Steps Toward Managing Lineage Metadata in Grid Clusters

Ashish Gehani Minyoung Kim
*SRI International**

Jian Zhang
Louisiana State University

Abstract

The lineage of a piece of data is of utility to a wide range of domains. Several application-specific extensions have been built to facilitate tracking the origin of the output that the software produces. In the quest to provide such support to extant programs, efforts have been recently made to develop operating system functionality for auditing filesystem activity to infer lineage relationships. We report on our exploration of mechanisms to manage the lineage metadata in Grid clusters.

1 Introduction

Numerous domain-specific projects have been developed to record the provenance of data [3, 18, 12, 1]. However, these systems require applications to be customized to utilize the functionality provided for tracking lineage.

Operating system functionality to transparently audit provenance metadata was prototyped in the Lineage File System (LFS) [19]. LFS inserted *printk* statements in a Linux kernel to record process creation and destruction, operations to open, close, truncate, and link files, initial reads from and writes to files, and socket and pipe creation. The output was periodically transferred to a local SQL database. The Provenance-Aware Storage System (PASS) [17] audits a superset of the events monitored by LFS, incorporating a record of the software and hardware environments of executed processes. PASS is implemented as a layer in a stackable filesystem [25] and stores its records using an in-kernel port of Berkeley DB [13], providing tight integration between data and metadata. Both LFS and PASS are designed for use on a single node, although their designs can be extended to the file server paradigm by passing the provenance records (and queries about them) from the clients to the server in the same way that other metadata is transmitted (and utilized).

Grids comprise resources from a diverse group of universities, research laboratories, and companies. Each one facilitates computation that would be infeasible without the federation of the members' systems. Scientists, such as physicists, astronomers, and computational biologists, use large data sets drawn from around the world and generated over long spans of time. Many Grid applications are written without support for tracking provenance although its presence would provide substantial utility. We are exploring mechanisms for building software infrastructure that would add such functionality.

Since Grid resources are shared by multiple administrative domains, users may not be able to modify the kernels on nodes in the clusters where their applications are executing. However, the primary platform used by Grid computing projects is Linux. For example, TeraGrid [24] contains a large collection of Linux clusters. Since the Linux and Grid communities share common goals of open standards, software, and infrastructure, it is likely that Linux will remain the predominant operating system used by Grids. Linux kernels with version 2.6.14 and greater contain hooks to support user-space filesystems [5], allowing us to develop lineage auditing functionality for applications executing on a Grid cluster that the user can insert without modifying the underlying software infrastructure.

Researchers from the Globus and Condor projects have argued that the lack of transparent file access and the inability to use unmodified programs has hindered the adoption of Grid computing [15]. They are addressing the issue by building interposition agents to provide this facility. Our work is complementary to their effort and is based on the same assumptions. Current Grid provenance collection focuses on capturing application-specific workflows [20].

*This material is based upon work supported by the National Science Foundation under Grant OCI-0722068. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

Section 2 describes our lineage auditing prototype that runs on a single node. Section 3 motivates our investigation of different strategies for managing lineage metadata. Section 4 recounts our initial forays into the problem of how to store the metadata for use in a distributed environment. Section 5 outlines hybrid approaches that we designed for particular contexts. Section 6 describes an algorithm to optimally replicate the lineage records stored at different nodes, subject to communication and storage cost constraints, in the case where the query workload is not known ahead of time. Section 7 describes related research. We conclude in Section 8.

2 Single Node Prototype

Process Table	
Field	Type
LPID	int(11)
Host	varchar(256)
IP	char(16)
Time	datetime
PID	int(11)
PID_Name	varchar(256)
PPID	int(11)
PPID_Name	varchar(256)
UID	int(11)
UID_Name	char(32)
GID	int(11)
GID_Name	char(32)
CmdLine	varchar(256)
Environ	text

Table 1: A record is added to this table for each process.

File Table	
Field	Type
LFID	int(11)
FileName	varchar(256)
Time	datetime
NewTime	datetime
RdWt	int(11)
LPID	int(11)
Hash	varchar(256)
Signature	varchar(256)

Table 2: When a file is read or written a record is added to this table.

