Evaluation of a Hybrid Approach for Efficient Provenance Storage

YULAI XIE, Huazhong University of Science and Technology
KIRAN-KUMAR MUNISWAMY-REDDY, Harvard University
DAN FENG, Huazhong University of Science and Technology
YAN LI and DARRELL D. E. LONG, University of California, Santa Cruz

Provenance is the metadata that describes the history of objects. Provenance provides new functionality in a variety of areas, including experimental documentation, debugging, search, and security. As a result, a number of groups have built systems to capture provenance. Most of these systems focus on provenance collection, a few systems focus on building applications that use the provenance, but all of these systems ignore an important aspect: efficient long-term storage of provenance.

In this article, we first analyze the provenance collected from multiple workloads and characterize the properties of provenance with respect to long-term storage. We then propose a hybrid scheme that takes advantage of the graph structure of provenance data and the inherent duplication in provenance data. Our evaluation indicates that our hybrid scheme, a combination of Web graph compression (adapted for provenance) and dictionary encoding, provides the best trade-off in terms of compression ratio, compression time, and query performance when compared to other compression schemes.

14

Categories and Subject Descriptors: E.4 [**Coding and Information Theory**]: Data Compaction and Compression; H.3.2 [**Information Storage and Retrieval**]: Information Storage—*File organization*

General Terms: Algorithms, Experimentation, Performance

Additional Key Words and Phrases: Provenance graphs, Web compression, dictionary encoding, storage

ACM Reference Format:
Xie, Y., Muniswamy-Reddy, K.-K., Feng, D., Li, Y., and Long, D. D. E. 2013. Evaluation of a hybrid approach for efficient provenance storage. *ACM Trans. Storage* 9, 4, Article 14 (November 2013), 29 pages.
DOI:http://dx.doi.org/10.1145/2501986

1. INTRODUCTION

Provenance is the metadata that represents the history or lineage of a data object. Provenance provides answers to questions like: Where was this object from? How are the ancestries of this object different from other objects? What kind of objects were used to create this object? Provenance has applications in various areas in the real world, such as experimental documentation, debugging, search [Shah et al. 2007], and security [King and Chen 2003].

This work was supported in part by the National Basic Research 973 Program of China under grant no. 2011CB302301, NSFC No. 61025008, 61173043, 61232004. This work was also supported in part by the National Science Foundation under awards IIP-0934401 and CCF-0937938, and by the Department of Energy under award DE-FC02-10ER26017/DE-SC0005417.
Authors' addresses: Y. Xie (corresponding author), Wuhan National Laboratory for Optoelectronics, School of Computer Science, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan, China, 430074; email: yulai.xiexie@gmail.com; K.-K. Muniswamy-Reddy, Harvard School of Engineering and Applied Science, 33 Oxford St., Cambridge, MA 02138; D. Feng, Wuhan National Laboratory for Optoelectronics, School of Computer Science, Huazhong University of Science and Technology, 1037 Luoyu Road, Wuhan, China, 430074; Y. Li and D. D. E. Long, Baskin School of Engineering, University of California, Santa Cruz, 1156 High St., Santa Cruz, CA 95064.
Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.
© 2013 ACM 1553-3077/2013/11-ART14 $15.00
DOI:http://dx.doi.org/10.1145/2501986

The provenance community has built a number of systems to collect provenance. They include database provenance systems such as Trio [Widom 2005], which records provenance of the tuples for a database system, Service-Oriented Architecture (SOA) provenance systems [Groth et al. 2006], which are typically associated with workflow engines that record provenance at the level of workflow stages, and system-call-based systems such as ES3 [Bose and Frew 2004], TREC [Vahdat and Anderson 1997], and PASS [Muniswamy-Reddy et al. 2006, 2009], which operate at the system call level and record the provenance of processes and files. There are also application-level solutions that record provenance at the level of business objects.

While these systems are a great step towards making provenance available to users, they neglect a crucial aspect that is needed for them to be practical: efficient provenance storage. Unoptimized provenance storage can take up a substantial amount of space. For instance, the base data in the PReServ [Groth et al. 2005] provenance store was 100KB, but the provenance exceeded 1MB. In MiMI [Jayapandian et al. 2007], an online database used to store protein information, the size of provenance (6GB) also far exceeded the size of original data (270MB). Similar results are also observed in other systems [Muniswamy-Reddy et al. 2006, 2009; Simmhan et al. 2006].

While reducing provenance size is important, we cannot, however, directly apply existing compression techniques to compress provenance, because the structure and access patterns of provenance are vastly different from regular data.

— Provenance needs to be queriable. If provenance is not readily queriable, then the value of provenance is reduced and users will not use the provenance. This precludes us from using techniques, like gzip, that allow for efficient storage, but are inefficient for query as uncompressing many bundles of provenance is needed before we can look up the necessary data.
— Provenance is a graph of connected objects. Its structural characteristics are similar to Web graphs. This implies that we can reuse some of the techniques developed for compressing Web graphs.
— Lossy compression techniques are not applicable, because losing an edge means a disconnect in the provenance graph. This can mean that users might miss a vital connection between objects.
— The provenance of an object is constantly evolving as it is modified, implying that provenance records of an object are randomly scattered around. Hence, we need algorithms that can handle this randomization.
— Provenance can also have a great deal of duplication, implying that deduplication can help us store provenance efficiently.

Accordingly, we developed a hybrid method for compressing provenance graphs by combining Web-graph-based compression and dictionary-based compression algorithms. The Web graph compression algorithm allows us to compress provenance while still satisfying the characteristics we observed. Dictionary encoding, thanks to its flexible processing granularity, allows us to eliminate repeated strings or substrings in the provenance graphs. Then, we compared the performance of this hybrid approach with a compression scheme designed explicitly for provenance compression: the Factorization And Inheritance (FAI) method proposed by [Chapman et al. 2008]. Our results indicate that our hybrid approach provides a better trade-off in terms of compression ratio, compression time, and query performance when compared to FAI compression schemes.

The contributions of this article are as follows.

(1) We identify the properties of provenance graphs that a practical and efficient compression scheme should consider.

(2) We present a hybrid approach that combines Web graphs compression and a dictionary encoding scheme to efficiently compress provenance graphs.
(3) We present a detailed evaluation of the different compression schemes along various axes including space, query performance, and compression time. We perform this analysis on provenance traces from a set of different provenance systems, such as PASS [Muniswamy-Reddy et al. 2006, 2009] and Karma [Cao et al. 2009].

The rest of the article is organized as follows. We present the background relevant to this article in Section 2. We analyze the properties of provenance graphs in Section 3. We explain the provenance compression techniques in detail in Section 4. In Section 5, Section 6, and Section 7, we evaluate how various compression methods perform on PASS traces and Karma traces, respectively. We discuss the related work in Section 8. In Section 9, we conclude.

2. BACKGROUND

Provenance is abstractly represented as a Directed Acyclic Graph (DAG) in most systems. The nodes in the graph represent the objects whose provenance we are storing. The edges represent relationships between the objects. A node records information necessary to identify an object, for example, the name of a file or the identifier of a process, while an edge indicates ancestor information or the dependency relationship between nodes. The structure of the graphs varies across different provenance models. In this section, we first explain three provenance models that the current provenance systems have used. Then, we describe the Factorization And Inheritance (FAI) compression scheme, which to the best of our knowledge, is the only compression scheme designed explicitly for compressing provenance.

2.1. Provenance Data Models

2.1.1. PASS Model. The Provenance-Aware Storage System (PASS) model is designed to represent provenance graphs generated by the PASS system. PASS is a storage system that observes systems calls, such as *read* and *write*, and infers dependencies between objects in the storage system. For example, when a process invokes a *read* system call on a file, PASS constructs a provenance edge that indicates the process depends on the file. A node in a PASS graph generally represents a file, a process, or a pipe. PASS also allows applications to integrate application-specific provenance, in which case, the nodes can be objects defined by an application. Each node can have attributes that describe its identity information. For a file, PASS records the type (i.e., FILE), file name, inode number, and assigns a unique ID to it. For a process, PASS records its type (i.e., PROC), PID, name, command-line arguments, environment variables, and execution time in addition to assigning a unique ID for the process. PASS also versions the nodes, and each node in the provenance graph represents a version of a file or process. Figure 1(a) shows how the scenario "Copying file A to file B using Process P" is represented by the PASS provenance model. The process P first reads information from file A, and this process creates a directed edge from A to P. P then writes the information to file B, which process, again, creates a directed edge from P to B. In this process, A is the ancestor of P, which is the ancestor of B.

2.1.2. Open Provenance Model. The Open Provenance Model (also referred to as OPM) aims to address the issue of interoperability across various provenance systems. A recent version, opm-v1.1 [Moreau et al. 2011], was released in July, 2010.

The OPM model defines three kinds of nodes, namely artifact, process, and agent. Artifact represents a stable state of a physical object or a digital object. Process indicates the operation on an artifact, and agent enables or facilitates the execution of a process. These three kinds of nodes are connected by five kinds of edges, namely, *used(R)*,

wasGeneratedBy(R), *wasControlledBy(R)*, *wasTriggeredBy*, and *wasDerivedFrom*. The Roles that are denoted by letter "R" in *used(R)*, *wasGeneratedBy(R)*, and *wasControlledBy(R)* are mandatory, and they express application-specific relationships between different nodes (e.g., we use *wasGeneratedBy(out)* to represent the relationship that an artifact is an output file of a process). Further, a node can optionally have a set of property-value pairs to describe its identity information, and the edge can be (though not mandatory) annotated with the time that indicates when a specific event (or causality relationship indicated by this edge) happens. Additionally, each node or edge belongs to an account which indicates a specific provenance layer. Figure 1(b) shows how the OPM model represents the copy operation "Copying file A to file B using Process P". Note that this model uses a different edge convention from the PASS model. In addition, this model has its own definition on node and edge types, providing very rich information to represent the provenance graphs.

2.1.3. Factorization And Inheritance (FAI) Model. In this model [Chapman et al. 2008], the provenance of a data item is called a provenance record. It comprises a set of provenance nodes, each of which consists of a manipulation (or workflow engine) and an input or arguments to the manipulation. For each data item, some of its identity information (e.g., name and ID) are also represented as data items and can share the same provenance with the data item they describe. For example, in Figure 1(c), the data item "File" contains a subitem "name", and the provenance of "name" is the same as the provenance of "File". This structure inherently has duplicates that unnecessarily take up too much space. Additionally, the ancestor relationship between different nodes is not as clear as in the previous two models.

2.2. Factorization And Inheritance (FAI) Compression Scheme

The FAI compression scheme is composed of a series of factorization and inheritance methods that compress provenance based on the FAI model. These methods are as follows.

Basic Factorization. If the provenance records of two data items are completely the same, one of them can be deleted.

Node Factorization. Node factorization finds common provenance nodes between different provenance records and stores only one copy of those nodes in the provenance store.

Argument Factorization. Argument factorization removes the common provenance node components (e.g., a manipulation, an input, or argument) between different provenance nodes.

Structural Inheritance. If two data items have a parent-child relationship and the child data item has the same provenance as the parent data item, then the provenance of this child data item needs not to be recorded. For example, the provenance of "name" in Figure 1(c) can be omitted.

Predicate Inheritance. This algorithm first partitions a dataset according to a set of user-defined boolean predicates, such as, is this data item a molecular? Then it finds the common provenance components between the provenance of data items that belong to the same predicate or category and stores only one copy in the provenance store.

