Queryable Compression of CamFlow Provenance Data

Aaron Bembenek
bembenek@g.harvard.edu

Lily Tsai
lilliantsai@college.harvard.edu

December 9, 2016

Abstract

Provenance describes the history of data in a system and can naturally be represented as a graph describing dependencies between different components in the system. Provenance graphs can grow to incredibly large sizes, motivating the need for compression schemes for provenance data storage in both memory and on disk. Optimally, a compression scheme will also support common queries on the encoded data. We introduce such a scheme for provenance data generated by the provenance-collecting system CamFlow, as well as a compression utility that implements this scheme and an interface for running queries on the compressed data. Our experiments show that we achieve compression ratios of around 11.5x relative to the raw JSON output of CamFlow and support reasonably efficient queries on the compressed graphs.

1 Introduction

Provenance is the history of how a piece of data came to be, including the processes used to generate that data, the inputs for those processes, and the environments under which those processes operated. As such, it ideally represents a complete computational record, to the extent that provenance hypothetically enables exact reproduction of a data item. Furthermore, because provenance systems often capture the computational record for many or all data items in the system, provenance data can give insight into common computational patterns and processing cycles.

Provenance data is often conceived as a directed acyclic graph. Provenance graphs can grow to enormous sizes because every operation in the system generates new nodes, edges, and their corresponding metadata. This is exacerbated by the fact that the size of the graph monotonically increases: information is never lost. Thus, for the sake of space efficiency, provenance graph compression is highly appealing. Furthermore, if the compression is carefully designed, it should be possible to perform queries on a compressed graph as efficiently as on the uncompressed version. Such queries allow users of a provenance system to debug or verify the origins of their data, ensure that their data can be authenticated by others, and easily reproduce results from previous scientific experiments.

This paper proposes a compression utility for provenance graphs and a query interface for these compressed graphs.[1] This utility will be used by the CamFlow project, a provenance capture system currently being developed and used at Harvard. We introduce the CamFlow system and provenance model as well as previous work on compressing provenance data in Section 2. Sections 3 and 4 describe our query model and compression schemes. Section 5 evaluates and discusses compression and query performance results, and Section 6 concludes.

2 Background and Related Work

2.1 Provenance and CamFlow

Provenance is naturally represented as a directed acyclic graph where, informally speaking, nodes represent causes and effects within the system, and an edge from one node to another represents that the two nodes

[1] Our implementation can be found at `https://github.com/aaronbembenek/prov-compress`.

are in a cause-and-effect relation where the source of the edge depends somehow on the edge's destination. Thus, to find all the causes that led to a particular effect, it is theoretically sufficient to follow the links upwards in the provenance graph from that effect node. While there are many models for provenance graphs, a common one is the W3C PROV model (Groth and Moreau [2013]), which specifies three different types of nodes: agents, entities, and activities. Entities consist of any logical object (physical or otherwise) such as a file or dataset, and activities consist of processes and other actions that create and modify entities. Activities are done on behalf of agents, and entities can be attributed to the activity of some agent. In this model, a provenance graph is comprised of labeled relationships between members of these categories. For instance, a "used-by" edge relation from an activity to an entity implies that the activity used the entity in some way; a concrete example is when a process (activity) reads a file (entity).

Our project is designed for the provenance-capture system CamFlow, a prototype system currently being developed at Harvard for the End-to-end Provenance project.[2] CamFlow works under the W3C Prov model, with node types {`agents, entities, activities`}, and edge relations types {`used, wasGeneratedBy, wasInformedBy, wasDerivedFrom`}. CamFlow also introduces "is-a-version-of" relations, which signify that the two nodes at the endpoints of the edge represent versions of the same object; in particular, the node at the tail is the version that directly supersedes the version represented by the node at the head. Versions are inserted into the provenance graph whenever there is an observable state change in an object. For instance, each time a task interacts with an external resource a new version of the task is created. Similarly, each time a file is written a new version of the file is created. We exploit the repetition introduced by versions in both our graph and metadata compression algorithms (described in Section 4). Versions provide fine-grained, temporally-sound information about the dependencies between different objects and also ensure that there are no cyclical dependencies (i.e., cycles) in the provenance graph.

2.2 Previous Work on Provenance Compression

While there is an abundance of research on graph compression and string compression, there is only a small existing literature specifically on provenance graph compression.

Chapman et al. [2008] exploit properties of provenance graphs in several compression schemes and identify multiple ways in which provenance graphs can contain a great deal of redundancy. They introduce three techniques to reduce provenance data storage costs—factorization, structural inheritance, and predicate-based inheritance—founded on the observation that provenance graphs often contain many identical (or extremely similar) subgraphs. Their provenance factorization scheme identifies these subgraphs and stores them separately from the rest of the provenance graph. Nodes then reference a single copy of the subgraph, cutting down on redundancy. Our graph compression technique is slightly similar in that it collapses parts of the provenance graph (namely versions of the same node), but otherwise their method is orthogonal to our own.

Predicate-based inheritance is reminiscent to our metadata compression scheme, but while they partition nodes into predicate-based groups and pull out common subsets of provenance from these groups, we use reference encoding to compress common provenance. Their structural inheritance algorithm omits provenance records only if the *entire* provenance record matches that of a direct ancestor; our metadata compression omits all parts of provenance records that match that of a previous version of a node.

Using their techniques, they reduce provenance data to 5%, 15%, and 12% respectively on MiMI, PReServ, and Karma provenance stores (comparable to our reduction of size to 9% on CamFlow). However, they envision a different model of provenance and we do not believe their system is directly applicable to provenance graphs generated by CamFlow. Furthermore, their compressed provenance data supports only queries about individual provenance records, unlike our query interface which supports additional queries about graph structure and node relations.