Our current prototype uses FUSE to intercede on *read()* and *write()* calls. Lineage records are stored in two MySQL [16] tables, shown in Table 1 and Table 2. When a read or write is intercepted, the calling process's details are extracted using the `/proc` interface. If the process has not read or written a file thus far, a new entry is created in the table of processes (shown in Table 1). The record is populated with an identifier that links records in the process and file tables (LPID), the hostname and IP address on which the process is running, the time the process was created, the process's name and PID, the parent process's name and PPID, the process owner's effective user name and UID, the process owner's effective group and GID, the command line used to invoke the process, and the environment variables and their values when the process was created.

The first time a file is read or written by a process, a record is added to the table of files (shown in Table 2). The record is populated with a record identifier, the filename, the last modification time of the file when the process first accesses it, the last time the current process modified the file if it has been opened for writing, whether the file was opened for writing, the identifier that allows linking to the appropriate record in the table of processes, a checksum to allow verification of the state of the file when it was opened, and a digital signature to attest the veracity of the lineage. Subsequent writes result in the `NewTime` field being updated. The field is used to invoke asynchronous callbacks that applications may have registered for so that they can be notified of changes to descendants or ancestors in their lineage tree.

3 Motivation

Current commodity filesystems only retain the metadata associated with the current state of a file. This requires a constant amount of storage (modulo implementations that allow arbitrary length extended attributes). Even experimental filesystems that retain a history of the metadata values only require storage (for most types of metadata) that grows linearly in the number of times a file is operated upon. In contrast, lineage metadata can grow exponentially since most outputs are produced by processes that utilize multiple inputs. There is a significant storage cost for replicating the metadata in a distributed environment. Simultaneously, the absence of the metadata at the local node degrades query performance since records must be retrieved over the network. These facts motivated our investigation of different strategies for managing lineage metadata.

4 Initial Approaches

A variety of applications operates on data in distributed environments, necessitating schemes to manage the associated metadata. We describe several approaches used in the past. Between them, they cover a large fraction of the strategies in use.

4.1 Auxiliary Files

A range of applications has adopted the strategy of using auxiliary files to store metadata. The conventions differ from Unix programs storing hidden files (prefixed with a period) in the user's home directory, to Linux window managers that generate thumbnails in the directory where the multimedia files reside, from storing preferences in system-wide and user-specific directories designated for the purpose, to files in *package* directories on Mac OS X (that supplant the forks of older Macintosh filesystems). The drawback of the strategy is that the filesystem is not aware of the application convention. Consequently, when a file is moved, the metadata is not transparently migrated along with the data, as illustrated in Figure 1.

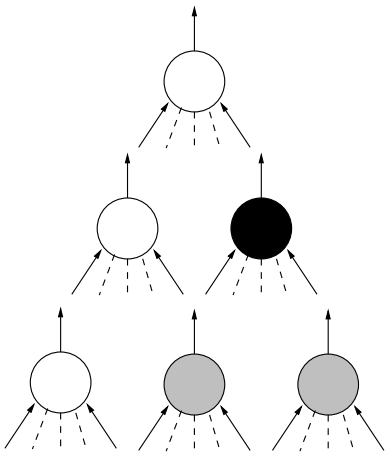

Figure 1: If lineage is stored in an auxiliary file, then if an operation is performed on a node that is not lineage-aware (shown in black), all metadata from preceding operations (marked in gray) is lost.

4.2 Filesystem Data Structure

It is possible to reduce the dependence on applications' management of metadata by migrating the auditing functionality to the filesystem layer. Since the FUSE [5] API supports extended attributes, we could store the lineage metadata in a data structure that resides directly in the filesystem. While this improves over the use of auxiliary files by allowing the filesystem to transparently migrate metadata along with data when a file is moved within the filesystem, the problem remains for the case where the file is moved to a remote node.