Note that any of the factorization methods just describe can be combined with one or more inheritance methods, and the two inheritance methods can also be combined to achieve a better compression ratio on the provenance dataset. However, FAI does not exploit the characteristics of provenance graphs and, as we will see, is not fine-grained enough to exploit duplicates existing in the provenance graphs.

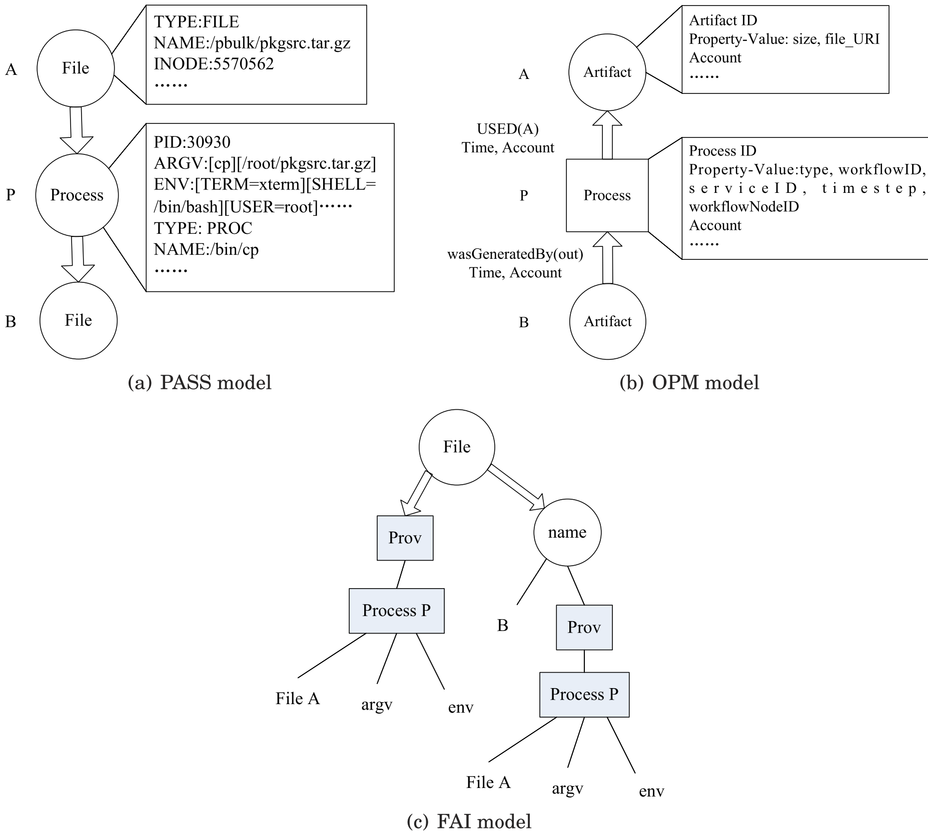


Fig. 1. Representation of the scenario "Copying file A to file B using Process P" by three different provenance models.

3. CHARACTERISTICS OF PROVENANCE GRAPHS

In this section, we outline five characteristics of provenance graphs that differentiate it from other metadata and motivate our research. These characteristics range from the composition of provenance graphs to the way provenance is used.

3.1. Data Composition of Provenance Graphs

A provenance graph is composed of two distinct parts, that is, identity information on provenance nodes and ancestor information on the provenance edge. We investigated their distribution in a large variety of provenance traces. Table I shows the distribution of these two data types of traces generated using various workflows. The traces represent a wide range of provenance systems/models and workloads. For example, the NetBSD trace was collected by compiling all the components of a NetBSD system, while the elaine-oct25 trace records the behavior of a person who developed a Python application and wrote a conference paper.

The table shows that there is no particular pattern in the distribution. In some traces, the provenance is dominated by the identity information and in others it is dominated by ancestry information. The identity information typically dominates in traces from the PASS and Karma systems. For PASS traces, this is because they have recorded a large variety of attributes to describe the identity information for every process node, for example, process ID, file name, time, environment variables,

Table I. Breakdown for the Percentage of Size of Identity and Ancestor Information in Various Provenance Traces

Provenance Trace	Description	Provenance System	Provenance Model	Size of Identity %	Size of Ancestor %	Source
NetBSD	Build of several components of NetBSD	PASS	PASS	93.42%	6.58%	[PassTrace 2008]
elaine-oct25	A researcher developed a Python application and wrote a conference paper	PASS	PASS	71.6%	28.4%	[PassTrace 2008]
linux-apr13	Build of the Linux kernel	PASS	PASS	72.14%	27.86%	[PassTrace 2008]
patch-apr17	Patching the Linux kernel	PASS	PASS	73.70%	26.30%	[PassTrace 2008]
am-utils	Compilation of am-utils	PASS	PASS	82.74%	17.26%	[PassTrace 2008]
blast-lite	A simple instance of the Blast biological workload	PASS	PASS	79.61%	20.39%	[PassTrace 2008]
PostMark	The PostMark file system benchmark	PASS	PASS	73.54%	26.46%	[PassTrace 2008]
NAM-WRF	Weather and Ocean Modeling	Karma	OPM	68.12%	31.88%	[Cheah et al. 2011]
NCFS	Weather and Ocean Modeling	Karma	OPM	60.58%	39.42%	[Cheah et al. 2011]
SCOOP	Weather and Ocean Modeling	Karma	OPM	71.15%	28.85%	[Cheah et al. 2011]
Gene2life	Bioinformatics and biomedical	Karma	OPM	70.43%	29.57%	[Cheah et al. 2011]
Motif	Bioinformatics and biomedical	Karma	OPM	54.08%	45.92%	[Cheah et al. 2011]
Animation	Computer animation rendering	Karma	OPM	59.21%	40.79%	[Cheah et al. 2011]
J062941	Managing large-scale, complex scientific data and metadata collections	Tupelo	OPM	33.49%	66.51%	[Challenge3 2009]
pc3opm	A set of assertions made by the services involved in a process	PASOA	OPM	33.21%	66.79%	[Challenge3 2009]
OPMGraph-complete	A sequence of steps with exit conditions	Taverna	OPM	33.92%	66.08%	[Challenge3 2009]

The table also provides a basic description of the traces and where they can be found.

and arguments. For Karma traces, the reason that identity information dominates is that they were collected based on the OPM model and recorded a sizeable amount of property values (though this is optional in OPM model) to describe the attributes (e.g., workflowID, serviceID, and timestep) in the identity information. Ancestry information takes up a larger percentage in the traces from Tupelo [Futrelle et al. 2009], PASOA [Groth et al. 2006], and Taverna [Missier et al. 2010]. These traces were generated based on the OPM model with ancestor information, such as role, time, and account, on edges, but without the optional property-value pairs in identity information.

3.2. Duplication in Provenance Graphs

As we saw in the previous section, identity information takes up most of the space in a PASS trace. Our further analysis shows that duplicates are common among these identity information. For instance, as many environment variables and arguments (e.g., USER=root, SHELL=/bin/bash) are shared and used by many processes, they appear in the identity information of many process nodes. According to the experimental results on NetBSD trace in Section 5, these duplicates can take up 76.02% in size of the original identity information.

In many OPM traces, it is also a very common case that a large amount of duplication exists in the provenance graphs. First, duplicates exist frequently in the annotation information in the provenance node information (e.g., type and workflowID in the property-value pairs) and edge information (e.g., time) in some OPM traces.

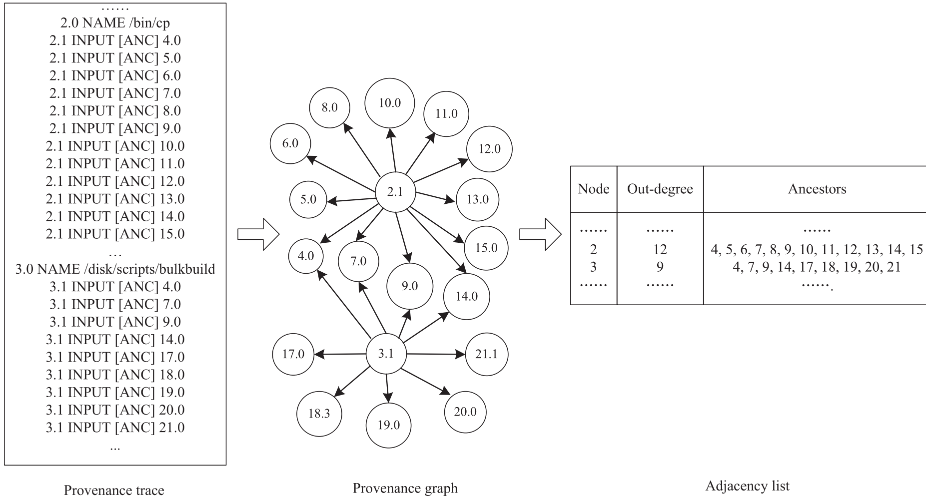


Fig. 2. Mapping from the provenance trace to adjacency list that represents the provenance graph. The expression "`2.1 INPUT[ANC] 4.0`" indicates that node 4 is an ancestor of node 2, resulting in a directed edge from node 2 pointing to node 4. This figure also shows that provenance graph exhibits the similar characteristics (i.e., similarity and locality) as Web graph. Note that though the versions of the nodes are not shown in the adjacency list, they are processed in the compression algorithm as shown in Section 4.1.

For instance, in the Karma trace, the type for every process node is "service" and the workflowID is also the same in the identity information of every node for an individual workflow graph. Second, the accounts, which are the same for every node and edge in an individual workflow, are recorded for every node and edge in an OPM trace. The experimental results on those Karma traces in Section 6 show that duplication can take up 34.86% to 37.50% in the identity information (or node information), and 38.83% to 40.26% in the ancestor information (or edge information) of six workflow graphs in the Karma traces.

3.3. Similarity to Web Graphs

A web graph has a node for each URL and an edge for each hyperlink from one Web page pointing to another. Existing Web graph compression algorithms [Boldi and Vigna 2004a] typically exploit the following two key properties to significantly compress Web graphs.

— *Locality.* Many links are within a URL domain, and therefore are not likely to point to pages far away.
— *Similarity.* Adjacent Web pages have a high probability to have a common set of neighbors.

Provenance graphs also have a similar organizational structure as Web graphs. Figure 2 shows the conversion from a snapshot of a NetBSD provenance trace (generated by the PASS system) to an adjacency list that represents the provenance graph. The notation "`A INPUT[ANC] B`" in the provenance trace means that B is an ancestor of A, indicating that there exists a directed edge from A pointing to B. In this way, a provenance graph is also a directed graph and each node (e.g., node 2 or 3) has a series of out-neighbors.

Provenance nodes 2 and 3 are considered similar here as they have common ancestors nodes 4, 7, 9, and 14. The reason for this is that many header files or library

Table II. Statistics of Provenance Nodes on Locality, Similarity, and Consecutiveness

Provenance Trace	Number of nodes in total	Number of local nodes	percentage of local nodes	Number of similar nodes	percentage of similar nodes
PostMark	2678	2321	86.67%	2302	85.96%
Am-utils	46097	14788	32.08%	12673	27.49%
Blast-lite	238	42	17.65%	50	21.01%

Table III. Statistics of Interspersed Provenance Records

Provenance Trace	Number of records in total	Number of records interspersed	Interspersed Percentage
Patch-apr17	303972	57326	18.86%
Postmark	18011	4386	24.35%
Am-utils	379518	31418	8.28%
Blast-lite	1323	28	2.12%

files that are represented by nodes like 4, 7, 9, and 14 are repeatedly used as input by many processes (e.g., nodes 2 and 3). Nodes 2 and 3 also exhibit locality. The ancestors of provenance node 2 are only between 4 and 15, and the ancestors of node 3 are only between 4 and 21. This is because many header files or library files (e.g., the ancestors of nodes 2 and 3) that are used as input by a process are probably in the same directory, so the ordering (and therefore assigned ID numbers) of these files are usually close to each other.