Cai et al. speculate that provenance graphs can be compressed by finding common *templates* within a graph using a hypothetical "powerful graph solving tool", but support this claim by running simulations on synthesized graphs. We exploit the semantics of CamFlow node types (particularly versions) to find

[2]See `http://end-to-end-provenance.github.io/`.

and reduce large subgraphs within the graph without relying on an efficient subgraph isomorphism solver. Furthermore, they achieve a maximum compression ratio of 2:1 on synthesized LittleJil data dependency graphs (50% of its original size, compared to our result of 9%). However, the data produced by LittleJil is radically different in size and format than that of CamFlow, making a fair comparison difficult.

Xie et al. [2011] and Xie et al. [2013] use a web graph compression scheme with a dictionary encoding of common metadata strings to encode provenance graphs. They base their algorithm on the WebGraph compression framework described by Boldi and Vigna [2004], which includes delta encoding the edges within an adjacency list as well as reference encoding adjacency lists. Xie et al. note that provenance graphs share certain properties with the web graph that make them amenable to similar compression techniques. Most importantly, both provenance graphs and the web graph have a natural ordering of nodes that exhibits *similarity* and *locality*. Similarity means that nodes that are close together in the ordering have similar neighbors, and locality means that nodes that are close together in the ordering are neighbors in the graph. Nodes in provenance graphs can be numbered in order of their creation. Nodes that were created in close succession were likely created by the same process, therefore sharing dependencies and exhibiting similarity. Furthermore, many nodes are probably created in close succession to the node that created them (and on which they are dependent); consequently, the natural ordering of provenance graphs should exhibit locality. The authors argue that other properties present in provenance graphs (and not necessarily in the web graph) can be used effectively during compression in addition to similarity and locality, such as the fact that nodes with the same name attribute often have similar dependencies (e.g., each time an executable is run, it requires the same libraries).

Our graph compression scheme is also inspired by web graph compression (for instance, we delta encode the edges in the adjacency list, à la Boldi and Vigna [2004]). Unlike Xie et al., instead of trying to find an ordering that is optimized for locality and similarity, we present an ordering of the nodes that tries to exploit a property specific to CamFlow-generated graphs (namely, that the same object is represented across a chain of versions). Additionally, we do not implement the full web graph compression scheme, such as reference encoding the adjacency list. Our work could be extended in this way.

Like Xie et al., our metadata compression also performs a dictionary encoding of common strings in the provenance data. However, we specialize encodings for built-in provenance strings and utilize delta encoding to capture provenance metadata shared between nodes. We also introduce special encodings for identifiers based on our node-ordering scheme.

Xie et. al achieve final compression sizes that are at best approximately 15% of the original size of the provenance data generated by the Karma system.

3 Query Interface

We provide an interface that enables a user to query different aspects of the provenance data. Our tool supports the following queries:

```
map<keys, values>            get_metadata(identifier);
vector<nodes>                get_all_ancestors(node);
vector<nodes>                get_direct_ancestors(node);
vector<nodes>                get_all_descendants(node);
vector<nodes>                get_direct_descendants(node);
vector<vector<nodes>>        all_paths(source_node, sink_node);
map<relation,vector<nodes>>  friends_of(file, task);
```

A user of our tool can acquire information about the provenance without manually investigating the graph. `get_metadata` provides a way to easily access information about any node or edge in the provenance graph, such as a node's type (file, task, etc.) or date of creation. Queries for ancestors and descendants allow a user to determine both local and global dependencies of specific nodes. `all_paths(A,B)` returns all the ways node B has influenced node A.

`friends_of` returns all the files that are related in the same way as the specified file to the specified task.

For example, if file A is created, read, and modified by task T, `friends_of(A, T)` will return all files created by task T, all files read by task T, and all files modified by task T (grouped by relation). A `friends` query of a file of interest returns other files that have been through similar processing cycles. This is particularly useful for users of scientific experiments or programs that run on large amounts of data.

This interface can be extended to provide other useful queries, such as queries to get all versions of a particular node or inputs/outputs of particular tasks (an extension of `friends`). We conjecture that our version compression scheme described in Section 4.2.2 will allow these types of queries to be executed more efficiently on a compressed graph than an uncompressed one.

For the purposes of comparing performance, we implement two versions of this interface: the first processes the original uncompressed JSON data, while the second works on our compressed provenance graph and metadata.

4 Compression Scheme

We provide a lossless compression scheme that allows a user to query parts of the provenance graph without decompressing the full graph. Our scheme consists of two separate algorithms, one for metadata compression, and the other for compression of the graph structure.

4.1 Metadata Compression


```
"entity": {"AAEAAAAAACDDXQEAAAAAAMVT1VmFSQxzAAAAAAAAAAA=": {
    "cf:id": "89539", "cf:type": "file", "cf:boot_id": 1507152837, "cf:machine_id": 1930185093,
    "cf:version": 0, "cf:date": "2016:11:30T00:11:48", "cf:jiffies": "4309511553", "cf:uid": 0,
    "cf:gid": 0, "prov:type": "file", "cf:mode": "0x81fd", "prov:label": "[file] 0",
    "cf:uuid": "57b7878c-46ea-86ca-d69e-82031b250b52"
  }}
}
"wasInformedBy": {"AABAAAAACIABAAAAAAAAAMVT1VmFSQxzAAAAAAAAAAA=": {
    "cf:id": "1", "cf:type": "version", "cf:boot_id": 1507152837,
    "cf:machine_id": 1930185093, "cf:date": "2016:11:30T00:11:48",
    "cf:jiffies": "4309511553", "prov:label": "version", "cf:allowed": "true",
    "prov:informant": "AQAAAAAAAEBRYAEAAAAAAMVT1VmFSQxzAAAAAAAAAAA=",
    "prov:informed": "AQAAAAAAAEBRYAEAAAAAAMVT1VmFSQxzAQAAAAAAAAA="
 }}
```


Figure 1: Sample Provenance Metadata

Each provenance node or edge is associated with a specific metadata entry consisting of a provenance type, identifier, and set of key-value pairs. Metadata is compressed using a mixture of reference encoding and dictionary-based encodings.