4.3 Local Database

The use of a local database to store metadata has a long history. Microsoft Windows incorporated a Registry two decades ago to allow applications to store key-value pairs. PASS [17] uses a version of Berkeley DB [13], and LFS [19] uses MySQL [16] to store the provenance metadata. This strategy addresses the issue of the rapid growth of the lineage metadata by opting only to retain it at the node where it was generated rather than propagating it along with a file. Effectively, it exploits the high level of redundancy in the lineage metadata of succeeding generations of data. As occurs in the cases where auxiliary files or filesystem data structures are used to store metadata, there is a loss of coupling between data and metadata when a file is moved to a different node.

4.4 File Server

A natural strategy to address the problem of migrating file access in distributed environments is to utilize a file server that stores the data and gate access to it through clients present on each node. However, this results in degraded performance. If the application used only a small subset of the data in the file, utilizing a file server would allow only the blocks of interest to be transmitted to the clients. Similarly, strong locality of reference would benefit from client caching, and the latency of operations on consecutive blocks could be reduced through precaching. In principle, it is possible to subordinate consistency to performance by weakening the synchronization between copies of the data. For example, Sun's Network File System [14] treats a client's copy of data as consistent for 3 seconds after retrieval from the server. However, this strategy is of limited utility for Grid computing workloads that may serially modify large fractions of a file that may be gigabytes or terabytes in size [21].

4.5 In-band Encoding

In-band encoding is traditionally used to add ownership metadata to multimedia content in the form of a watermark so that applications consuming the data can validate access rights. We designed an algorithm for in-band encoding of lineage metadata in video streams [8]. The embedded metadata persists despite being operated upon by applications that are not lineage-aware. However, the manipulations performed by such legacy software are not recorded, resulting in an incomplete record of the content's lineage. As with watermarks, the metadata must be embedded in a manner that does not degrade the user experience. Since lineage metadata is continuously growing, at some point it will exceed the inherent capacity of the channel in which the embedding is occurring. Further, this capacity is a function of the data and is difficult to predict.

Traditional in-band encoding is limited to lossy data formats, such as JPEG images, MPEG videos, and MP3 audio. However, analogous "in-band" encoding can be effected for lossless data formats that were designed to be extensible, such as XML. For example, lineage metadata can be inserted in an XML tree without affecting the semantics encoded by an earlier schema (using DTD [4], for instance). However, the same problem that occurs for lossy data recurs for extensible formats, which is the fact that manipulations of the data by applications that are not lineage-aware will result in incomplete lineage records.

The fact that in-band encoding is not agnostic to the data format means that it cannot be used for embedding metadata in arbitrary file content. Even when it can, the filesystem would need to be extended with functionality for reading and writing each in-band encoding format, with the accompanying implications for engineering effort required and robustness of the resulting system.

4.6 Headers and Footers

In practice, even lossy multimedia formats utilize standardized headers, such as JFIF for JPEG images, for storing metadata. This strategy is tenable in the case that the headers have a fixed length. When the metadata needs to be updated, the modification can be done in place. However, lineage metadata grows rapidly with the number of levels and fan-in of the lineage tree, as illustrated in Table 3. (We assume a minimal representation for a primitive operation, including a 32 bit process identifier and a 160 bit signature from the owner. Each input and output is assumed to be represented by a globally unique identifier consisting of a 32 bit IP address, a 32 bit filesystem identifier (such as an *inode*), and a 32 bit timestamp.)

Fan-in / Levels	1	2	3	4
2	0.09	0.14	0.19	0.24
3	0.14	0.34	0.65	1.05
4	0.19	0.75	2.02	4.30
5	0.24	1.56	6.13	17.30

Table 3: The space needed to represent a lineage tree depends on two factors. The first is the length of the sequence of reuse of the file by different programs. This corresponds to the number of levels in the tree. The second factor is the number of files that are used in the process of producing a single new version of a file. The values in the table are the number of kilobytes needed for a given average number of levels and fan-in.

