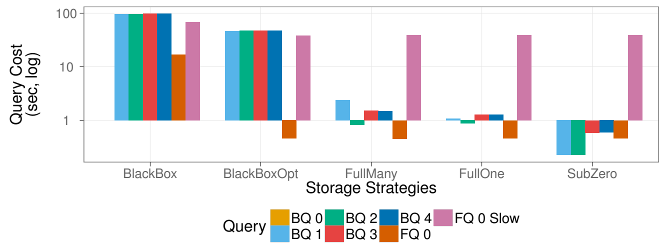
70× lower than cell-level lineage, respectively, and only 1.49 and 1.95× higher than executing the workflow.
- Even though the genomics benchmark executes operators very quickly, SubZero can find the optimal mix of black-box and region lineage that scales to the amount of available storage. SubZero uses a black-box only strategy when the available storage is small, and switches from space-efficient to query-optimized encodings with looser constraints. When the storage constraints are unbounded, SubZero improves forward queries by over 500× and backward queries by 2-3×.

The current prototype is written in Python and uses BerkeleyDB for the persistent store, and libspatialindex for the spatial index. The microbenchmarks are run on a 2.3 GHz linux server with 24 GB of RAM, running Ubuntu 2.6.38-13-server. The benchmarks are run on a 2.3 GHz MacBook Pro with 8 GB of RAM, a 5400 RPM hard disk, running OS X 10.7.2.

A. *Astronomy Benchmark*



(a) Disk and runtime overhead



(b) Query costs. Y-axes are log scale

Fig. 5. Astronomy Benchmark

In this experiment, we run the Astronomy workflow with five backward queries and one forward query as described in Section II-A. The 22 built-in operators are all expressed as mapping operators and the UDFs consist of one payload operator that detects celestial bodies and three composite operators that detect and remove cosmic rays. This workflow exhibits considerable locality (stars only depend on neighboring pixels), sparsity (stars are rare and small), and the queries are primarily backward queries. Each workflow execution consumes two 512×2000 pixel (8MB) images (provided by LSST) as input, and we compare the strategies in Table VIII.

Figure 5(a) plots the disk and runtime overhead for each of the strategies. $BlackBox$ and $BlackBoxOpt$ show the base cost to execute the workflow and the size of the input arrays – the goal is to be as close to these bars as possible. $FullOne$ and $FullMany$ both require considerable storage space (66×, 53×) because the three cosmic ray operators generate a region pair for every input and output pixel at the same coordinates. Similarly, both approaches incur 6× and 44× runtime overhead to serialize and store them. $FullMany$ must also construct the spatial index on the output cells. The SubZero optimizer instead picks composite lineage that only stores payload lineage for the small number of cosmic rays and stars. This reduces the runtime and disk overheads to 1.49× and 1.95× the workflow inputs. By comparison, this storage overhead is negligible compared to the cost of storing the intermediate and final results (which amount to 11.5× the input size).

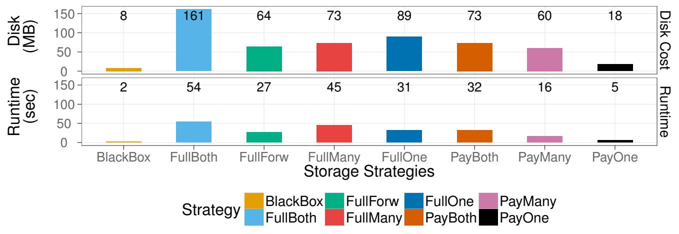
Figure 5(b) compares lineage query execution costs. $BQ\ x$ and $FQ\ x$ respectively stand for backward and forward query x. All of the queries use the entire array optimization described in Section VI-C whereas $FQ0Slow$ does not. $BlackBox$ must re-run each operator and takes up to 100 secs per query. $BlackBoxOpt$ can avoid rerunning the mapping operators, but still re-runs the computationally intensive UDFs. Storing region lineage reduces the cost of executing the backward queries by 34× ($FullMany$) and 45× ($FullOne$) on average. SubZero benefits from executing mapping functions and reading a small amount of lineage data and executes 255× faster on average. $FQ\ 0\ Slow$ illustrates how the all-to-all optimization improves the query performance by 83× by avoiding fine-grained lineage all-together.