We perform reference encoding by constructing default metadata entries for both node metadata and edge metadata. The default node metadata contains a value for each key present in any node metadata entry (similarly for the default edge metadata). For every metadata entry, we record only the differences from the values in the default metadata. We use special encodings for values such as dates when recording differences. Most of the CamFlow metadata share values for keys such as `cf:boot_id` or `cf:type`, and reference encoding allows us to exploit this similarity.

We then perform a second iteration of reference encoding that encodes the metadata of a version of a particular node with reference to the first version's metadata. This exploits a semantic property of provenance graphs: if a node is a version type, this means that the node was introduced to prevent dependency cycles in the graph. As such, versions nodes derived from the same node have nearly identical metadata.

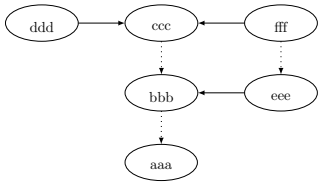


Figure 2: Running example. Nodes are labeled with mock CamFlow identifiers. Dashed arrows represent version edges.

Dictionary-based encoding is performed using both built-in provenance keyword dictionaries and a dictionary of common strings found within the specific provenance metadata generated for a program. Built-in provenance keywords (provenance types, keys, and values) are encoded as fixed-length bit strings. CamFlow uses a fixed set of keywords to represent particular keys and values, which makes dictionary-based encoding a particularly effective encoding method. We also construct a dictionary of the top 300 most common strings in the provenance metadata (restricted to strings that are not provenance keywords) and encode these values as fixed-length bit strings. This allows us to capture repetition of values such as a node's `UUID` that are specific to an individual program's provenance, but occur with relatively high frequencies. For any string that cannot be encoded using one of the built-in keywords or does not commonly occur, we record its length (restricted to an 8-bit integer) and record the string directly.

We perform a special encoding for node and edge identifiers after observing that identifiers (strings of over 20 chars) comprised a large portion of the metadata (un-encoded, the identifiers would comprise over 33% of the total compressed metadata). We note that there is an identifier for each metadata entry and two identifiers for each edge (specifying the nodes joined by the edge). For example, in Figure 1, we see the metadata for edge relation `wasInformedBy` includes three identifiers: its own identifier, and the identifiers of the two nodes related by the edge. To reduce the space taken by identifiers, we encode identifiers as follows: all node identifiers are mapped to `Node_Ids` (see Section 4.2.1), which are integers representable in $\log_2 n$ bits, where n is the number of nodes in the graph. All edge identifiers are mapped to integers of bit length $2\log_2 n$: the most significant $\log_2 n$ represent the `Node_Id` of the head node of the edge, and the lower $\log_2 n$ bits represent the `Node_Id` of the tail node. This means that we do not need to record the keys for the head/tail of the edge (at the cost of using slightly larger integers to represent an edge identifier). We list the identifiers in order of increasing `Node_Id`, allowing us to associate an identifier with its metadata without any explicit mapping.

A decoder requires the dictionary of common strings, the list of identifiers, and the encoded metadata. It is assumed that the the provenance keyword encodings are already known by the decompressor. Because the list of identifiers and the dictionary of common strings must necessarily be decompressed for our querier to work, we use `xz` to compress the corresponding files (and later to decompress the list/dictionary for querying). Note, however, that we do not use `xz` to process the encoded metadata so that we can query the metadata in compressed form.

4.2 Graph Compression

While the metadata contains enough information to reconstruct the provenance graph, it does not support efficient graph queries. To enable faster graph queries, we explicitly encode the provenance graph in addition to the metadata. In the following subsections we describe the way we encode the compressed graph and give some intuition for how our encoding supports relatively efficient graph operations. We use the graph presented in Figure 2 as the basis for a running example. As described in Section 2.1, an edge indicates that the node at the tail of the edge depends on the node at the head. Dashed arrows in our figures represent version edges.

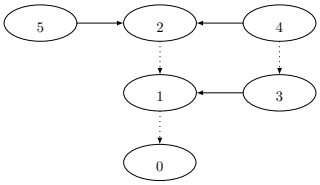


Figure 3: Running example. Nodes are labeled with `Node_Id`s assigned during a breadth-first search on the transpose of the aggregate graph.

4.2.1 Node Ordering

As explained in Section 2.2, the effectiveness of some graph compression schemes relies on finding an ordering of the nodes that enables efficient encoding. The first priority of our numbering scheme is to number versions of the same object consecutively, with earlier versions in the chain receiving lower `Node_Id`s (recall that CamFlow represents some objects by multiple nodes that are connected by version edges). The second priority is to find an ordering such that nodes that share an edge are close together in the ordering. We assign numbers during a breadth-first search on the transpose of what we call the aggregate graph, which is a graph where all nodes that represent a single object (i.e., are connected by version edges) are aggregated into a single node. When one of these nodes is encountered during the graph traversal, we number all of the nodes it has subsumed with consecutive numbers. We found empirically that the order given by a breadth-first search on the transpose of the aggregate graph led to better compression than a breadth-first search in the forward direction or a depth-first search in either direction. Figure 3 shows the assignment of `Node_Id`s in our running example.