Since the metadata is growing in size, if it is stored in a header, periodically it will exceed the space allocated for its use. In this case, the data can be shifted on disk to increase the allocation. However, the cost of doing this for large data files on the order of gigabytes or terabytes, as generated by Grid workloads, is untenable. Alternatively, the lineage metadata could be stored as a footer at the end of the data. However, as the lineage metadata grows in size, an analogous problem occurs, that is, each time the file's size must be increased, rewriting the lineage metadata introduces significant latency.

5 Hybrid Approaches

We experimented with a number of hybrid strategies to address the shortcomings of the initial approaches that we described above. We have not implemented the overloaded namespace but do have a partial realization of Bonsai and a complete prototype of the distributed database scheme.

5.1 Overloaded Namespace

In an effort to support extant applications that operate on data in local filesystems but also transfer files between networked nodes, we created the notion of an overloaded namespace. If an application accesses a file in the local filesystem's namespace, it will be operated upon with traditional POSIX semantics. However, if the file is opened in the overloaded namespace, it can only be read or written serially from beginning to end. When the file is read, the content returned will be the catenation of the data and the lineage metadata extracted from the local filesystem. When the file is written, after it is closed, the lineage metadata is extracted and stored in the local filesystem.

The behavior of the overloaded namespace allows legacy applications, such as *ftp*, *scp*, and web browsers, to transparently transfer lineage metadata along with the data by simply requesting files in the lineage-augmented namespace. The mechanism for implementing this functionality is now described. Using FUSE, we can modify the behavior of the *read()* and *close()* system calls.

Appending Lineage

When a *read()* is being performed on a file in the overloaded namespace, the file's lineage metadata is serialized and stored in a lineage buffer. After some (and possibly no) *read()*s of the file's content complete, a *read()* operation performed on the file will result in an **EOF**; that is, the end of the file has been reached. The unused portion of the read buffer is filled from the lineage buffer. After subsequent *read()* calls consume the rest of the lineage buffer, our code can insert the length and hash of the lineage metadata and generate an **EOF**. The calling application, such as *ftpd*, remains unaware that it has been provided with the catenation of the data, a serialized version of the lineage metadata, and a lineage size and hash.

Extracting Lineage

When a *close()* is performed on a file that had been opened for writing in the overloaded namespace, we anticipate that the file contains the data, a serialized version of the file's lineage metadata, and the lineage size and hash. Using the lineage size information, the lineage metadata and hash are extracted from the tail of the file. If the metadata and hash match, the file is truncated to its original size and the lineage metadata is deserialized and inserted in the local metadata store. The calling application, such as *ftp*, can remain unaware that it has provided the catenation of the data, a serialized version of the lineage metadata, and the lineage size.

As long as the overloaded namespace is used on both ends and the client and server applications only perform

sequential reads or writes from the beginning to the end of the file, the lineage metadata is transparently transferred along with the data.

5.2 Pruning Lineage Trees

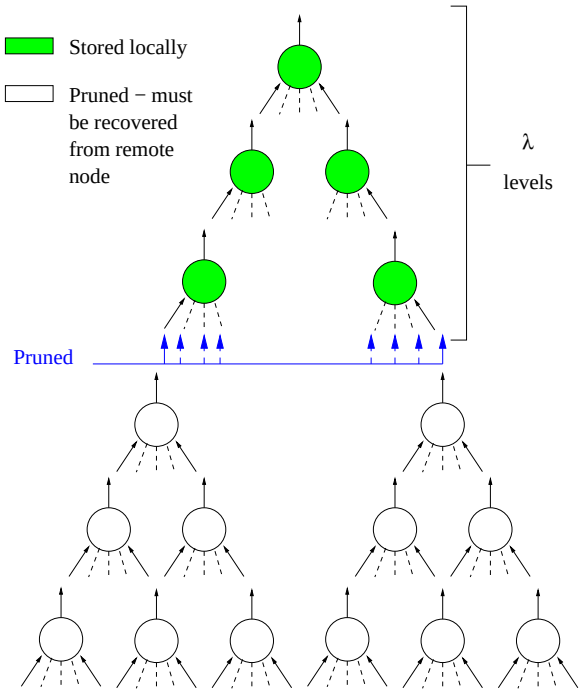

Figure 2: Bonsai [7] *prunes* out lineage graph information from nodes generated more than λ steps earlier. It replaces each vertex at level $(\lambda + 1)$ with a pointer to the node where that vertex was generated originally. During queries, this pointer is used to reconstruct the subtree rooted at the pruned node.