B. *Genomics Benchmark*

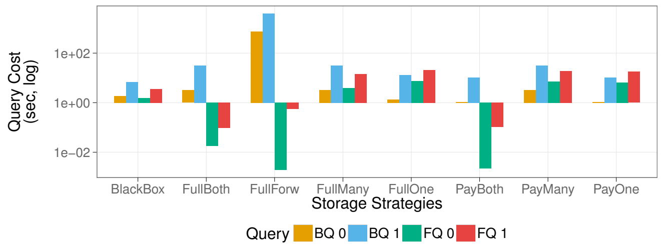
In this experiment, we run the genomics workflow and execute a lineage workload with an equal mix of forward and backward lineage queries (Section II-B). There are 10 built-in mapping operators, and the 4 UDFs are all payload operators. In contrast to the astronomy workflow, these UDFs do not exhibit significant locality, and perform data shuffling and extraction operations that are not amenable to mapping functions. In addition, the operators perform simple calculations, and execute quickly, so there is a less pronounced trade off between re-executing the workflow and accessing region lineage. In fact, there are cases where storing lineage actually *degrades* the query performance. We were provided a 56×100 matrix of 96 patients and 55 health and genetic features. Although the dataset is small, future datasets are expected to come from a larger group of patients, so we constructed larger datasets by replicating the patient data. The query performance and overheads scaled linearly with the size of the dataset and so we report results for the dataset scaled by 100×.

We first show the high variability between different static strategies (Table VIII) and how the query-time optimizer (Section VII-A) avoids sub-optimal query execution. We then show how the SubZero cost based optimizer can identify the optimal strategy within varying user constraints.

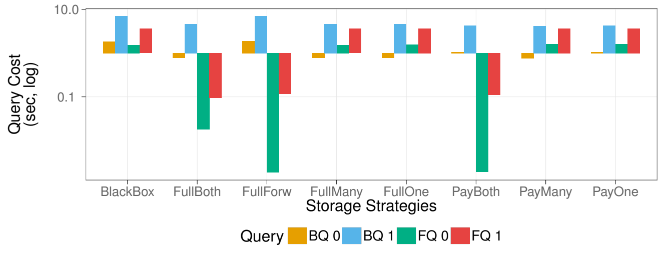
1) Query-Time Optimizer: This experiment compares the strategies in Table VIII with and without the query-time optimization described in Section VII-A. Each operator uses



(a) Disk and runtime overhead



(b) Query costs (static). Y-axes are log scale.



(c) Query costs (dynamic). Y-axes are log scale.

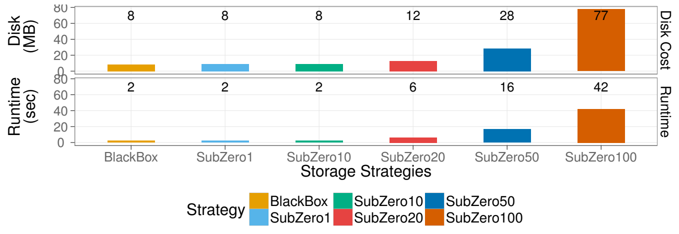
Fig. 6. Genomics benchmark. Queries run with (dynamic) and without (static) the query-time optimizer described in Section VII-A.

mapping lineage if possible, and otherwise stores lineage using the specified strategy. The majority of the UDFs generate region pairs that contain a single output cell. As mentioned in previous experiments, payload lineage stores very little binary data, and incurs less overhead than the full lineage approaches (Figure 6(a)). Storing both forward and backward-optimized lineage ($PayBoth$ and $FullBoth$) requires significantly more overhead – 8 and 18.5× more space than the input arrays, and 2.8 and 26× runtime slowdown.