We have further investigated the number of provenance nodes that have these two properties in some provenance traces as shown in Table II. We specify a node has the locality property (i.e., "Local node") if the biggest ancestor of this node minus the smallest ancestor is not beyond 10. A node has the similarity property (i.e., "Similar node") if it has at least one common ancestor with at least one node in the preceding 10 nodes. One can see that the PostMark trace has the biggest percentage of similar nodes and local nodes in these traces. The reason is that the postmark process generates a lot of small files, which all have the postmark process as their common ancestor, thus the provenance nodes that represent these files are mostly similar nodes. Further, as many of these nodes have the postmark process as their only ancestor, these nodes also exhibit locality.

3.4. Provenance Changes with an Object's Evolution

The provenance of an object changes every time the object is modified. Given that there are many processes executing in a system and that they all generate or modify data, provenance will be concurrently generated into one single provenance stream. Hence, an object's provenance is usually interspersed with provenance of other objects in the provenance stream. Table III shows the statistical results about interspersed provenance records in some of the PASS traces. One can see that in some of the provenance traces (e.g., PostMark), provenance generation is more out-of-order (i.e., more provenance records are interspersed). This is because a large number of processes concurrently operate data. For instance, there can exist hundreds of transactions (e.g., read and write) per second in the PostMark benchmark, whereas other provenance traces (e.g., blast-lite) generate provenance in a more sequential way as only a few processes are concurrently executing. However, this "interspersed" phenomena exists in all traces. This indicates that provenance is a dynamic stream, but not static data, and we need algorithms that incrementally compress dynamic provenance data as opposed to algorithms that will only optimize global static data.

3.5. Provenance Needs to be Queriable

For provenance to be useful, it needs to be queriable even when it is compressed. So a compression scheme that makes it not amenable to querying or has too much query overhead is not preferred.

4. COMPRESSION ALGORITHMS

In this section, we describe the compression algorithms that we evaluate in this article. These compression algorithms are based on Web compression and dictionary encoding. These algorithms take advantage of the properties of provenance graphs.

4.1. Web Compression Algorithms

Two critical ideas lie behind Web compression algorithms [Boldi and Vigna 2004a]: first, encoding the ancestor list of one node by using the similar ancestors of another node as a reference, thus efficiently avoiding encoding the duplicate data; second, delta encoding [Witten et al. 1999], that is, encoding the gaps between the ancestors of a node rather than the ancestors themselves, which typically requires more bits to be encoded. Note that for consecutive numbers in the ancestors list, Web compression algorithms employ RLE (Run Length Encoding) technology to encode them by only recording the start number and length, reducing the storage space needed.

As we can see in Figure 2, a provenance graph can be represented by a set of provenance nodes which have a series of ancestors. The provenance nodes are numbered from 1 to N, in order, during provenance generation. We use Out(x) to denote the ancestor list of node x. The Web compression algorithm to encode this list is detailed as follows.

(1) *Reference Compression*. Find the node with the most similar ancestor list in the preceding W ancestor lists. W is a window parameter. Let y be such a reference node, $x - y$ is the reference number and Out(y) (ancestor list of y) is the reference list. The encoding of Out(x) can be divided into three parts: the reference number $x - y$, a bit list to identify the common ancestors between Out(x) and Out(y), and *Extra nodes* that identifies the rest of the ancestors in Out(x).
(2) *Run Length Encoding*. Separate the consecutive numbers from the *Extra nodes*, and then represent each set of consecutive numbers by using its left extreme and the length.
(3) *Delta Encoding*. Let $x_1, x_2, x_3, ..., x_k$ be the residual nodes from the preceding steps. If $x_1 \leq x_2 \leq ... \leq x_k$, then encode them as $x_1 - x$, $x_2 - x_1$, ..., $x_k - x_{k-1}$.

Algorithm 1 contains the pseudocode. The algorithm makes one pass over the whole dataset, and encodes the dataset node by node. For each node, the algorithm encodes its ancestor list as the three steps detailed before. For *Reference Compression*, the algorithm finds the reference list in the preceding W ancestor lists. Note that if the node identifier (such as x) does not exceed W, there are only $x - 1$ ancestor lists to be chosen from. On the other hand, if we want to query the ancestor list (Out(x)) of node x, we have to first decode its reference list Out(y), and then we have to decode the reference list of Out(y), and so on. This would form a reference list chain, with a long chain obviously resulting in bad query performance. We confine the length of the chain to a maximum value of L, and only ancestor list of the node of which the chain length does not exceed L can be chosen as reference list. Once we identify the reference list, we use a bit list B to indicate the common nodes between $Out(x)$ and its reference list $Out(y)$ (i.e., $Out(x - r)$). For *Run Length Encoding*, it is necessary to sort ancestor list first, and then if there exists one or more sequences of consecutive numbers, we should use *left extreme* and *length* to encode them. At the last step, the

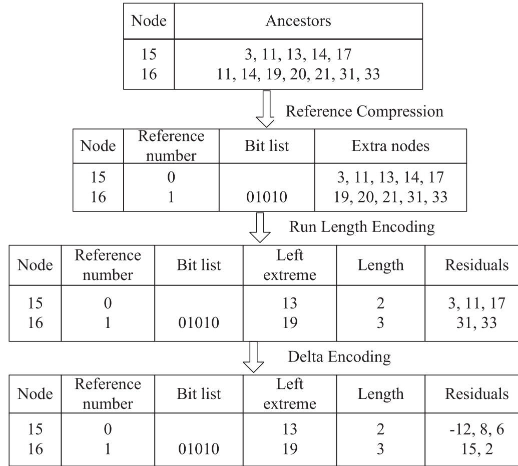
ALGORITHM 1: Web compression scheme

Input: Provenance Dataset D. For each node x in D, we use $Out(x)$ to represent its ancestor list. x refers to the sequential ID of the provenance node.
Input: Window parameter W
Input: Chain Length L
Output: encoded dataset D using web compression algorithm

1: **for each** node x in dataset D **do**
2:   **if** $x \leq W$ **then**
3:     $W = x - 1$
4:   **else**
5:     reset W to its initial input value
6:   **end if**
7:   y=NULL
8:   **for** $k = x - W$ to $x - 1$ **do**
9:     **if** (the number of common nodes between $Out(x)$ and $Out(k)$) > (the number of common nodes between $Out(x)$ and $Out(y)$) && the chain length of k does not exceed L **then**
10:       $y = k$
11:     **end if**
12:   **end for**
13:   $r = x - y$ /*$Out(y)$ is the reference list of $Out(x)$*/
14:   the chain length of x=(the chain length of y)+1
15:   **for** $i = 1$ to j /*the number of ancestors in $Out(x - r)$*/ **do**
16:     **if** the i-th item in $Out(x - r)$ also appears in $Out(x)$ **then**
17:       B(i-1)=1
18:     **else**
19:       B(i-1)=0
20:     **end if**
21:   **end for**/*The end of the first step, $Out(x)$ is encoded as r, B and the extra nodes (i.e., the nodes in $Out(x)$ that do not appear in $Out(x - r)$).*/
22:   **for each** sequence of consecutive numbers $(i, i + 1, ..., i + k)$ in extra nodes **do**
23:     encode the sequence as left extreme i and length $k + 1$.
24:   **end for**/*The end of the second step (Run Length Encoding).*/
    /*Let the ancestors of x yet to be encoded after the above step be $x_1, x_2, x_3, ..., x_k$.*/
25:   **if** $x_1 \leq x_2 \leq ... \leq x_k$ **then**
26:     we encode gaps $x_1 - x$, $x_2 - x_1$, ..., $x_k - x_{k-1}$.
27:   **end if**/*The end of the third step (Delta Encoding)*/
28: **end for**

algorithm employs a technology similar to "delta encoding" to encode the difference between adjacent numbers in a list instead of the numbers themselves.

Assume the number of nodes in the provenance set is N, the average number of ancestors of $Out(x - r)$ is j, and the average number of sequences of consecutive numbers in the ancestors is p, the algorithm runs in time $O(N * (W + j + p))$. The main data structure used are two arrays, one storing the ancestor list of the node to be encoded, the other storing the ancestor lists of W previous nodes. Assume the average number of the nodes in an ancestor list is n and the average size of a provenance node is s, the space used is $n * s * (W + 1)$. Note that the value W can significantly impact compression performance since it decides the scope of the possible reference lists for each provenance node. The value L can significantly impact query performance as it specifies the maximum depth of the reference list chain to be decoded for a query. We make a quantitative analysis of their impact on the compression and query performance in Section 7.2. The final compression result for the ancestor list of a node is composed of the reference number r, a bit list B, an array of left extremes and lengths, and the gaps. We then use variable-length code such as ζ_3 [Boldi and Vigna 2004b] to encode



Node	Ancestors
15	3, 11, 13, 14, 17
16	11, 14, 19, 20, 21, 31, 33

Reference Compression

Node	Reference number	Bit list	Extra nodes
15	0		3, 11, 13, 14, 17
16	1	01010	19, 20, 21, 31, 33

Run Length Encoding

Node	Reference number	Bit list	Left extreme	Length	Residuals
15	0		13	2	3, 11, 17
16	1	01010	19	3	31, 33

Delta Encoding

Node	Reference number	Bit list	Left extreme	Length	Residuals
15	0		13	2	-12, 8, 6
16	1	01010	19	3	15, 2

Fig. 3. An example on Web compression algorithm.

them (except the bit list) as 0/1 sequence. Note that for nodes that have versions in provenance traces such as PASS, we do not use a Web algorithm to encode the version numbers of all the ancestors of a node, instead we use a separate array to store them and index them using the node number when they are loaded into database.

Figure 3 shows a sample run of the three steps of the Web algorithm. In this example, node 15 is the reference for node 16 (reference number is $16 - 15 = 1$) and has no reference itself. The case in Figure 2 is simpler. The ancestor list of node 2 can be encoded only using *Run Length Encoding*. The ancestor list of node 3 can be encoded using the first two steps: *Reference Compression* and *Run Length Encoding*, with node 2 serving as the reference list.

4.2. Dictionary Encoding

Dictionary encoding (also called "enumerated storage" [Boncz 2002]) has been widely used in a variety of areas [Chen et al. 2001; Graefe and Shapiro 1991; Poss and Potapov 2003; Zukowski et al. 2006]. It scans the entire database or text files to find frequently occurring strings and then replaces them with integer codes. In this process, a mapping table from strings to integer codes is created and updated.

For provenance, when we load the provenance dataset into a database, we also look for frequently occurring strings in the provenance graph data, especially those that take up too much space, for example, the arguments and environment variables in the PASS system. Then we use integer codes to encode them and store this mapping relationship into a dictionary database. When we query these arguments or environment variables, we have to look up the corresponding entries in the dictionary database. We measure the overhead of this method on query performance in Sections 5 and 6.