4.2.2 Version compression

Version edges are extremely common in CamFlow-generated provenance graphs. In Table 1 we present statistics on a sample of provenance graphs generated by CamFlow during machine-learning experiments. These results are representative of the same statistics for the other graphs in our benchmark suite. As is clear from this table, the number of distinct objects is typically very small compared to the number of nodes in the graph, and version edges make up a large proportion (roughly 50%) of all edges in the graph. Our graph encoding takes advantage of this fact by collapsing together nodes that are versions of the same object and only implicitly representing version edges.

Program	Total nodes	Distinct objects	Total edges	Version edges
`face_recognition.py`	9973	1545	17172	8428
`plot_prediction_latency.py`	19016	1553	35250	17463
`plot_stock_market.py`	18046	1597	33227	16449
`svm_gui.py`	7031	1568	11273	5463

Table 1: Statistics on CamFlow graphs generated during machine-learning workloads.

There are several important properties of object versions that we exploit in our graph encoding. The first is that nodes that are connected by version edges occur in (often long) chains; that is, there is at most one version edge from a node and never more than one version that directly follows another version. In the example given in Figure 3, the nodes 0, 1, and 2 form such a chain and 3 and 4 form another chain. During pre-processing, we number the nodes in each chain consecutively, starting from the earliest version. If we know the `Node_Id` assigned to this first version and we know the total number of nodes in the chain, we can derive all the relevant edges between versions. Say that the first version is assigned `Node_Id` n and that there are k versions in total for that object. Without explicitly representing any per-edge information, we know that there is an edge $(i, i-1)$ for all $n < i < n+k$. Thus, we can get by with only encoding the `Node_Id` of the earliest version of each object and the number of versions of that object, and do not need to

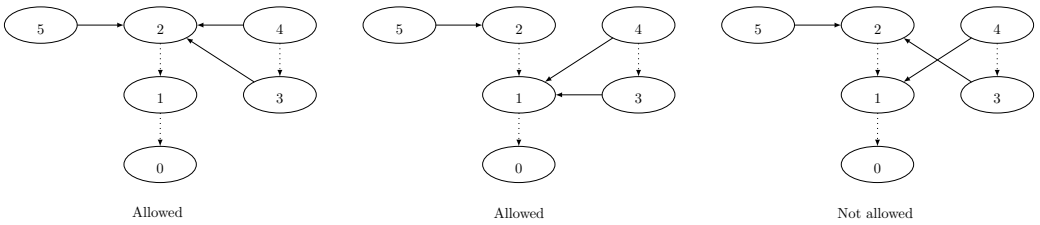


Figure 4: Variations on running example. The rightmost example is disallowed by the monotonicity of cross-chain edges. This is visually apparent because the edges $(3, 2)$ and $(4, 1)$ cross.

`Node_Id`	Out-degree	Out-edges	In-degree	In-edges
0	0		3	3, 4, 5
3	2	1, 2	0	
5	1	2	0	

Figure 5: Running example. Adjacency list representation of aggregate graph.

explicitly represent version edges.

We also exploit the fact that dependencies between versions of different objects are consistent. Say that we have two distinct chains of versions $\langle a_0, \ldots, a_n \rangle$ (representing object A) and $\langle b_0, \ldots, b_m \rangle$ (representing object B), where a_i is the version that precedes a_{i+1} and b_i is the version that precedes b_{i+1}. Say that there are at least two edges from nodes representing object A to nodes representing object B. Choose any two of these edges (a_i, b_k) and (a_j, b_l) where $i < j$. It must be that $l \geq k$. Otherwise, a_j would directly depend on an earlier version of B than the version a_i depends on, even though a_j is a later version of A than a_i. This would be temporally inconsistent, since by the time A is in the state represented by a_j, B must be in at least the state represented by b_k and therefore A in state a_j could not interact with an earlier state of B than that represented by b_k. Figure 4 gives some visual intuition for this restriction, which we will exploit in our encoding when we aggregate together all the nodes within a chain. Hereafter we refer to this property as the monotonicity of cross-chain edges.

In our encoding, we represent the provenance graph using an adjacency list representation. The adjacency list for each node includes both out-edges and in-edges to more efficiently support a range of graph queries. For the sake of simplicity, we discuss only out-edges here (in-edges are treated according to a similar scheme). We encode the aggregate graph previously described in Section 4.2.1; that is, versions of the same object are aggregated together into a single node numbered with the `Node_Id` of the earliest version in the chain.[3] In the adjacency list for an aggregate node, we list the destination of all the out-edges that start at any version subsumed by that aggregate, *except for version edges*. The edges are listed in non-decreasing order of destination `Node_Id` (note that an adjacency list may include duplicates). Crucially, we list the `Node_Id` of the actual destination in the original graph of an out-edge, and not the `Node_Id` of the aggregated node that has subsumed that destination node in the aggregate graph. Figure 5 shows the adjacency list for our running example. Note that the entry for 3 has out-edges listed to 1 and 2—reflecting the edges $(3, 1)$ and $(4, 2)$—and not to 0, the `Node_Id` for the aggregated node for the chain $\langle 0, 1, 2 \rangle$. The entry for node 0 has no out-edges since none of the nodes in the chain has an out-edge to a node outside the chain.