If the details of a primitive operation are stored only at the node where it was created, the metadata for the corresponding file must then include a pointer to the node from where the details can be retrieved. A *lineage daemon* must then run on each node and be able to service queries about such pointers. To reduce the likelihood of being unable to retrieve the details of a primitive operation when the original node is inaccessible, we could replicate the information at several other nodes. The probability of all relevant nodes being unreachable at the same time would drop rapidly as the number of replicas increases. Note that this is orthogonal to the question of how the metadata is propagated from one node to another.

Instead of replicating the lineage metadata corresponding to a primitive operation at the time it is generated, at which point it would have to be replicated at a random set of nodes, we delay the propagation until we know where it is likely to be used to answer queries. Such queries are bound to originate from nodes where data resides with a compound lineage tree that includes the primitive operation in question. Therefore, we are able to optimize our choice of where to replicate the metadata by propagating it along with the data, and simply pruning the lineage tree at the point where our replication goal has been met. In effect, if we wish to replicate a primitive operation's lineage metadata at λ other nodes, we can propagate it with the data when the next λ successive processes consume the data and its successive descendants. This is the essence of Bonsai [7], which is illustrated in Figure 2.

5.3 Distributed Database

The purpose of the lineage daemon was to store the metadata and make it available to remote nodes. Since distributed databases are designed for this purpose, we opted to use one as the substrate for managing the lineage metadata. Our single node prototype, described in Section 2, used MySQL schema to store the lineage. Hypertable [11] seemed to be a natural fit as it is a distributed database with schema and a query language HQL [10] similar to SQL. The issue of linking files when they crossed filesystems with disjoint namespaces was left unaddressed.

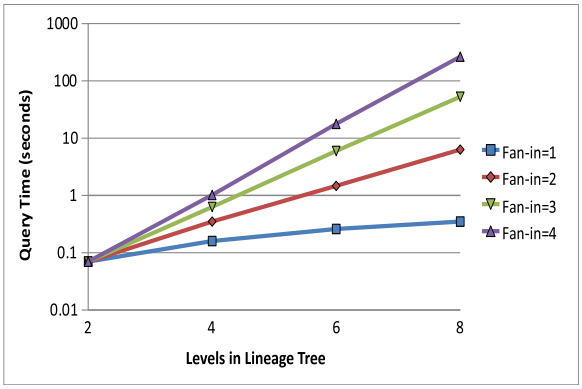

Figure 3: Response times from MySQL.

We constructed a synthetic workload that had files with a varying number of levels and fan-in in their lineage trees. Queries were then performed on the MySQL-based prototype and the Hypertable-based prototype, both running on a 2.4 GHz Intel Core Duo-based machine with 3.5 GB memory and running Fedora Core 8. Each query requested the entire known lineage of a file. The time it took to answer the query is plotted as a function of the number of levels in the lineage tree. Each plot line represents lineage trees with a different fan-in. Figure 3 shows the query response times for the MySQL-

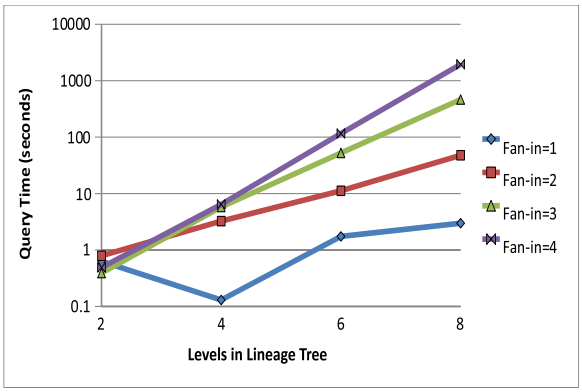

Figure 4: Response times from Hypertable.

based prototype, while Figure 4 plots the corresponding response times from Hypertable. Preliminary analysis indicates that the dramatically greater time it takes to retrieve a lineage tree from Hypertable has to do with its reliance on Hadoop [9] as its backing store. We concluded that there remains significant room for optimization.