Dictionary encoding is very efficient for eliminating the duplicates since the granularity on which it operates is flexible. It can be a whole string, the prefix of it, or an arbitrary substring in a big string. For example, an edge in an OPM graph is usually annotated with time which indicates when this process (or dependency relationship) happens and is usually in a format of standard coordinated Universal Time (i.e., UTC). This "Time" is usually represented using a string that consists of year, month, day, hour, minute, and seconds. In most cases, year, month, and day are the same for every edge in an OPM graph. So the repeated substrings that consist of year, month, and day can be selected and encoded.

Table IV. Database Schema for Web+Dictionary, FAI Techniques and Uncompressed Case

<table>
<tr><th>Compression techniques</th><th>Provenance composition</th><th>Databases</th><th>Provenance records</th></tr>
<tr><td rowspan="2">web+dictionary</td><td>identity</td><td>IdentityDB
DictionaryDB</td><td>(node ID, attribute, value)
(id, duplicate)</td></tr>
<tr><td>ancestor</td><td>AncestorDB
VersionDB</td><td>(node ID, ancestors coding)
(node ID, versions of ancestors)</td></tr>
<tr><td rowspan="2">FAI</td><td>identity</td><td>IdentityDB
DictionaryDB</td><td>(node ID, attribute, value)
(id, duplicate)</td></tr>
<tr><td>ancestor</td><td>AncestorDB</td><td>(node ID, ancestor)</td></tr>
<tr><td rowspan="2">original (uncompressed)</td><td>identity</td><td>IdentityDB</td><td>(node ID, attribute, value)</td></tr>
<tr><td>ancestor</td><td>AncestorDB</td><td>(node ID, ancestor)</td></tr>
</table>

Table V. FAI Techniques: Resulting Size (% of original)

FAI techniques	NetBSD	Linux-apr13
Basic Factorization	100%	100%
Node Factorization	100%	100%
Argument Factorization	37.43%	48.73%
Structural Inheritance	100%	100%
Predicate Inheritance	100%	100%

4.3. Combination of Web Compression Algorithms and Dictionary Encoding

As we have stated previously, Web compression algorithms can compress the ancestor information well and dictionary encoding can eliminate the duplicates existing in the provenance graph data. Their combination provides a practical and efficient method to compress a provenance graph. Additionally, since Web compression and dictionary encoding are both lightweight compression schemes, this method retains a good query performance on provenance datasets.

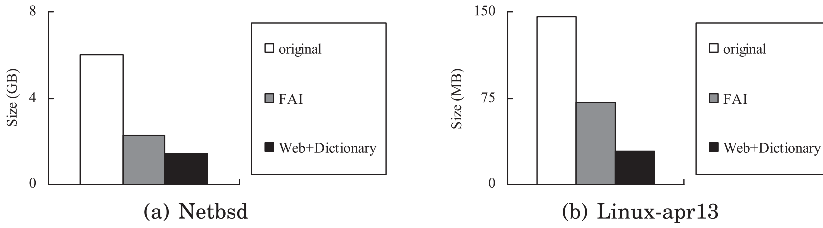
5. EVALUATION ON PASS TRACE

5.1. Experimental Setup

All experiments in this part were run on a Linux 2.6.34.6-54.fc13.i686 machine, with Pentium(R) dual-core E6500 2.93 GHz*2 CPU, 2GB memory, and one 500GB WDC WD5000AAKS-00A7B2 hard drive. We used two provenance traces in our experiment: NetBSD trace (6GB) and linux-apr13 trace (146MB). These two traces were acquired by collecting the provenance when compiling all the system components of a NetBSD system and the Linux kernel respectively. The details on these traces can be found in Table I.

We loaded these provenance traces into BerkeleyDB databases and compressed them using FAI and our hybrid method respectively. The DB cache used in our experiments is 16MB. Table IV shows the schema that we used for these two techniques and the uncompressed case.

In our hybrid approach, we stored the provenance in two primary databases: AncestorDB and IdentityDB. AncestorDB contains all the ancestor information of a provenance trace, and IdentityDB stores all the identity information of a provenance trace. We encoded the provenance using a series of compression technologies while storing the provenance to the database. For example, we used dictionary encoding to compress the identity information and Web compression algorithms to compress the ancestor information. For dictionary encoding, we used a database called DictionaryDB to store all the mapping relationships between the duplicate strings and their encoded integer codes. For the Web compression algorithm, we encoded the ancestors of a provenance



(a) Netbsd (b) Linux-apr13

Fig. 4. Compression size in NetBSD and Linux-apr13 trace.

node into 0/1 sequence. In addition, we used a database called VersionDB to store the versions of all the ancestors of a provenance node.

FAI is composed of a series of factorization and inheritance methods (see Table V). Argument factorization achieves the best compression ratio and is used for the rest of this part; it finds duplicate strings in IdentityDB, such as ARGV and ENV, which can be used as input parameters to a process, encodes them using integer codes, and then stores them in DictionaryDB. For other techniques, since no provenance records and no nodes (i.e., the combination of a process and its input as defined in the FAI model) are identical for different data items, the methods that find common nodes or even identical provenance graphs (such as basic factorization or node factorization) do not work. For predicate inheritance, we partition a PASS trace into three categories: File, Process, and Pipe. Since there exists no common provenance component for each category, predicate inheritance also does not work. Note that, though the provenance items such as "TYPE: FILE" and "TYPE: PROC" are common in the File set and Process set respectively, they are not treated as provenance components in the FAI model. This is because they cannot be used as the input parameters to a process like ARGV and ENV when a PASS trace is converted to FAI (see Figure 1). In addition, PASS traces do not display structural inheritance properties, so the structural inheritance method also does not work. Hence, in order to fairly represent the performance of FAI, we use FAI to represent argument factorization for compressing PASS traces.

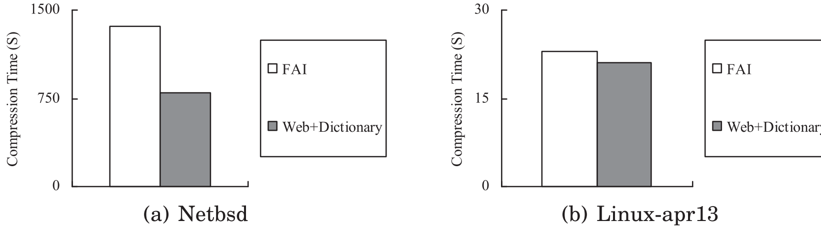
5.2. Compression Performance

Figure 4 shows the sizes of the two provenance traces compressed using the FAI and the Web+dictionary methods respectively. Web+dictionary reduces the data to 19.83%–23.40% of the original size, and to 40.69%–62.51% of the data size that FAI produces. The reason for this is that FAI can only eliminate the duplicate strings such as ARGV and ENV, which are used as input parameters to a process node, while dictionary encoding has a more flexible compression granularity. It can eliminate the common substrings in strings such as Name and Freezetime. For example, all the Names of the files in a directory have the same common prefix. In addition, Web compression makes a more in-depth compression by seeking the locality and similarity between the ancestors of different nodes, then uses the Web compression algorithm to encode all the ancestors that belong to a node to a 0/1 sequence. Table VI shows the compression size breakdown for the FAI and Web+dictionary techniques. Web+dictionary significantly outperforms FAI for 7.53%–16.4% on compressing identity information, 86.2%–95.4% on compressing ancestor information, and 37.49%–59.31% for the total information.

We then look at the compression time (see Figure 5). Note that since we compress the provenance when we store it into the databases, the compression time includes the time needed to perform the IO to store the provenance into the databases. Figure 5 shows that, though Web+dictionary significantly outperforms FAI on compression ratio, it does not pay for this on compression time. In contrast, it takes less time

Table VI. Compression Size Breakdown (in MB) for FAI and Web+Dictionary Techniques

Traces	FAI			web+dictionary		
	identity	ancestor	total	identity (improvement)	ancestor (improvement)	total (improvement)
NetBSD	1569	677	2246	1311(16.4%)	93.46(86.2%)	1404.46(37.5%)
Linux-apr13	29.20	41.94	71.14	27.00(7.53%)	1.95(95.4%)	28.95(59.3%)



(a) Netbsd (b) Linux-apr13

Fig. 5. Compression time in NetBSD and Linux-apr13 trace.

Table VII. Compression Time Breakdown (in seconds) for FAI and Web+Dictionary Techniques

Traces	FAI			web+dictionary		
	identity	ancestor	total	identity (improvement)	ancestor (improvement)	total (improvement)
NetBSD	1243.8	123.2	1367	462.49(62.8%)	330.6(−168.34%)	793.09(41.98%)
Linux-apr13	12.75	10.16	22.91	6.95(45.5%)	14.06(−38.39%)	21.01(8.29%)

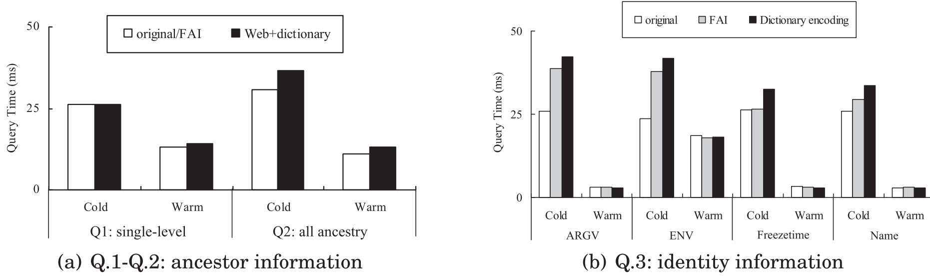
for compressing all the provenance sets when compared to FAI. Table VII further shows the compression time breakdown for these two techniques. Web+dictionary significantly outperforms FAI on compressing identity information. The reason for this is twofold. First, Web+dictionary eliminates more duplicate strings (not only ARGV and ENV, but also Name and Freezetime) than FAI in the identity information, making the size of the provenance records loaded to IdentityDB much smaller. In turn, this reduces the time needed to store the provenance records. On the other hand, FAI employs a hash table to store the duplicate strings and their frequencies. As the provenance sets are very big (146M and 6G respectively), the hash table can become very big, thus the time to query it during compression can be a big overhead. We also see that the Web compression algorithm can incur time overhead over the uncompressed case because of its three compression steps which consume lots of memory and CPU time. However, the total compression time is still much smaller than the FAI case.

5.3. Query Performance

To compare the compression methods on query performance, we ran the following three queries on the NetBSD trace.

(Q.1) Given a version of an object, retrieve all the objects on which it directly depends.
(Q.2) Find all the ancestry of a version of a specific object.
(Q.3) Retrieve the identity information of a specific object.

We chose these queries because they represent different query complexity. The queries also cover both the ancestor and identity information. The first involves a single-level ancestor query of a version of an object, which is similar to the adjacency query in the Web graphs. The second has to do the first query recursively until all the ancestors of a version of a specific object are found. The third query involves the



(a) Q.1-Q.2: ancestor information

(b) Q.3: identity information

Fig. 6. Query performance of the PASS traces on both cold cache and warm cache cases. Q.1 looks up the ancestors on which a version of a specific object directly depends. Q.2 looks up all the ancestry of a version of a specific object. Q.3 looks up the elements in identity information of IdentityDB. Each number in the figure is the average of at least 50 trials on a large number of different nodes.

retrieval of the identity information that describes a node, such as ARGV, ENV, Freezetime, Name, etc. We run experiments on both cold cache (i.e., reboot machine before each query) and warm cache.