We need to support any queries that can be run on the original, un-aggregated graph, which means that we need to be able to disaggregate the adjacency list for an aggregate node and determine the actual source for each edge in the adjacency list. Looking at a node in isolation, there is not enough information to make this determination. However, given any two aggregate nodes, we can reconstruct all edges between any individual nodes subsumed by the aggregate nodes. Say we have two objects A and B that correspond to chains with `Node_Id`s $\langle a_0, \ldots, a_n \rangle$ and $\langle b_0, \ldots, b_m \rangle$, respectively, in the original graph, where $a_{i+1} = a_i + 1$ and $b_{i+1} = b_i + 1$. To find the exact edges between the chains for A and B in the original graph, we first find

[3]We refer to the un-aggregated graph parsed directly from the CamFlow output as the "original graph."

Node_Id	Out-degree	Out-edges	In-degree	In-edges
0	0		3	3, 1, 1
3	2	5, 1	0	
5	1	7	0	

Figure 6: Running example. Delta-encoded adjacency list representation of aggregate graph.

the edges from any version of A to B. These will be manifest in our encoded graph as a sorted contiguous block of Node_Ids in the adjacency list for a_0 in the range $[b_0, b_m]$. Call this list $L^B = \langle L_0^B, \ldots, L_k^B \rangle$. We then examine the adjacency list in the encoded graph for b_0 and find all incoming edges in the range $[a_0, a_n]$ (which will also appear as a sorted contiguous block). Call this list $L^A = \langle L_0^A, \ldots, L_k^A \rangle$.

Note that the lengths of L^A and L^B must be equal to the number of edges from any version of A to any version of B, since each edge from a node in A to a node in B appears exactly once in the outgoing list for a_0 and exactly once in the incoming list for b_0. Furthermore, we claim that (L_i^A, L_i^B) is an edge in the original graph, and that each edge between A and B in the original graph corresponds to exactly one pair (L_i^A, L_i^B). This follows from an inductive argument that leverages the monotonicity of cross-chain edges. For the base case, recognize that (L_0^A, L_0^B) must be an edge in the original graph. Since L_0^A is in the incoming adjacency list for b_0, L_0^A must be the source of some edge leading to a node in B. Furthermore, L_0^A is the earliest version of A to go to a node in B, since the Node_Ids in the adjacency lists appear in sorted order. This means that L_0^A must have an edge to L_0^B, since L_0^B is the earliest version of B that is the endpoint of an edge starting in A (no version later than L_0^A can point to L_0^B by the monotonicity of cross-chain edges). Thus, we have accounted for exactly one edge between the chains A and B: (L_0^A, L_0^B). Since the endpoints for an edge show up exactly once in the outgoing list for A and the incoming list for B, L_0^A cannot be the source of any other edge going from A to B, and L_0^B cannot be the destination of any other edge in this set. Thus, to determine the other k edges from A to B, we need only to concern ourselves with the lists $(L_1^A, \ldots, L_k^A)$ and $(L_1^B, \ldots, L_k^B)$. An analogous argument to the base case shows that (L_1^A, L_1^B) is an edge, and the argument extends naturally to the remaining edges.

For a concrete example, consider the problem of trying to find the edges from the chain $A = \langle 3, 4 \rangle$ to the chain $B = \langle 0, 1, 2 \rangle$ in our running example (see Figures 3 and 5). By looking at the outgoing edges for 3 in our encoded graph, we see that there are edges from some nodes in A to the nodes 1 and 2. Looking at the incoming edges for 0, we see that there are incoming edges from 3 and 4. Taking $L^A = \langle 3, 4 \rangle$ and $L^B = \langle 1, 2 \rangle$, we determine that in the original graph the only edges between the chains A and B are $(3, 1)$ and $(4, 2)$. To determine all of node 4's outgoing edges, we check to see if node 4 is the earliest version in its chain; since it is not, we know that it has an edge to its predecessor 3. These version edges do not need to be represented explicitly in our encoding.

4.2.3 Delta encoding and header information

We also apply delta encoding to the adjacency lists for the aggregate nodes in the encoded graph. Say that the adjacency list for an aggregate node with Node_Id a is $\langle a_0, \ldots, a_n \rangle$, where each a_i is a Node_Id. We recode this as a list $\langle \delta_0, \ldots, \delta_n \rangle$ where

$$\delta_0 = \begin{cases} 2(a_0 - a) & a_0 > a \\ 2(a - a_0) + 1 & a_0 < a \end{cases}$$

and $\delta_{i+1} = a_{i+1} - a_i$. The special case arises because the difference $a_0 - a$ can be negative (the rest of the deltas must be positive, since the adjacency list is sorted). This delta encoding is inspired by Boldi and Vigna [2004], who use delta encoding to encode the adjacency lists of the web graph. See Figure 6 for the delta-encoded version of our running example.

As a header to our encoding, we need to include some basic metadata necessary to interpret our encoded graph, such as the number of bits used to store various pieces of information. Because we hypothesize

that nodes that represent the aggregation of multiple versions have a different shape than nodes that only represent a single node in the original graph, we record different numbers for the number of bits needed to represent information for these two classes of nodes. Similarly, because the transpose of the graph has a very different shape than the graph itself, we record different numbers for the number of bits needed to represent information for out-edges and in-edges. In future work, we could explore using even finer-grained information (e.g., record the number of bits need to represent the out-degree of an aggregate node that represents a chain of between 25 and 50 versions).