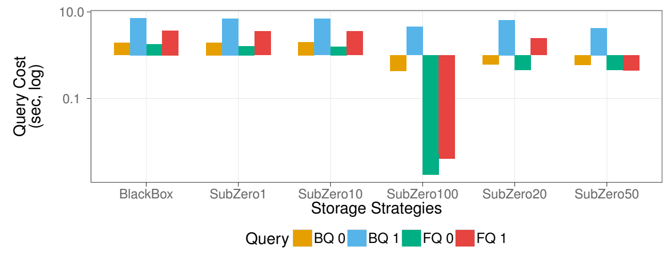
Figure 6(b) highlights how query performance can *degrade* if the executor blindly joins queries with mismatched indexed lineage (e.g., backward-optimized lineage with forward queries)[5]. For example, $FullForw$ degraded backward query performance by 520×. Interestingly, the BQ1 ran slower because the query path contains several operators with very large fanins. This generates so many intermediate results that performing index lookups on each one is slower than re-running the operators. Note however, that the forward optimized strategies improved the performance of FQ0 and FQ2 because the fanout is low.

Figure 6(c) shows that the query-time optimizer executes the queries as fast as, or faster than, $BlackBox$. In general, this requires accurate statistics and cost estimation, the optimizer limits the query performance degradation to 2× by

[5]All comparisons are relative to $BlackBox$



(a) Disk and runtime overhead



(b) Query costs. Y-axes are log scale.

Fig. 7. Genomics benchmark. SubZero*X* has storage constraint *X* MB

dynamically switching to the $BlackBox$ strategy. Overall, the backward and forward queries improved by up to 2 and 25×, respectively.

2) Lineage Strategy Optimizer: The previous section compared many strategies, each with different performance characteristics depending on the operator and query. We now evaluate the SubZero strategy optimizer on the genomics benchmark. Figure 7 illustrates that when the user increases storage constraints from 1 to 100MB (with unbounded runtime constraint), the optimizer picks more storage intensive strategies that are predicted to improve the benchmark queries. SubZero chooses $BlackBox$ when the constraint is too small, and stores forward and backward-optimized lineage that benefits all of the queries when the minimum amount of storage is available (20MB). Materializing further lineage has diminishing storage-to-query benefits. $SubZero100$ uses 50MB to forward-optimize the UDFs using $(MANY, ONE)$, which reduces the forward query costs to sub-second costs. This is because the UDFs have low fanout, so each join in the query path is a small number of hash lookups. Due to space constraints, we simply mention that specifying and varying the runtime overhead constraints achieves similar results.

C. Microbenchmark

The previous experiments compared several end-to-end strategies, however it can be difficult to distinguish the sources of the benefits. This subsections summarizes the key differences between the prevailing strategies in terms of overhead and query performance. The comparisons use an operator that generates synthetic lineage data with tunable parameters. Due to space constraints we show results from varying the fanin, fanout and payload size (for payload lineage).

Each experiment processes and outputs a 1000x1000 array, and generates lineage for 10% of the output cells. The results scaled close to linearly as the number of output cells with lineage varies. A region pair is randomly generated by selecting a cluster of output cells with a radius defined by $fanout$, and selecting $fanin$ cells in the same area from the input array. We generate region pairs until the total number of output cells is equal to 10% of the output array. The payload strategy uses a payload size of *fanin×4 bytes* (the payload is expected to be very small). We compare several backward optimized strategies ($\leftarrow FullMany$, $\leftarrow FullOne$, $\leftarrow PayMany$, $\leftarrow PayOne$), one forward lineage strategy ($\rightarrow FullOne$), and black-box ($BlackBox$). We first discuss the overhead to store and index the lineage, then comment on the query costs.