Figure 6(a) shows the query performance of the Web scheme on ancestor information. In queries of both Q.1 and Q.2, we use the node identifier as keyword. For Q.1, after the ancestors's codes are retrieved from the AncestorDB, they have to be decoded before we can get the related ancestor IDs under the Web compression case. The versions of these ancestors can be retrieved from the VersionDB if needed, while in both original and FAI case, the ancestor IDs (including versions) can be directly retrieved from the AncestorDB. The query time under the Web scheme has a comparable performance with the original/FAI case on cold cache. On one hand, the Web scheme makes the ancestor records that need to be examined smaller than the original/FAI case. In turn, this reduces the query time. On the other hand, Web decompression during query incurs time overhead. Their combination makes the query performance comparable with the original/FAI case. For Q.2, as more decompression processes are needed, the time overhead becomes a little bigger, that is, 18.91% over the original/FAI case. For comparison, in the warm cache case, the query performance of both original/FAI and Web+dictionary case significantly improves over the cold cache case.

Figure 6(b) shows the query performance on each distinct element (e.g., ARGV and ENV) in the identity information of IdentityDB. For each query, we use the node number and the element that we want to retrieve in IdentityDB as keywords. For dictionary encoding, we first query the IdentityDB database to acquire the integer code of the element, then query the DictionaryDB to get the final strings of the element. From Figure 6(b), one can see that dictionary encoding results in an overhead of 9.27%–22.3% compared to the FAI case on cold cache. The reason for this is that, for elements such as Freezetime and Name, the query under dictionary encoding has to look up the dictionary entry in the DictionaryDB every time it gets a integer code from the IdentityDB. This “join” operation complicates the query and results in the overhead. Though for elements such as ARGV and ENV, in which cases FAI also has to look up the DictionaryDB, the DictionaryDB under dictionary encoding case incurs bigger overhead because the DictionaryDB in this case contains the duplication information of not only ARGV and ENV like in FAI case, but also Freezetime and Name. For the warm cache case, we see that the time needed for querying ENV is much longer than those of the others because the size of a large number of environment variables, such as PATH and USER, collected by PASS for each process object, is considerably larger than the other data and the cache size.

Table VIII. Overview of Various Workflow Characteristics in Karma Traces

Workflow Name	Scientific domain	Failure rate	Number of Manipulations in a workflow	Number of Provenance Graphs	Total Graphs Size (MB)
NAM-WRF	Weather and ocean modeling	22%	6	5990	160
NCFS	Weather and ocean modeling	27%	7	298	10.3
SCOOP	Weather and ocean modeling	21%	6	7987	208
Gene2life	Bioinformatics and biomedical	25%	8	7990	257
Motif	Bioinformatics and biomedical	50%	138	498	294
Animation	Computer animation rendering	47%	22	430	45.6

Table IX. Database Schema for Web+Dictionary, FAI Techniques and Uncompressed Case

Compression techniques	Provenance composition	Databases	Provenance records
web+dictionary	identity	ArtifactDB	(node string, account, size, fileURL)
		ProcessDB	(node string, account, workflowID, serviceID, timestep, workflowNodeID)
		DictionaryDB	(id, duplicate)
	ancestor	AncestorDB	(node ID, ancestors coding)
		NodeDB	(graphID, node ID, node string)
		TimeDB	(node string, times)
		RoleDB	(node string, roles)
		AccountDB	(node string, accounts)
FAI	identity	ArtifactDB	(node string, account, size, fileURL)
		ProcessDB	(node string, account, workflowID, serviceID, timestep, workflowNodeID)
	ancestor	AncestorDB	(node string/ID, ancestor node string/ID, role, account, time)
		NodeDB	(node ID, node string)
original (uncompressed)	identity	ArtifactDB	(node string, account, size, fileURL)
		ProcessDB	(node string, account, workflowID, serviceID, timestep, workflowNodeID)
	ancestor	AncestorDB	(node string, ancestor, role, account, time)

Table X. FAI Techniques: Resulting Size (% of original)

FAI techniques	NAM-WRF	NCFS	Animation	Gene2life	Motif	SCOOP	Total
Basic Factorization	100%	100%	100%	100%	100%	100%	100%
Node Factorization	100%	100%	100%	100%	100%	100%	100%
Argument Factorization	37.9%	41.7%	35.9%	45.0%	36.5%	42.8%	39.5%
Structural Inheritance	100%	100%	100%	100%	100%	100%	100%
Predicate Inheritance	78.75%	83.69%	81.80%	78.60%	83.33%	78.85%	80.31%
Argument Factorization and Predicate Inheritance	37.9%	41.7%	35.9%	45.0%	36.5%	42.8%	39.5%

6. EVALUATION ON OPM TRACE

6.1. Experimental Setup

We used the same machine as used with PASS traces. The provenance traces we used were extracted from a 10GB noisy provenance database [Cheah et al. 2011] generated by using the Karma [Cao et al. 2009] system. These traces consist of provenance generated from six kinds of workflows (see Table VIII). The workflows are from different scientific domains and accordingly have different characteristics. For example,

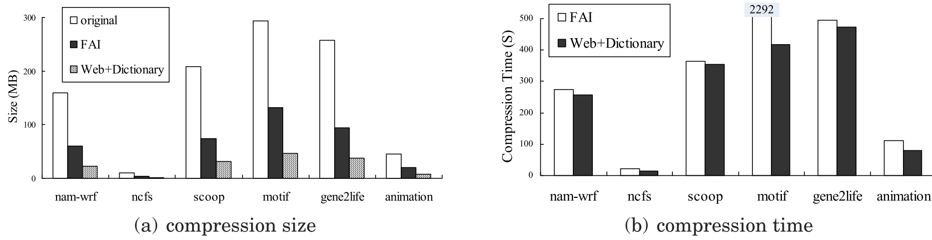
NAM-WRF, NCFS, and SCOOP workflows are used in the area of weather and ocean modeling, Gene2life and Motif workflows are used in the area of bioinformatics and biomedicine, and Animation workflow is used in the computer animation rendering field. Table VIII also shows the expected failure rates of those workflows (for example, due to intermediate data loss or system crash). Some workflows (such as Motif and Animation) have a high failure ratio, while other workflows (such as NAM-WRF and SCOOP) have a much lower failure ratio. The workflows also differ in the size. Some workflows can be as many as 138 manipulations in a run, while the smallest one has only 6 manipulations. The last two columns show the number and size of provenance graphs generated by each workflow. Some provenance graphs sets (such as Gene2life) are big and have more provenance graphs, while others (such as NCFS) are small and have much fewer provenance graphs.

We compressed these traces using FAI and our hybrid method respectively. In both cases, we stored the compressed provenance records in a Microsoft SQL Server 8.0 database. Table IX shows the schema that we used for these two techniques and uncompressed case.

In our hybrid approach, ArtifactDB and ProcessDB store the identity information of files and processes respectively. The DictionaryDB stores the mapping between frequently occurred strings and their encoded integers for duplicate strings that occur in identity information. For ancestor information, we assign each node string a unique identifier (in the order of appearance in the trace) to make the Web compression easier. For example, we encode the node string `Process_25367` as a node identifier (ID) 10. We store the mapping relationship from this identifier to the corresponding node string in NodeDB. Since a node can have a series of ancestors, we encode them into one ancestors coding using the Web compression algorithm and store each record in a concise format (i.e., (Node identifier, ancestors coding)) in AncestorDB. Note that we do not store all the ancestor information of a provenance trace into only one AncestorDB, but we instead split the AncestorDB into multiple small databases, each of which is responsible for storing only one provenance graph. We assign each AncestorDB a graphID. Without this, the node identifiers will be globally numbered and can become very large as some of our provenance traces have millions of provenance nodes. We also use TimeDB, RoleDB, and AccountDB to store the time, role, and account information, respectively. The duplicate strings in these fields are also stored in DictionaryDB.

The databases used by FAI are shown in Table IX. Note that FAI does not compress any duplicate strings in the identity information (i.e., ArtifactDB and ProcessDB). The AncestorDB stores the detailed information of every edge. FAI finds the duplicate node strings in AncestorDB and encodes them using integer codes, and then stores them into NodeDB.

For FAI, only argument factorization and predicate inheritance techniques can work on these OPM traces (see Table X). They both eliminate duplicate components, specifically the node strings of a file or a process repeatedly used on the edges in our traces. The difference between argument factorization and predicate inheritance is that argument factorization finds duplicates as long as the number of components exceeds a user-specified threshold. In our experiments, we set this threshold to 2 to achieve the maximum compression ratio. Predicate inheritance considers a component as a duplicate string only when the component appears commonly in the provenance of data items that belong to the same boolean predicate. So predicate inheritance only compresses a subset of data compressed by argument factorization. For instance, in the NAM-WRF trace, data items A, B, and C have a provenance process node that belongs to the same workflowNodeID 3D_Model_Data_Interpolator. If A and B are generated using exactly the same workflow schema, we put them into a set that has the same



(a) compression size (b) compression time

Fig. 7. Compression performance for various Karma workflow traces.

boolean predicate. C is put into another set that has a different boolean predicate. So we only have to keep one copy of the provenance node that belongs to the workflowNodeID 3D_Model_Data_Interpolator using argument factorization. But we have to keep one copy of it for A and B, and another copy for C using the predicate inheritance technique. Other FAI techniques, such as basic factorization or node factorization, like with the PASS traces, do not work well since no provenance record or node is identical for different data items. Hence, we use FAI to represent argument factorization for compressing OPM traces.

6.2. Compression Performance

Figure 7 shows the compression size and time for various workflow traces using FAI and the Web+dictionary methods respectively. Web+dictionary outperforms FAI in both cases. The reason for the improvement of compression size is that FAI can only eliminate those duplicate node strings in the ancestor information, while Web+dictionary seeks locality and similarity between the ancestors of different nodes and encodes all the ancestors that belong to a node to a 0/1 sequence. In addition, it reduces the duplicate strings in the annotation information for both ancestor and identity information. This exploits the redundancy in the provenance graph to the maximum extent possible.

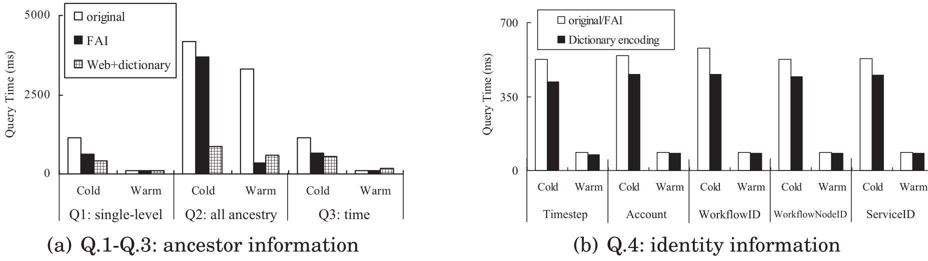
The reason for the improvement on compression time is twofold. First, Web+dictionary eliminates the duplicate strings in the identity information, making the size of the provenance records loaded to ArtifactDB and ProcessDB much smaller. In turn, this reduces the time needed to store the provenance records. Second, FAI employs a hash table to store the duplicate node strings and the frequency with which they appear in the edge (or ancestor) information. For provenance graphs that have a large number of nodes, such as motif, the time to query the hash table increases as the number of nodes increases. This indicates that FAI is not suitable for compressing large provenance sets.

6.3. Query Performance

To compare the query performance of compression methods, we ran a series of queries on the NAM-WRF trace as follows.

(Q.1) Given an object, retrieve all the objects on which it directly depends.
(Q.2) Find all the ancestry of a specific object.
(Q.3) Retrieve the time information of an edge.
(Q.4) Retrieve the identity information of a specific object.