We also need to encode enough information to recover the `Node_Id`s of the nodes in the aggregate graph and find where the information of each node is in the block of encoded node data. We do this by encoding a list of pairs. The ith pair provides information for the ith node in the aggregated graph (when the nodes are listed in sorted order by `Node_Id`, which is how they appear in the encoded adjacency list representation). The first element in the pair is the location in the block of encoded nodes of the adjacency list entry for the ith aggregate node; the second element is the number of nodes in the original graph that were aggregated into the ith aggregate node. The first elements are delta encoded across the pairs. The purpose of this list, and its encoding, is similar to that of the offset array in Boldi and Vigna [2004]. When a query mechanism loads our encoded graph, it must first read the header information and this list of pairs so that it knows how to index into and extract information from the encoded node data.

5 Evaluation

We evaluate our system on two dimensions. The first dimension is space usage: we compare a graph compressed using our utility to the uncompressed graph and the graph compressed using a generic compression utility like `xz`. The second dimension of our evaluation is query performance and memory usage, which involves comparing time spent and memory used while querying compressed and uncompressed versions of the same graph.

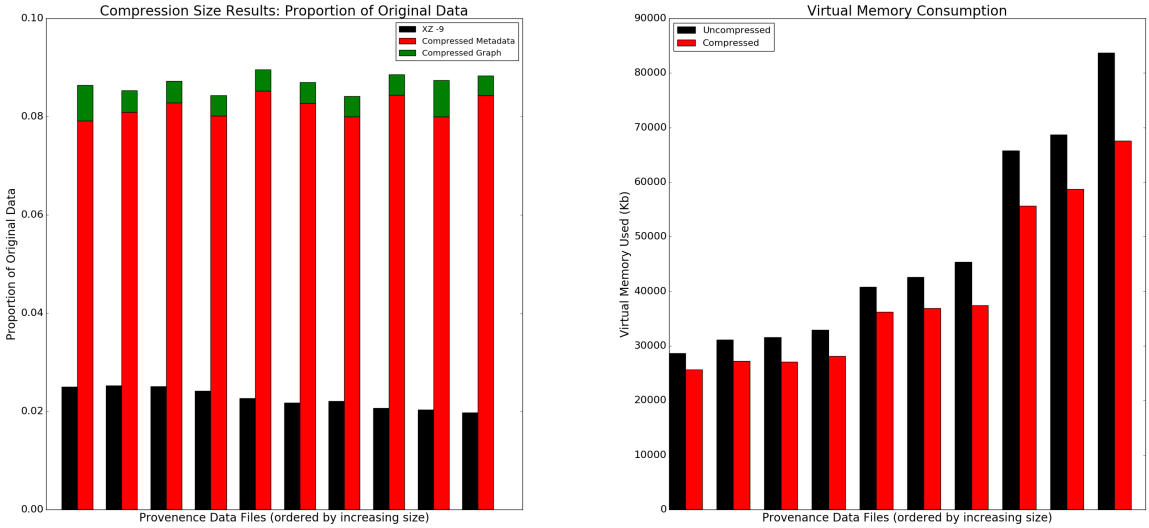
Program	Size (Kb)
`topics_extraction_with_nmf_lda.py`	4980
`plot_out_of_core_classification.py`	5819
`svm_gui.py`	5982
`plot_tomography_l1_reconstruction.py`	6514
`face_recognition.py`	9067
`plot_species_distribution_modeling.py`	9681
`plot_outlier_detection_housing.py`	10707
`plot_stock_market.py`	17518
`plot_prediction_latency.py`	18581
`plot_model_complexity_influence.py`	24046

Table 2: Size of CamFlow data produced during machine-learning workloads.

The provenance data used for evaluation is generated from a set of example machine learning Python programs taken from the scikit-learn package.[4] These programs are representative of experiments in the machine learning domain that read from different sets of input data files, manipulate this data, and write to multiple output files. Provenance from machine learning programs can potentially provide users valuable information about how training data influences the decisions made by the program. The provenance generated by our example programs range from 5MB to 24MB as shown in Table 2, allowing us to measure how our performance scales.

The compression utility we use for comparison is `xz` run with argument (-9), which provides the best compression `xz` can achieve at the expense of a slightly longer compression time. We choose `xz` because it

[4]See `http://scikit-learn.org/stable/auto_examples/`.



(a) Compressed Size (proportional to original)

(b) Querier Virtual Memory Usage

Figure 7: Disk and Memory Consumption of Compressed Data

outperforms all other commonly used compression utilities such as `gzip` or `bzip/bzip2` on our generated provenance data. `xz` provides a lower bound on the smallest size of compressed data we can hope to achieve.

We compress the provenance data to 8.3-8.8% of its original size, performing approximately 4 times worse than `xz`, which compresses the data to 2.0-2.5% of the original size. Our results compare favorably to those of previous work on large provenance systems such as Chapman et al. [2008], Xie et al. [2013], and Cai et al., some of which observed reductions to only 50% of the original data size with their compression scheme. As expected, the compressed metadata comprises more than 80% of the total compressed provenance data. The compressed graph is included as an added piece of information that allow users to perform graph queries on compressed data without unpacking the compressed metadata.

We also tested our final graph compression scheme, which uses both version aggregation and delta encoding, against a scheme that uses only delta encoding. We found that our final scheme produced encoded graphs that were 33-50% of the size of those produced by the alternate compression scheme. These numbers perhaps reflect the fact that version edges make up around 50% of the edges in the graphs on which we benchmarked our compression, and our final compression scheme does not explicitly represent these edges. There might also be some additional space savings because the final scheme compresses aggregate graphs, which have far fewer nodes than the original graphs (although these aggregate nodes are of higher degree and are thus likely to be more expensive on a per-node basis).