6 Rebalancing Record Replication

The Bonsai scheme [7] attempted to trade the storage cost used for replicating lineage records corresponding to primitive operations for a reduction in the number of remote queries that would need to be made to reconstruct a complete lineage tree. Since the pruning operation was limited to inspecting the lineage tree of a single file, it could not optimize across multiple files' lineage trees. For example, if a remotely stored subtree occurred in multiple local files' lineage, the increased likelihood of queries relating to that subtree was not accounted for in choosing whether to replicate the subtrees' records locally.

We now describe a construction for choosing how to perform record replication by optimizing across all the lineage metadata within the Grid cluster.

6.1 Merging Lineage Trees

We can merge the lineage trees to form a lineage graph $G = (V, E)$. The set of nodes V in the graph is the union of the nodes from all the lineage trees. The set of edges E is the union of all the edges from all the lineage trees. An edge $e \in E$ in the graph has weight w_e which is the number of lineage trees in which that edge appears. If a query workload is known in advance, then each edge weight w_e (where $e = (u, v)$) can be defined to reflect the quantity of communication between nodes u and v for answering queries. The weight w_v of node v is the quantity of lineage metadata stored by the node.

6.2 Defining Sub-Clusters

We group the nodes of the graph into sub-clusters where each node in a sub-cluster maintains replicas of the lineage metadata of all the other nodes in the same sub-cluster. When a query from node u needs access to the lineage metadata from a node v, communication with any node in the same sub-cluster as v suffices. If u and v are in the same sub-cluster, there is no communication cost but there is a storage cost for replicating the lineage metadata of u at v and that of v at u. Similarly, if u and v are in different sub-clusters, then there is communication cost for answering a query but no storage cost for replicating the lineage metadata of u at v and v at u.

6.3 Partitioning into Sub-Clusters

Given a lineage graph, we wish to find an optimal partition $S_1, S_2, \ldots, S_k$, $\bigcup_i S_i = V$ of the graph into sub-clusters to minimize the communication and storage costs in the Grid cluster. Each component S_i of the partition is a sub-cluster. Let $S(v)$ be the sub-cluster that contains the node v. Let E_{in} (*intra-sub-cluster* edges) be the subset of edges that are within a cluster, that is, $E_{in} = \{e = (u, v) \in E | S(u) = S(v)\}$ and E_{out} (*inter-sub-cluster* edges) be the subset of edges that cross from one sub-cluster to another, that is, $E_{out} = \{e = (u, v) \in E | S(u) \neq S(v)\}$. Furthermore, we denote the set of *intra-sub-cluster* edges in sub-cluster S_i with $E_{in}(S_i) \subseteq E_{in}$.

6.4 Constructing the Objective

The benefit of such a partition is that we can eliminate intra-sub-cluster communications, which can be calculated to be $\sum_{e \in E_{in}} w_e$. The cost of the partition is the extra storage needed at each node to maintain the lineage metadata from the other nodes in the same sub-cluster. That is, the total lineage metadata in a sub-cluster is $\sum_{v \in S_i} w_v$ and is duplicated at all the nodes in the sub-cluster S_i. This results in a total amount $|S_i| \sum_{v \in S_i} w_v$ of lineage metadata in the sub-cluster.

Our goal is to find a partition that maximizes our benefit after removing the cost, that is, we want to maximize the following objective function:

$$\sum_{e \in E_i} w_e - \sum_k |S_i| (\sum_{v \in S_i} w_v). \quad (1)$$

Intuitively, adding more nodes to a sub-cluster increases the first term since less inter-cluster communication is needed to answer queries. However, it also increases the absolute value of the second term since more storage is used at each of the nodes in the sub-cluster.