Like in PASS trace, these queries represent different query complexity, and cover both the ancestor and identity information. The first and second queries involve a single-level and a full ancestry query respectively. The third and the fourth queries



(a) Q.1-Q.3: ancestor information

(b) Q.4: identity information

Fig. 8. Query performance for Karma traces on both cold cache and warm cache cases. Q.1 performs a single-level query of the ancestors upon which an object directly depends. Q.2 looks up all the ancestry of a specific object. Q.3 looks up the *time* information on the edges. Q.4 looks up the element in identity information of ProcessDB. Each number in the figure is the average of at least 50 trials on a large number of different nodes. Note that most of the OPM traces do not support version.

involve the retrieval of the annotation information of an edge and a node respectively. The results were also obtained with both cold cache and warm cache.

Q.1 and Q.2 perform similar queries as with those PASS traces by querying the AncestorDB (recursively for Q.2). The difference is that we do not have to query versions as Karma provenance nodes do not have versions. As shown in Figure 8, in both Q.1 and Q.2, Web compression performs better than FAI on cold cache. The reason for the improvement is twofold. First, Web compression significantly reduces the number of provenance records in AncestorDB. The query on the ancestor of a node will return a series of records in FAI, but only one record under Web compression. Second, the Web compression algorithm further reduces the size of each record by exploiting the similarity and locality in the ancestors, making the size of the records to be read much smaller than in the FAI case. Though Web decompression can incur time overhead during the ancestor queries, we have reduced this impact by confining the length of the provenance chain to 5 to avoid a decompression creep, while for warm cache case, the query time of the original (uncompressed) case for Q.2 is still high. The reason for this is that the number and size of provenance records in the uncompressed AncestorDB are both very large. So it is impossible for all the related data records during the recursive query to be kept in cache. We also see that Web+dictionary was outperformed by FAI on warm cache. This is due to the multiple decompression processes during recursive query that consume lots of CPU time.

In Q.3, the Web+dictionary encoding slightly outperforms FAI on cold cache. The improvement is because, for the Web+dictionary encoding case, the TimeDB stores all the time information of a node string (these time information are on the edges that start from this node string) in only one database record, while FAI uses one record for storing the time information on each edge in AncestorDB. So the number of the entries that need to be queried in TimeDB in Web+dictionary encoding is much fewer than in AncestorDB in FAI. On the other hand, the time information in TimeDB has been encoded using dictionary encoding, so the size of the record is much smaller than in the FAI case. Though querying the DictionaryDB incurs overhead, the total time in the Web+dictionary case is still smaller than in the FAI case. Similarly, in Q.4, the elements (such as *workflowID*) are encoded and stored using small integer numbers in ProcessDB with the dictionary encoding. Hence the query time of ProcessDB is much smaller than in the uncompressed case and FAI. Despite the cost of querying the DictionaryDB for the final strings, the total query performance with dictionary encoding is still better than the uncompressed and FAI case on cold cache. By contrast in warm cache case, the query times dramatically decrease and the dictionary encoding case gives comparable performance as the original/FAI case.

7. EVALUATION ON OTHER ASPECTS

7.1. Provenance Maintenance

We have discussed earlier about how to reduce the size of a static provenance set, and we now describe how to change/modify the compressed provenance in the database when the data item associated with the provenance is modified or deleted. Generally, updates to the provenance database can be classified into three cases: data creation, data modification, and data deletion.

ALGORITHM 2: Decode the encoded ancestor list of N to get $L0$. Function:Ancestor(N, ancestor list $L0$)

Input: N, the node ID of which the ancestor list to be decoded
Output: the decoded ancestor list L0 of N
1: decode the encoded ancestor list of N to get the reference number r
2: **if** $r = 0$ **then**
3: continue to decode the ancestor list of N to get the *left extreme*, *length* and *residuals*, then we can get the ancestor list $L0$.
4: **else**
5: Function:Ancestor($N - r$, ancestor list $L1$)
6: continue to decode the ancestor list of N to get the Bit list L, then we can get the common ancestors between $N - r$ and N.
7: continue to decode the ancestor list of N to get the *left extreme*, *length* and *residuals*, then we can get the ancestor list $L0$.
8: **end if**

ALGORITHM 3: Update provenance when inserting a new provenance record into AncestorDB

Input: $N \rightarrow P$, the new provenance record to be inserted
Output: a new AncestorDB: the ancestor list of $N + i$ $(0 \leq i \leq W)$ has been re-encoded
1: decode the encoded ancestor list of N to get $L0$
2: get the new ancestor list of N: $L1 = L0 + P$
3: re-encode the ancestor list of N using web compression scheme
4: store the re-encoded ancestor list of N into AncestorDB
5: **for** $n = N + 1$ to $N + W$ **do**
6: decode the ancestor list of n to get the reference number r
7: **if** $r = n - N$ **then**
8: re-encode the ancestor list of n using web compression scheme
/*N should still be chosen as the reference list of n*/
9: store the encoded ancestor list of n into AncestorDB
10: **end if**
11: **end for**

The first is the creation of a new data item. In this case, we have to insert the provenance (including identity information and ancestor information) associated with the data item into the databases. This involves four steps. First we encode the node string (e.g., `Process_25367`) that represents this data item into an identifier (e.g., 10) using dictionary encoding (this makes Web compression much easier), and store this mapping into DictionaryDB. Second we compress the ancestor list of this new node using the Web compression scheme and store it into AncestorDB. Then we encode the duplicate strings in the identity information of this node and store the mapping between strings and their IDs into DictionaryDB. At last, we store the encoded identity information into IdentityDB. Note that compressing the ancestor list of a new node

requires decoding the previous W ancestor lists first. For each decoding of the ancestor list, it involves a recursion search of the reference list until the reference number is 0 (see Algorithm 2). It runs in time $O(h)$, of which the h is the length of the reference list chain. So decoding W ancestor lists takes $O(W*h)$, since encoding the ancestor list of each node takes $O(W+j+p)$ (see Algorithm 1), the total runtime is $O(W+j+p)$ + $O(W*h) = O(W*h+j+p)$.

The second is the modification of a data item. For example, a process P modifies the content of a file N by invoking a write system call on it. In this case, a new provenance record $N \rightarrow P$ that indicates N depends on P (i.e., P is a new ancestor of N) is created. Note that N may already have some ancestors before this write process. For example, N may depend on many header files or library files when it was created. So we have to combine this new provenance record ($N \rightarrow P$) into the ancestor list of N. Algorithm 3 contains the pseudocode of this process. It mainly involves three steps. The first step is to decode the ancestor list of N, the second is to reencode the ancestor list of N, and the third step is to reencode the ancestor lists of $N+i$ ($i = 1, 2, ..., W$) if any of them took the ancestor list of N as reference list. Note that for the third step, $N+i$ still takes N as reference node because the latter still has the biggest number of common ancestors as the former.

Consider the runtime of Algorithm 3. The decoding of ancestor list of N takes time $O(h)$, of which the h is the length of the reference list chain. Reencoding the ancestor list of N needs to first decode the previous W ancestor lists, so it runs in time $O(W*h)$. The reencoding of the ancestor list of n (i.e., $N+r$) takes N as reference node and does not need to decode the previous W ancestor lists first, so it only takes time $O(1)$. Note that there can exist multiple n that takes N as reference node. So this step takes $O(m)$ if the number of this kind of n is m. So in total, Algorithm 3 takes $O(h)$ + $O(W*h)$ + $O(m) = O(W*h+m)$ in time complexity.

ALGORITHM 4: Update provenance when deleting the ancestor list of a node from database

Input: N, the identifier of the node to be deleted
Output: a new AncestorDB: 1. without the ancestor list of node N; 2. the ancestor lists of some nodes have been re-encoded

```
1: for n = 1 to N + W do
2:     decode the ancestor list of n to get the reference number r
3:     if r = n − N then
4:         re-encode the ancestor list of n using web compression scheme
           /*N should not be chosen as reference list again*/
5:         store the encoded ancestor list of n into AncestorDB
6:     end if
7: end for
8: delete the ancestor list of N from AncestorDB
```

The third is the deletion of a data item. If this data item has child A (i.e., this data item is the ancestor of A), then the provenance of this data item should not be deleted since its provenance is part of the provenance chain of its child. But if this data has no child, then we have to do the following three things: (1) Delete its provenance record in AncestorDB using the node identifier of the data item as keyword. (2) Delete the record of node in IdentityDB using its node string as keyword. (3) Delete the record of node in TimeDB using its node string as keyword. Deleting the ancestor list of a node in AncestorDB involves a complicated process since its ancestor list may have already been used as reference list by the next W ancestor list. Algorithm 4 contains the related pseudocode to deal with this. The algorithm checks the following W ancestor lists of

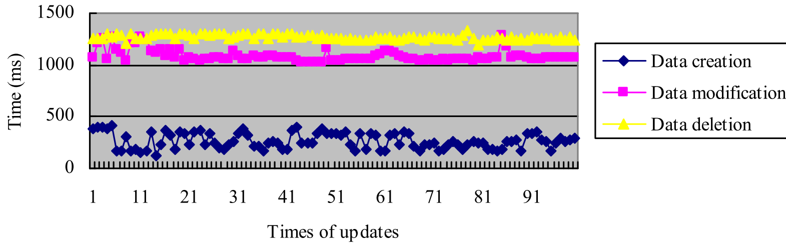


Fig. 9. Provenance update on databases where provenance is compressed by Web compression and dictionary encoding. There are three different types of provenance updates: data creation, data modification, and data deletion. All updates are performed on cold cache.

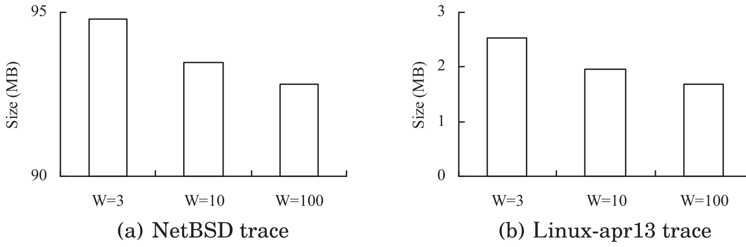
the deleted node N, and judges whether any of them took the ancestor list of N as reference list and reencodes them if needed.

For the runtime of this algorithm, if node n takes N as reference node, the reencoding of the ancestor list of n needs to first decode the ancestor list of itself and the previous W nodes (except N). Assume h is the length of the reference list chain for each ancestor list to be decoded, the reencoding of ancestor list of n takes time $O(W*h)$ + $O(W+j+p)$ = $O(W*h+j+p)$ (see Algorithm 1 for the definition of j and p). Let m be the number of this kind of node n, Algorithm 4 runs in time $O(m*(W*h+j+p))$.

We randomly generate a series of data creation, modification, and deletion operations and apply them to the compressed provenance database (Microsoft SQL server 8.0). Figure 9 shows the performance of provenance updates with respect to different types of operations. One can see that the performance of data creation outperforms both data modification and deletion, and the performance of data modification slightly outperforms data deletion. The reason for this is that data creation only needs to encode the ancestor list of the newly created node, while data modification involves a more complex process. It needs to not only encode the ancestor list of the modified node, but also needs to encode the ancestor lists of all the nodes that take the modified node as reference node. For data deletion, it also involves encoding multiple nodes that previously took the deleted node as reference node. Note that all these updates involve querying the databases and decoding the Web compressed provenance records which cost much time. However, as we see in Figure 9, the performance of data creation (an average of 300ms) even outperforms the query performance (around 450ms on nam-wrf trace for most of the queries on cold cache (see Figure 8)). In addition, most of the update time caused by data modification and deletion operations are still within the 1000–1300ms range, which is within the same order of magnitude of the query operations. This shows that updating a compressed database can generally be done efficiently.