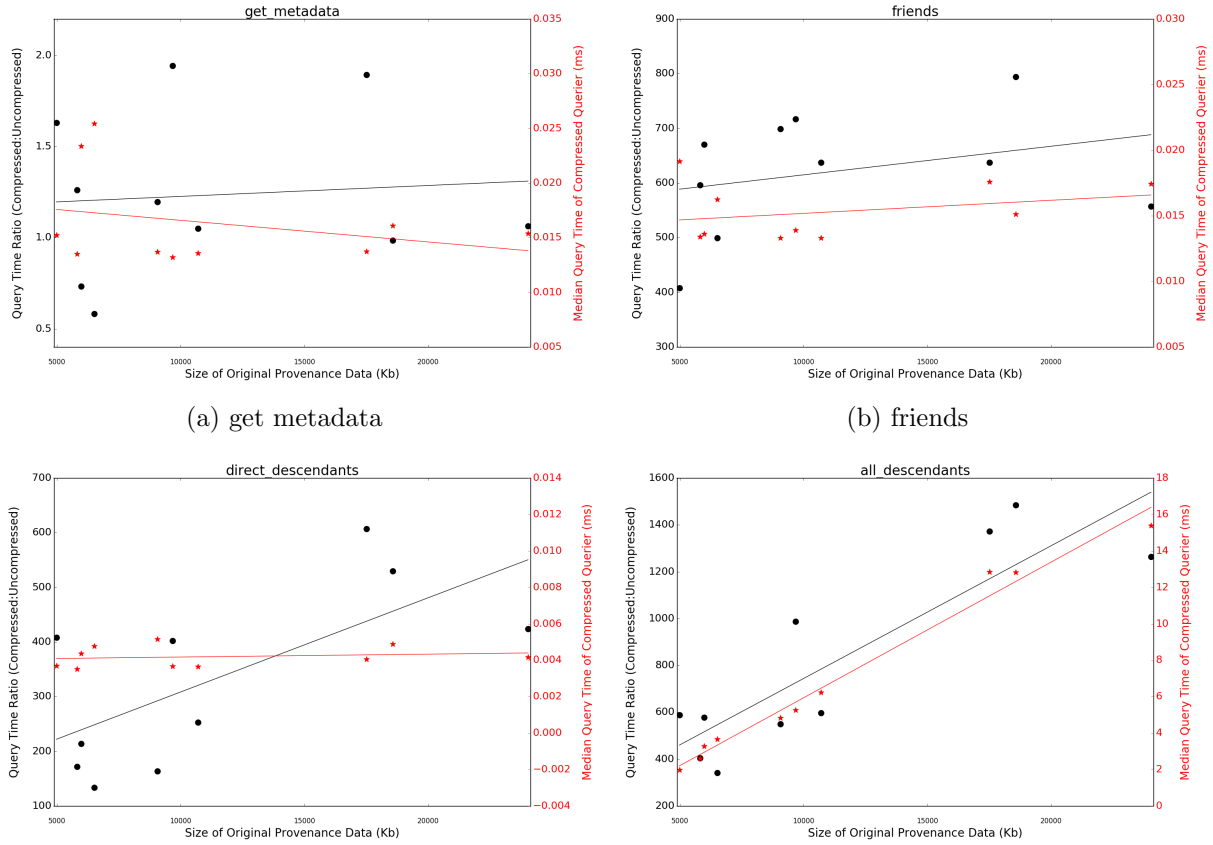
Our compression scheme performs equally well on small provenance data as on larger provenance data, unlike `xz`, which achieves lower compression ratios on smaller provenance data.

We find the the time taken to compress ranges from 1 to 8 seconds and scales linearly with the size of the original provenance data; this is expected given the linear-time algorithms used in our compression schemes.

5.1 Query Performance

We test query performance on a 12GB DRAM machine with one 2-core Intel(R) Core(TM) i5-5300U processor clocked at 2.30GHz. Hyperthreading is enabled, resulting in 4 available logical cores. The machine runs a 64-bit Linux 3.2.0 operating system, and all code is compiled with `g++-5.3`.

All queries except for `friends_of` are run on a sample of 100 node identifiers present in the graph. The `friends_of` query is run on a sample of file identifiers and task identifiers in the graph.



(a) get metadata

(b) friends

(c) direct descendants

(d) all descendants

Figure 8: Querier Performance on Compressed Provenance Data

In the following discussion, we refer to the querier operating on compressed data as Q_c and the querier operating on uncompressed data as Q_u. The performance results of Q_c and Q_u are depicted in Figure 8. We show the ratio of query times of Q_c to query times of Q_u, as well as the median of all sample times taken by Q_c to perform the query. We include a line of best fit for both the ratios and the medians to demonstrate how these scale as provenance data size increases. We omit graphs for ancestor queries because the results for ancestors queries are similar those for descendant queries.

Metadata query performance results are shown in Figure 8a. Queries by Q_c and Q_u perform approximately equally well (Q_c performs 0.6-2 times slower than Q_u), with performance staying approximately constant even as the provenance data size grows (our fastest and slowest results differ by only 0.012ms). We attribute this to the fact that at most two metadata entries need to be unpacked by Q_c for any one metadata query: if the node under question is a version, its metadata is reference encoded off of the metadata of the first version seen. Thus, the metadata for the first version would be unpacked, followed by that of the node under question (operations taking time $O(m)$, where m is the length of the node's metadata). Q_u must also perform a constant-time lookup for each metadata query followed by an $O(m)$ operation. This results in a negligible difference between Q_u and Q_c's metadata query performances.