Figure 8 compares the runtime and disk overhead of the different strategies. For referenc, the size of the input array is 3.8MB. The best full lineage strategy differs based on the operator fanout. $FullOne$ is superior when $fanout \leq 5$ because it doesn't need to create and store the spatial index. The crossover point to $FullMany$ occurs when the cost of duplicating hash entries for each output cell in a region pair exceeds that of the spatial index. The overhead of both approaches increases with fanin. In contrast, payload lineage has a much lower overhead than the full lineage approaches and is independent of the fanin because the payload is typically small and does not need to be encoded. When the fanout increases to 50 or 100, $PayMany$ and $FullMany$ require less than 3MB and 1 second of overhead. The forward optimized $FullOne$ is comparable to the other approaches when the fanin is low. However, when the fanin increases it can require up to $fanin\times$ more hash entries because it creates an entry for every distinct input cell in the lineage. It converges to the backward optimized $FullOne$ when the fanout and fanin are high. Finally, $BlackBox$ has nearly no overhead.

Figure 9 shows that the query performance for queries that access the backward/forward lineage of 1000 output/input cells. The performance scales mostly linearly with the query size. There is a clear difference between $FullMany$ or $PayMany$, and $FullOne$ or $PayOne$, due to the additional cost of accessing the spatial index (Figure 9). Payload lineage performs similar to, but not significantly faster than full provenance, although the query performance remains constant as the fanin increases. In comparison (not shown), $BlackBox$ takes between 2 to 20 seconds to execute a query where *fanin=1* and around 0.7 seconds when *fanin=100*. Using a mis-matched index (e.g, using forward-optimized lineage for backward queries) takes up to two orders of magnitude longer than $BlackBox$ to execute the same queries. The forward queries using $\rightarrow FullOne$ execute similarly to $\leftarrow FullOne$ in Figure 9 so we do not include the plots.

D. Discussion

The experiments show that the best strategy is tied to the operator's lineage properties, and that there are orders of magnitude differences between different lineage strategies. Science-oriented lineage systems should seek to identify and exploit operator fanin, fanout, and redundancy properties.

Many scientific applications – particularly sensor-based or image processing applications like environmental monitoring

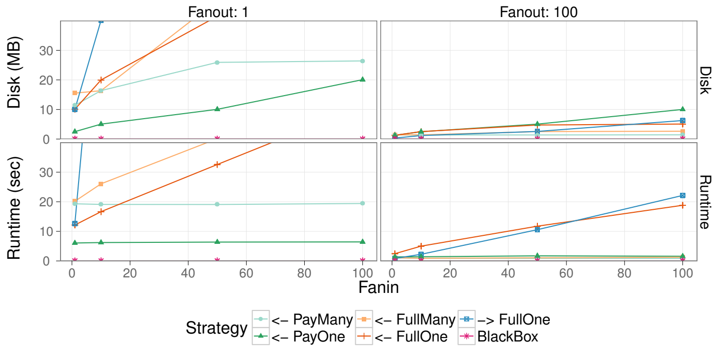


Fig. 8. Disk and runtime overhead

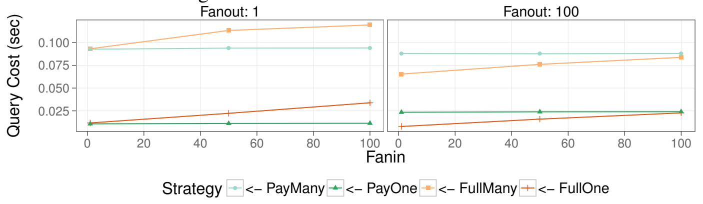


Fig. 9. Backward Lineage Queries, only backward-optimized strategies

or astronomy – exhibit substantial locality (e.g., average temperature readings within an area) that can be used to define payload, mapping or composite operators. As the experiments show, SubZero can record their lineage with less overhead than from operators that only support full lineage. When locality is not present, as in the genomics benchmark, the optimizer may still be able to find opportunities to record lineage if the constraints are relaxed. A very promising alternative is to simplify the process of writing payload and mapping functions by supporting variable granularities of lineage. This lets developers define coarser relationships between input and outputs (e.g., specify lineage as a bounding box that may contain inputs that didn't contribute to the output). This also allows the lineage system perform lossy compression.