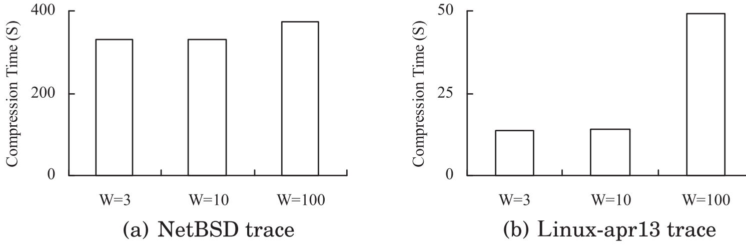
7.2. Sensitivity Analysis on Web Compression Algorithm

7.2.1. Window Parameter W. Note that for the Web algorithm, a larger W value would produce a better compression ratio because it enlarges the scope of the possible reference lists, but this would be at the expense of compression and decompression speed. We measure the compression performance with respect to W for both NetBSD and Linux-apr13 traces as shown in Figures 10 and 11. For compression size (see Figure 10), the performance of the Web compression algorithm increases as W increases. This is because a bigger window increases the likelihood of finding similar reference lists. As for compression time (see Figure 11), the case when $W = 100$ has the worst performance. This is because the Web scheme finds the reference list in the preceding W list,



(a) NetBSD trace (b) Linux-apr13 trace

Fig. 10. The performance of Web compression scheme on compression size with respect to different value of W. $L = 5$.



(a) NetBSD trace (b) Linux-apr13 trace

Fig. 11. The performance of Web compression scheme on compression time with respect to different value of W. $L = 5$.

Table XI. Statistic of Reference List Chain Length in NetBSD Trace

Chain Length	> 5	> 6	> 7	> 8	> 9	> 10
Number of Nodes	1330653	1294641	1262618	1232514	1204246	1177654
% of Total Nodes	24.2%	23.6%	23.0%	22.4%	21.9%	21.4%

and a large W imposes a high overhead on memory and CPU time. We use $W = 10$ to get a good trade-off for compressing provenance in our experiments.

7.2.2. Reference List Chain Length L. As we have discussed before, a long reference list chain can result in bad query performance. But how often is the reference list chain very long in practice? We analyze the distribution of chain length in the NetBSD trace as shown in Table XI. The chain length of 24.2% of total nodes is over 5, but there still exists 21.4% of total nodes where the chain length is over 10. We measure the query time with respect to the chain length as shown in Figure 12. It can be seen that the time increases nearly linearly as the length of chain increases. In our experiments, we tailor the algorithm to compress data such that the length of the reference list chain is automatically restricted to a maximum value of L. However, the value of L cannot be too small, because a small value of L can significantly impact the compression ratio (as shown in Figure 13). The reason for this is that it is probably with a small value of L wherein choosing a node as reference node will result in the chain length of the compressed node to exceed L even though this node has the most common nodes with the node that is compressed. In this case, the algorithm (i.e., Algorithm 1) automatically chooses another appropriate node as reference node. We choose $L = 5$ in our algorithm to achieve a best trade-off.

7.2.3. Breakdown of Performance on Similarity and Locality. In the Web algorithm, the first step (*Reference Compression*) encodes the ancestors of a node by exploiting the similarity between itself and other node with similar ancestors, and the second and third steps exploit the locality by encoding the gaps between ancestors rather than themselves.

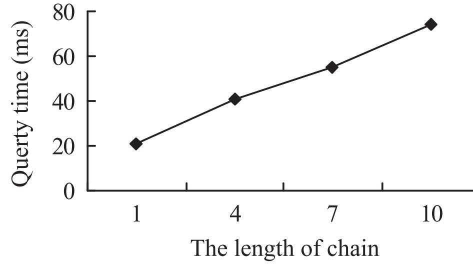


Fig. 12. Query time with respect to the reference list chain length in NetBSD trace. The numbers show the query results on single-level ancestry information. Each number in the figure is the average of at least 50 trials on a large number of different nodes. Further, the experiments were run on a cold cache.

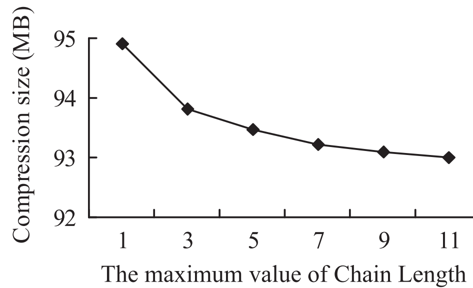


Fig. 13. Compression size with respect to the maximum value of reference list chain length L in NetBSD trace.

Figure 14 shows the breakdown of the Web algorithm performance on similarity and locality for ancestry information. One can see that, only exploiting similarity or locality can both significantly reduce the provenance size, resulting a provenance store of 6.1%–26.6%, and 5.7%–25.1% of the original size respectively. Then, the performance of exploiting locality typically outperforms the performance of exploiting similarity. This is because we can always encode the gaps as long as there exists ancestors, but exploitation of similarity requires that the previous W nodes have similar ancestors, which does not always exist. Third, the Web algorithm (exploiting both similarity and locality) only outperforms *similarity* 11.03%–21.69%, and outperforms *locality* 2.13%–15.95%. This indicates that a big number of nodes exploit both similarity and locality. Note that we do not further divide the performance of locality into the second step (*Run Length Encoding*, i.e., gap of 1) and third step (*Delta Encoding*) in the Web algorithm. This is because in some of the traces, consecutive numbers among ancestors do not always exist so frequently. So the numbers we show here are comprehensive results combining the second step with the third step.

7.3. Comparison with Other Compressors

Currently, the format of the provenance sets can be typically categorized into two types: XML and non-XML. The XML format is adopted in the most of the OPM traces. The non-XML format is adopted in the traces such as PASS. We apply XML compressors and some classical compressors to them respectively. As shown in Figures 15 and 16, some XML compressors, such as XMill [Liefke and Suciu 2000], or traditional compressors, like bzip2 and gzip, may produce a better compression ratio than our hybrid approach, render querying difficult or inefficient.

However, as they are well-known and popular tools, we still measure their query performance as shown in Figure 17. Note that in order to get an exact query result,

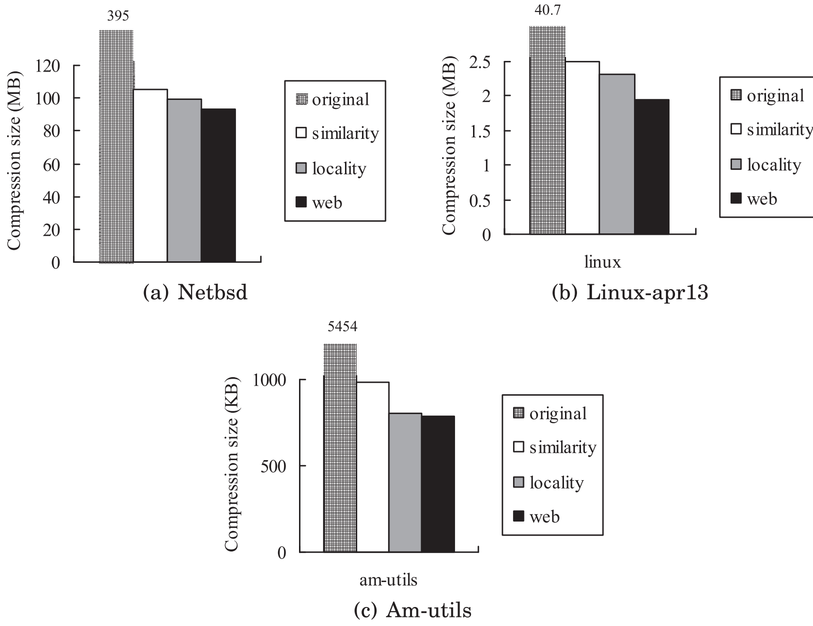


Fig. 14. Breakdown of Web algorithm performance on locality and similarity for ancestry information. *Similarity* indicates the Web algorithm that exploits only similarity (i.e., first step of Web algorithm), *Locality* represents the Web algorithm that utilizes only locality (i.e., the second and third steps of Web algorithm). The *original* size represents the total (i.e., uncompressed) size of the ancestry information.

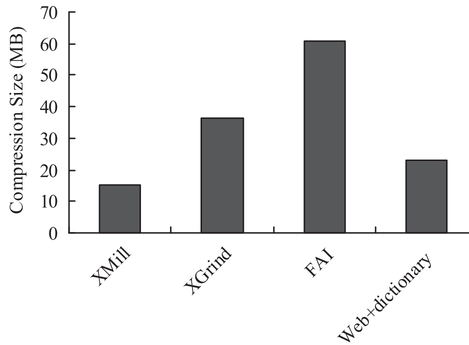


Fig. 15. Compression size of a variety of compressors on NAM-WRF provenance trace.

we do not apply them directly to the original provenance trace, but we first store the provenance trace into BerkeleyDB databases. Since BerkeleyDB stores the database as a file, we then compressed each such database using bzip2 (or gzip). As the provenance set (6GB NetBSD trace) we used is not small, we did not store the whole provenance set into a single database, but instead we split the provenance stream into multiple pieces and stored each of them into an individual database. We then built indices for these databases and compressed them using bzip2 (or gzip). One can see that from Figure 17, our compression method significantly outperforms the bzip2 and gzip on query performance. This is because we need to decompress the corresponding bzipped or gzipped database before retrieving the corresponding provenance information from the database. This consumes big time overhead. A take-away point here is that our

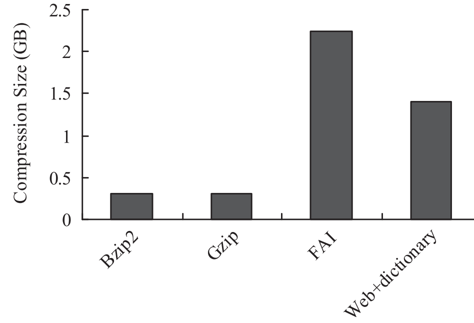


Fig. 16. Compression size of a variety of compressors on NetBSD provenance trace.

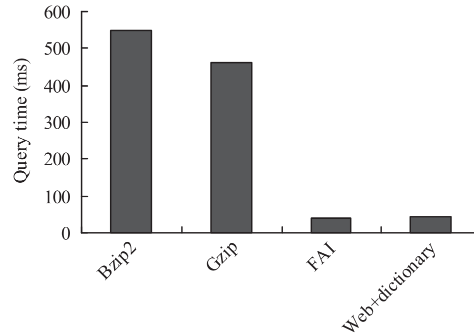


Fig. 17. Query performance of a variety of compressors on NetBSD provenance trace. The numbers show the query results on identity information (specifically, argument information). Each number in the figure is the average of at least 50 trials on a large number of different nodes. Further, the experiments were run on a cold cache.

compression method achieves the best trade-off between compression ratio and query performance.

7.4. Summary

Our compression method (Web+dictionary encoding) performs significantly better than the FAI algorithm on PASS and OPM traces on both compression ratio and compression time, and incurs only a small overhead over the FAI algorithm on PASS traces in terms of query performance. In addition, our approach has the best trade-off in terms of space and performance when compared to classical tools such as bzip2 and gzip.