We note that Q_u's performance on descendant, ancestor, and friend queries should be expected to be near optimal: upon initialization, Q_u iterates through the full, uncompressed metadata and constructs maps from each node to its descendants, ancestors, and friends. Q_c unpacks portions of the compressed metadata or graph as necessary, meaning that more work is required for any particular query. This also affects how each querier's performance scales as the size of the provenance data increases. We also note that the variance in our results is quite large, which is a consequence of noise in our sampling technique (e.g., different nodes may have different numbers of ancestors).

Unsurprisingly, Q_u performs direct descendant and ancestor queries in time $O(1)$: performance is constant regardless of the size of the provenance data. Q_c's performance of these queries depends on the number of versions of a node and the incoming/outgoing edges to/from these versions, which varies by program (see Table 1). However, we note that query performance of Q_c does seem to be independent of the size of the provenance data. As shown in Figure 8c, although Q_c performs anywhere from 110-600 times worse than Q_u (we attribute the large range to noise in our data), each query still takes less than 0.006ms.

Queries for all descendants or ancestors naturally take longer as the size of the provenance data grows. However, performance of Q_c worsens at a greater rate than does performance of Q_u as the size of the provenance data increases: we see a linear trend in the query time ratio compared to provenance data size. As shown in Figure 8d, Q_c performs on approximately 300 times worse than Q_u on small provenance data, and up to 1600 times worse on large provenance data. We note, however, that all queries are still on the order of a few milliseconds.

The results for the `friends_of` query in Figure 8b are similar to those for direct descendants/ancestors: Q_u achieves $O(1)$ query performance and Q_c's query performance depends on the number of versions in the graph and edges to/from version nodes. Q_u constructs a constant-time-access mapping from file→task→friend-files at initialization, whereas Q_c conserves memory (by not constructing such mappings) and requires more work to execute a particular query. We see that Q_c ranges from 400-800 times worse than Q_u. However, all query times remain relatively constant (Q_c's fastest and slowest results differ by only 0.007ms), and the slowest query performed by Q_c requires only 0.02ms.

We evaluate querier memory usage in addition to query times (see Figure 7b). The amount of memory used to construct a querier operating on uncompressed provenance data can be up to 110% of the memory used to construct a querier for our compressed data. This difference in memory usage increases as the size of the original provenance data increases. This is a promising result because memory usage will become a bottleneck for very large provenance graphs and we expect our querier will provide significant memory savings on particularly large graphs. The querier for compressed data can also be further optimized to reduce its memory footprint. However, the querier for uncompressed data must necessarily keep all the data in memory (or significantly increase its query times if it reads from disk).

6 Conclusion

We provide a compression scheme for provenance data generated by the CamFlow provenance system that reduces the size of the data to around 8-9% of the original size. We also describe and evaluate a query interface on the compressed provenance, and show that performing queries on compressed provenance data instead of on the original provenance data can reduce the memory footprint of the querier. Queries are slower on the compressed graph than an uncompressed version, but do not seem to be prohibitively expensive.

Because we separate the graph and metadata compression, our scheme would also be usable by a client that is only interested in provenance node relations and graph structure. In this scenario, we can relay just the compressed graph (and not the metadata) to the user in a queryable form and reduce the memory footprint of the compressed-data querier even further.

There are many possible directions for future work and optimizations. In terms of compressing the graph itself, we could explore different ways to order the nodes in the aggregated graph, and could also experiment with applying some of the more sophisticated web graph encoding techniques, such as the reference encoding of adjacency lists. An interesting possible direction could be developing a compression scheme that allows CamFlow to generate an encoded graph on-the-fly, which might require updating already encoded information. We can also specialize our metadata compression scheme further by choosing an "optimal" reference node or edge, or even an "optimal" set of nodes or edges. However, finding an optimum would further increase compression time. As noted in Section 3, our graph compression scheme should provide efficient implementations of additional queries such as `get_all_versions` that can be added to our query interface.

References

P. Boldi and S. Vigna. The webgraph framework i: Compression techniques. In *Proceedings of the 13th International Conference on World Wide Web*, WWW '04, pages 595–602, New York, NY, USA, 2004. ACM. ISBN 1-58113-844-X. doi: 10.1145/988672.988752. URL `http://doi.acm.org/10.1145/988672.988752`.

Diana Cai, Youngjune Gwon, and Loren McGinnis. Storing, compressing, and querying the little-jil workflow provenance. Unpublished class final project.

Adriane P. Chapman, H. V. Jagadish, and Prakash Ramanan. Efficient provenance storage. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, SIGMOD '08, pages 993–1006, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-102-6. doi: 10.1145/1376616.1376715. URL `http://doi.acm.org.ezp-prod1.hul.harvard.edu/10.1145/1376616.1376715`.

Paul Groth and Luc Moreau. PROV-overview. W3C note, W3C, April 2013. http://www.w3.org/TR/2013/NOTE-prov-overview-20130430/.

Yulai Xie, Kiran-Kumar Muniswamy-Reddy, Dan Feng, Yan Li, and Darrell D. E. Long. Evaluation of a hybrid approach for efficient provenance storage. *Trans. Storage*, 9(4):14:1–14:29, November 2013. ISSN 1553-3077. doi: 10.1145/2501986. URL `http://doi.acm.org/10.1145/2501986`.

Yulai Xie et al. Compressing provenance graphs. In *The 3rd USENIX Workshop on the Theory and Practice of Provenance*, 2011.