6.5 Augmenting Edges

To solve the optimization problem, we use a transformation that removes the weight on the nodes while keeping the same objective function. We first augment the graph with a set of *augmenting edges*. We then consolidate the edges if multiple edges exist between pairs of nodes. Multiple edges may exist because we added augmenting edges.

For each pair of nodes u and v, we add an edge e^a with weight $w_{e^a} = (w_u + w_v)$. Intuitively, this augmenting edge captures the fact that node u stores the lineage metadata from node v, and v stores the lineage metadata of u. Such augmenting edges are added between every pair of nodes. Figure 5 shows an example with augmenting edges. The weights of solid edges denote the reduction in communication costs, while the weights of the dashed edges reflect the storage costs for replicating the lineage metadata.

Let E^a denote the set of augmenting edges and $E^a_{in}(S_i)$ be the set of augmenting edges that are within sub-cluster S_i, that is, $E^a_{in}(S_i) = \{e = (u, v) \in E^a | S(u) = S(v) = S_i\}$. After the graph is augmented this way, the storage cost within a sub-cluster S_i, $|S_i| \sum_{v \in S_i} w_v$, can be expressed in terms of the weights on the edges in $E^a_{in}(S_i)$, that is, as $\sum_{e^a \in E^a_{in}(S_i)} w_{e^a}$.

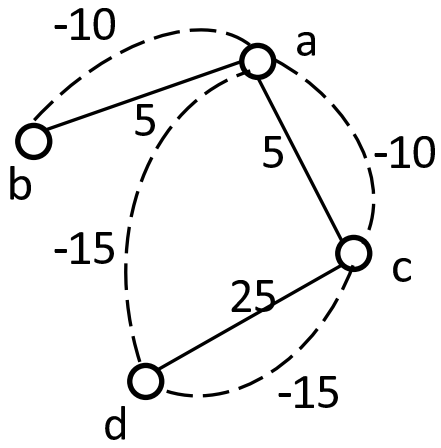

Figure 5: Each augmenting edge is weighted with cost to store the lineage metadata at the two nodes it is incident upon.

6.6 Consolidating Edges

We now have two types of edges, one of which captures the costs and the other the benefits. There may be pairs of nodes between which both types of edges exist. We can combine the edges to obtain a final edge. Specifically, if there are edges e and e^a between u and v, we can consolidate the two edges into one edge e^f whose weight is $w_{e^f} = w_e - w_{e^a} = w_e - (w_u + w_v)$. This is illustrated in Figure 6.

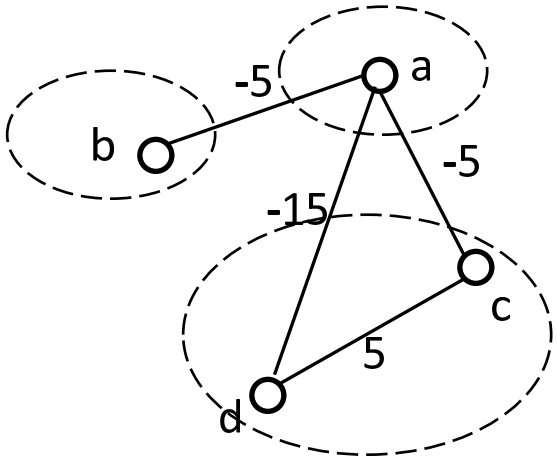

Figure 6: The augmenting edges can be consolidated with the initial set of edges, as illustrated here for the graph shown in Figure 5. The dashed circles show the partition into sub-clusters that maximize the savings in communication after accounting for the cost for replicating the lineage metadata.

Let E^f be the set of edges after consolidation. Let E^+ be the set of edges e^f for which the weight w_{e^f} is positive and E^- be the set of edges for which the weight w_{e^f} is negative. Let W^+ be the sum of the weights of the edges in E^+ and W^- be the absolute value of the sum of the weights of the edges in E^-. Note that $W^- > 0$ must hold since absolute values are used. E^+ and E^- (and hence W^+ and W^-) are defined by the graph and are not affected by how the graph is partitioned.