IX. Related Work

There is a long history of provenance and lineage research both in database systems and in more general workflow systems. There are several excellent surveys that characterize provenance in databases [8] and scientific workflows [9], [10]. As noted in the introduction, the primary differences from prior work are that SubZero uses a mix of black-box and region provenance, exploits the semantics of scientific operators (making using of mapping functions) and uses a number of provenance encodings.

Most workflow systems support custom operators containing user-designed code that is opaque to the runtime. This presents a difficulty when trying to manage cell-level (e.g., array cells or database tuples) provenance. Some systems [4], [11] model operators as black-boxes where all outputs depend on all inputs, and track the dependencies between input and output datasets. Efficient methods to expose, store and query cell-level provenance is an area of on-going research.

Several projects exploit workflow systems that use high level programming constructs with well defined semantics.

RAMP [12] extends MapReduce to automatically generate lineage capturing wrappers around Map and Reduce operators. Similarly, Amsterdamer et al [13] instrument the PIG [14] framework to track the lineage of PIG operators. However, user defined operators are treated as black-boxes, which limits their ability to track lineage.

Other workflow systems (e.g., Taverna [3] and Kepler [15]), process nested collections of data, where data items may be imagees or DNA sequences. Operators process data items in a collection, and these systems automatically track which subsets of the collections were modified, added, or removed [16], [17]. Chapman et. al [18] attach to each data item a provenance tree of the transformations resulting in the data item, and propose efficient compression methods to reduce the tree size. However, these systems model operators as black-boxes and data items are typically files, not records.

Database systems execute queries that process structured tuples using well defined relational operators, and are a natural target for a lineage system. Cui et. al [19] identified efficient tracing procedures for a number of operator properties. These procedures are then used to execute backward lineage queries. However, the model does not allow arbitrary operators to generate lineage, and models them as black-boxes. Section V describes several mechanisms (e.g., payload functions) that can implement many of these procedures.

Trio [5] was the first database implementation of cell-level lineage, and unified uncertainty and provenance under a single data and query model. Trio explicitly stores relationships between input and output tuples, and is analogous to the full provenance approach described in Section V.

The SubZero runtime API is inspired by the PASS [20], [21] provenance API. PASS is a file system that automatically stores provenance information of files and processes. Applications can use the *libpass* library to create abstract provenance objects and relationships between them, analagous to producing cell-level lineage. SubZero extends this API to support the semantics of common scientific provenance relationships.

X. Conclusion

This paper introduced SubZero, a scientific-oriented lineage storage and query system that stores a mix of black-box and fine-grained lineage. SubZero uses an optimization framework that picks the lineage representation on a per-operator basis that maximizes lineage query performance while staying within user constraints. In addition, we presented *region lineage*, which explicitly represents lineage relationships between sets of input and output data elements, along with a number of efficient encoding schemes. SubZero is heavily optimized for operators that can deterministically compute lineage from array cell coordinates and small amounts of operator-generated metadata. UDF developers expose lineage relationships and semantics by calling the runtime API and/or implementing mapping functions.

Our experiments show that many scientific operators can use our techniques to dramatically reduce the amount of redundant lineage that is generated and stored to improve query performance by up to $10\times$ while using up to $70\times$ less storage space as compared to existing cell-based strategies. The optimizer successfully scales the amount of lineage stored based on application constraints, and can improve the query performance of the genomics benchmark, which is amenable to black-box only strategies.. In conclusion, SubZero is an important initial step to make interactively querying fine-grained lineage a reality for scientific applications.