8. RELATED WORK

Barga and Digiampietri [2007] presented four levels of provenance framework that can minimize the repeated dependency provenance information during provenance generation. In addition, they classified provenance into dependency provenance and annotation provenance based on a provenance query requirement. This is similar to our classification (identity information and ancestor information) based on the composition of a basic provenance graph. Thus the compression method we proposed can be exactly applied to their provenance store.

In prior work [Xie et al. 2011, 2012], we have presented the case for efficient provenance storage using the combination of Web compression and dictionary encoding. In this work, we present an in-depth analysis on the provenance characteristics and

a comprehensive evaluation across various provenance workloads and provenance models.

There has been considerable work [Adler and Mitzenmacher 2001; Boldi and Vigna 2004a; Randall et al. 2001; Suel and Yuan 2001] in the domain of Web graph compression. Adler and Mitzenmacher [2001] and Randall et al. [2001] both proposed to utilize reference compression to compress Web graphs. Randall et al. [2001] also suggested encoding the gaps between the numbers in the ancestor list instead of encoding the numbers themselves. The critical Web graph compression framework was presented by Boldi and Vigna [2004a]. They obtained good compression performance by fully exploiting the locality and similarity of Web pages. The Web algorithm we use is based on this framework.

The "Frame of Reference" [Goldstein et al. 1998] encodes a series of integer numbers by computing the offset from each of them to a reference number. This is similar to the reference compression in the Web scheme that encodes an ancestor list by using another ancestor list as reference list. The "Run Length Encoding" [Roth and Horn 1993] generally encodes a set of repeated characters by using a single character and the number of repeated characters, while the similar technology we use in the Web algorithm is another typical case, that is, encoding consecutive numbers by only recoding the starting number and the length of the consecutive numbers. The "Delta Encoding" technology [Witten et al. 1999] (sometimes called differential compression [Roth and Horn 1993]) encodes the difference between adjacent numbers in a list. These technologies work together to achieve a best compression ratio in the Web compression algorithm.

The "Pattern Substitution" scheme [Roth and Horn 1993] is similar to dictionary encoding. The difference is that Pattern Substitution can use any notation to represent a frequently occurring string, while dictionary encoding uses integer codes to replace the strings. Dictionary encoding is used more often and is the prevalent compression scheme in databases today.

There are also classical techniques like LZ-based compression algorithms [Ziv and Lempel 1977] (e.g., gzip). These techniques present an upper bound on the compression that is possible. However, since they do not preserve the structure of the data, the resulting compressed graph will not be amenable to querying.

Some of the XML compressors [Liefke and Suciu 2000], though they achieve a good compression ratio, also result in a provenance store that is not suitable for querying. Even those that support keyword and path query [Tolani and Haritsa 2002] cannot provide the rich query language that supports joins as in a traditional database. Comparatively, our compression methods can be applied to various trace formats, and provide both good compression ratio and good query performance.

9. CONCLUSIONS

Efficient provenance storage is an essential step towards the adoption of provenance. As provenance accumulates over time, it can occupy significant portions of storage. Today, users have two non-options: archive it in a nonqueriable format or discard the provenance. In this article, we have addressed this crucial issue and have provided users with a practical solution for storing provenance efficiently. First, we analyze provenance graphs collected from different provenance models and workloads. Based on the analysis, we identify general properties of provenance graphs that a practical and efficient compression scheme should consider when compressing a provenance graph. We then present a hybrid approach that combines Web graphs compression and a dictionary encoding scheme to efficiently compress provenance graphs. We contrast this approach with a set of commonly used compression algorithms on provenance traces from different provenance systems such as PASS and different

provenance models such as Open Provenance Model. Our findings indicate that the hybrid approach is the most efficient along all axes.

ACKNOWLEDGMENTS

The authors would like to thank the sponsors of the SSRC and CRIS, including the National Science Foundation, Los Almos National Laboratory, LSI, IBM Research, NetApp, Samsung Information Systems America, Seagate Technology, Northrop Grumman, Symantec, Hitachi, CITRIS, the Department of Energy Office of Science, the NASA Ames Research Center and Xyratex.

REFERENCES

Adler, M. and Mitzenmacher, M. 2001. Towards compressing web graphs. In *Proceedings of the IEEE Data Compression Conference*.

Barga, R. S. and Digiampietri, L. A. 2007. Automatic capture and efficient storage of escience experiment provenance. *Concur. Comput. Pract. Exper.* 1–10.

Boldi, P. and Vigna, S. 2004a. The webgraph framework I: Compression techniques. In *Proceedings of the 13th International World Wide Web Conference*.

Boldi, P. and Vigna, S. 2004b. The webgraph framework II: Codes for the world-wide web. In *Proceedings of the International Data Compression Conference*.

Boncz, P. A. 2002. Monet: A next generation dbms kernel for query-intensive application. Ph.D. thesis, Universiteit van Amsterdam, Amsterdam, The Netherlands. http://oai.cwi.nl/oai/asset/14832/14832A.pdf.

Bose, R. and Frew, J. 2004. Composing lineage metadata with xml for custom satellite-derived data products. In *Proceedings of the 16th International Conference on Scientific and Statistical Database Management*.

Cao, B., Plale, B., Subramanian, G., Robertson, E., and Simmhan, Y. 2009. Provenance information model of karma version 3. In *Proceedings of the 3rd IEEE International Workshop on Scientific Workflows (SWF'09)*.

Challenge3. 2009. The third provenance challenge. http://twiki.ipaw.info/bin/view/Challenge/ParticipatingTeams3.

Chapman, A. P., Jagadish, H. V., and Ramanan, P. 2008. Efficient provenance storage. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Cheah, Y.-W., Plale, B., Kendall-Morwick, J., Leake, D., and Ramakrishnan, L. 2011. A noisy 10GB provenance database. In *Proceedings of the 2nd International Workshop on Traceability and Compliance of Semi-Structured Processes, in conjunction with the 9th International Conference on Business Process Management*.

Chen, Z., Gehrke, J., and Korn, F. 2001. Query optimization in compressed database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 271–282.

Futrelle, J., Gaynor, J., Plutchak, J., Myers, J. D., McGrath, R. E., Bajcsy, P., Kastner, J., Kotwani, K., Lee, J. S., Marini, L., Kooper, R., Mclaren, T., and Liu, Y. 2009. Semantic middleware for e-science knowledge spaces. In *Proceedings of the 7th International Workshop on Middleware for Grids, Clouds and e-Science*.

Goldstein, J., Ramakrishnan, R., and Shaft, U. 1998. Compressing relations and indexes. In *Proceedings of the International Conference on Data Engineering*.

Graefe, G. and Shapiro, L. 1991. Data compression and database performance. In *Proceedings of the ACM/IEEE-CS Symposium on Applied Computing*. 22–27.

Groth, P., Miles, S., Fang, W., Wong, S. C., Zauner, K., and Moreau, L. 2005. Recording and using provenance in a protein compressibility experiment. In *Proceedings of the International ACM Symposium on High-Performance Parallel and Distributed Computing*.

Groth, P., Jiang, S., Miles, S., Munroe, S., Tan, V., Tsasakou, S., and Moreau, L. 2006. An architecture for provenance system. Tech. rep. http://eprints.soton.ac.uk/263196/1/provenanceArchitecture7.pdf.

Jayapandian, M., Chapman, A. P., Tarcea, V. G., Yu, C., Elkiss, A., Ianni, A., Liu, B., Nandi, A., Santos, C., Andrews, P., Athey, B., States, D., and Jagadish, H. V. 2007. Michigan molecular interactions (MiMI): Putting the jigsaw puzzle together. *Nucleic Acids Res. 35,* D566-571.

King, S. T. and Chen, P. M. 2003. Backtracking intrusions. In *Proceedings of the ACM Symposium on Operating Systems Principles*.

Liefke, H. and Suciu, D. 2000. XMill: An efficient compressor for xml data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Missier, P., Soiland-Reyes, S., Owen, S., Tan, W., Nenadic, A., Dunlop, I., Williams, A., Oinn, T., and Goble, C. 2010. Taverna, reloaded. In *Proceedings of the 22nd International Conference on Scientific and Statistical Database Nanagement (SSDBM'10).*

Moreau, L., Clifford, B., Freire, J., Futrelle, J., Gil, Y., Groth, P., Kwasnikowska, N., Miles, S., Missier, P., Myers, J., Plale, B., Simmhan, Y., Stephan, E., and van den Bussche, J. 2011. The open provenance model core specification (v1.1). *Future Gener. Comput. Syst. 27,* 6, 743–756.

Muniswamy-Reddy, K.-K., Holland, D. A., Braun, U., and Seltzer, M. I. 2006. Provenance-aware storage systems. In *Proceedings of the USENIX Annual Technical Conference*.

Muniswamy-Reddy, K.-K., Braun, U., Holland, D. A., Macko, P., Maclean, D., Margo, D., Seltzer, M. I., and Smogor, R. 2009. Layering in provenance systems. In *Proceedings of the USENIX Annual Technical Conference*.

Passtrace. 2008. http://www.eecs.harvard.edu/syrah/pass/traces/.

Poss, M. and Potapov, D. 2003. Data compression in oracle. In *Proceedings of the International Conference on Very Large Data Bases*.

Randall, K., Wickremesinghe, R., and Wiener, J. 2001. The link database: Fast access to graphs of the web. Res. rep. 175, Compaq Systems Research Center, Palo Alto, CA.

Roth, M. A. and Horn, S. J. V. 1993. Database compression. *SIGMOD Rec. 22,* 3, 31–39.

Shah, S., Soules, C. A. N., Ganger, G. R., and Noble, B. D. 2007. Using provenance to aid in personal file search. In *Proceedings of the USENIX Annual Technical Conference*.

Simmhan, Y. L., Plale, B., and Gannon, D. 2006. A framework for collecting provenance in data-centric scientific workflows. In *Proceedings of the IEEE International Conference on Web Services*.

Suel, T. and Yuan, J. 2001. Compressing the graph structure of the web. In *Proceedings of the IEEE Data Compression Conference*.

Tolani, P. M. and Haritsa, J. R. 2002. XGRIND: A query-friendly xml compressor. In *Proceedings of the International Conference on Data Engineering*. 225–234.

Vahdat, A. and Anderson, T. 1997. Transparent result caching. Tech. rep. CSD-97-974,8.

Widom, J. 2005. Trio: A system for integrated management of data, accuracy, and lineage. In *Proceedings of the International Conference on Innovation Data Systems Research (CIDR)*.

Witten, I., Moffat, A., and Bell, T. 1999. *Managing Gigabytes: Compressing and Indexing Documents and Images*. Morgan Kaufmann Publishing, San Francisco.

Xie, Y., K. Muniswamy-Reddy, K., Long, D. D. E., Amer, A., Feng, D., and Tan, Z. 2011. Compressing provenance graphs. In *Proceedings of the 3rd USENIX Workshop on the Theory and Practice of Provenance*.

Xie, Y., K. Muniswamy-Reddy, K., Feng, D., Yan, L., Long, D. D. E., Tan, Z., and Chen, L. 2012. A hybrid approach for efficient provenance storage. In *Proceedings of the 21st ACM International Conference on Information and Knowledge Management*.

Ziv, J. and Lempel, A. 1977. A universal algorithm for sequential data compression. *IEEE Trans. Inf. Theory 23,* 3, 337–343.

Zukowski, M., Heman, S., Nes, N., and Boncz, P. 2006. Super-scalar ram-cpu cache compression. In *Proceedings of the International Conference on Data Engineering*.

Received October 2012; revised April 2013; accepted June 2013