References

[1] Z. Ivezi, J. Tyson, E. Acosta, R. Allsman, S. Anderson, *et al.*, "LSST: From science drivers to reference design and anticipated data products." [Online]. Available: http://lsst.org/files/docs/overviewV2.0.pdf

[2] M. Stonebraker, J. Becla, D. J. DeWitt, K.-T. Lim, D. Maier, O. Ratzesberger, and S. B. Zdonik, "Requirements for science data bases and SciDB," in *CIDR*, 2009.

[3] T. Oinn, M. Greenwood, M. Addis, N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe, "Taverna: lessons in creating a workflow environment for the life sciences," in *Concurrency and Computation: Practice and Experience*, 2006.

[4] H. Kuehn, A. Liberzon, M. Reich, and J. P. Mesirov, "Using genepattern for gene expression analysis," *Curr. Protoc. Bioinform.*, Jun 2008.

[5] J. Widom, "Trio: A system for integrated management of data, accuracy, and lineage," Tech. Rep., 2004.

[6] P. Tamayo, Y.-J. Cho, A. Tsherniak, H. Greulich, *et al.*, "Predicting relapse in patients with medulloblastoma by integrating evidence from clinical and genomic features." *Journal of Clinical Oncology*, p. 29:14151423, 2011.

[7] R. Ikeda and J. Widom, "Panda: A system for provenance and data," in *IEEE Data Engineering Bulletin*, 2010. [Online]. Available: http://ilpubs.stanford.edu:8090/972/

[8] J. Cheney, L. Chiticariu, and W. C. Tan., "Provenance in databases: Why, how, and where," in *Foundations and Trends in Databases*, 2009.

[9] S. Davidson, S. Cohen-Boulakia, A. Eyal, B. Ludscher, T. McPhillips, S. Bowers, M. K. Anand, and J. Freire, "Provenance in scientific workflow systems."

[10] R. BOSE and J. FREW, "Lineage retrieval for scientific data processing: A survey," in *ACM Computing Surveys*, 2005.

[11] J. Goecks, A. Nekrutenko, J. Taylor, and T. G. Team, "Galaxy: a comprehensive approach for supporting accessible, reproducible, and transparent computational research in the life sciences." in *Genome Biology*, 2010.

[12] R. Ikeda, H. Park, and J. Widom, "Provenance for generalized map and reduce workflows," in *CIDR*, 2011. [Online]. Available: http://ilpubs.stanford.edu:8090/985/

[13] Y. Amsterdamer, S. Davidson, D. Deutch, T. Milo, J. Stoyanovich, and V. Tannen, "Putting lipstick on pig: Enabling database-style workflow provenance," in *PVLDB*, 2012.

[14] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig latin: A not-so-foreign language for data processing," in *SIGMOD*, 2008.

[15] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock, "Kepler: an extensible system for design and execution of scientific workflows," in *SSDM*, 2004.

[16] M. K. Anand, S. Bowers, T. McPhillips, and B. Ludscher, "Efficient provenance storage over nested data collections," in *EDBT*, 2009.

[17] P. Missier, N. Paton, and K. Belhajjame, "Fine-grained and efficient lineage querying of collection-based workflow provenance," in *EDBT*, 2010.

[18] A. P. Chapman, H. Jagadish, and P. Ramanan, "Efficient provenance storage," in *SIGMOD*, 2008.

[19] Y. Cui, J. Widom, and J. L. Viener, "Tracing the lineage of view data in a warehousing environment," in *ACM Transactions on Database Systems*, 1997.

[20] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer, "Provenance-aware storage systems," in *NetDB*, 2005.

[21] K.-K. Muniswamy-Reddy, J. Barillariy, U. Braun, D. A. Holland, D. Maclean, M. Seltzer, and S. D. Holland, "Layering in provenance-aware storage systems," Tech. Rep., 2008.