6.7 Optimizing the Partitioning

Given the set of consolidated edges, the transformed optimization problem is to find a partition into sub-clusters that maximizes the difference between the total weight of the intra-sub-cluster positive edges and the total weight of the intra-sub-cluster negative edges, that is:

$$\sum_{e \in E^+_{in}} w_e - \sum_{e \in E^-_{in}} |w_e| \tag{2}$$

where $|w_e|$ denotes the absolute value of edge weights that were negative.

To facilitate the optimization process, W^- can be added to the objective function. This does not change the selection made using the objective function because W^- is the same regardless of how the cluster is partitioned.

The above optimization problem is equivalent to maximizing the total intra-sub-cluster positive edge weights and inter-cluster negative edge weights, that is:

$$\sum_{e \in E^{+}_{in}} w_e + \sum_{e \in E^{-}_{out}} |w_e| \quad (3)$$

6.8 Approximate Partitioning

The above objective function is used in graph correlation clustering. The problem can be solved using semi-definite programming to search for a partition that approximates the optimal one [2, 22].

Each node is represented by a point x_i on the unit sphere in R^n, where n is the number of nodes in the graph. If two nodes are in the same sub-cluster, the two corresponding points, x_i and x_j, should be close, that is, $x_i \cdot x_j$, the inner product of x_i and x_j, should be close to 1. If the two nodes are in different sub-clusters, the two corresponding points should be orthogonal, that is, $x_i \cdot x_j$ should be close to 0. The optimization problem then reduces to

$$\begin{array}{ll} \text{maximize:} & w_e(x_i \cdot x_j) + w_e(1 - x_i \cdot x_j) \\ \text{subject to:} & \forall\, i\ x_i \cdot x_i = 1 \\ & \forall\, i, j\ x_i \cdot x_j \geq 0 \end{array}$$

Once solutions for x_i are obtained, the sub-clusters can be formed as follows. A number of hyper-planes that divide the unit sphere into sections can be picked. The nodes with corresponding points that fall in the same section are allocated to the same sub-cluster. Alternatively, the points on the sphere can be mapped to a high-dimensional Euclidean space while preserving their relative distances, and other standard clustering techniques can be applied. The resulting partition will yield a 0.76 approximation of the optimal solution [2, 22].

7 Related Work

Several Grid environments account for data provenance in their design. $^{my}Grid$ [26] allows application-level annotation of the data's provenance, which it then stores in the user's repository. This does not enable other users of the data to determine its provenance. The Provenance Aware Service Oriented Architecture (PASOA) project arranges for data transformations to be reported to a central provenance service [23], which can be queried by other users as well. The centralized approach ensures that the provenance metadata does not have to be replicated. However, in the event that the metadata is heavily accessed, the latency of performing remote lookups can degrade application performance.

The GALE project [6], which aims to let monolingual users query information from newscasts and documents in multiple languages, provides a motivating use case for a decentralized approach to provenance metadata management. Input data is transformed multiple times for automatic speech recognition, machine translation between languages, and distillation to extract responses to a query. There are several steps in the pipeline of operations and they can be performed by multiple versions of software being developed in parallel by experts from 15 universities and corporations. Since the functionality of different revisions of the same tool can also differ, the description of the tool that produced a piece of data serves as an input for subsequent tools in the pipeline. This metadata is currently maintained in a file that accompanies the data. If low latency access to the provenance of data were available, maintenance of the accompanying file would be obviated. The low latency is of significance because the metadata would enable querying to determine which combinations of tools in the pipeline have yielded a better-quality output.

8 Conclusion

Lineage metadata differs from other types of filesystem attributes in notable ways. First, it grows in size significantly over time. Second, when a file moves from one node to another, it is important for many applications for the lineage metadata to be retained. We described a number of schemes to store the metadata, including in auxiliary files, filesystem data structures, local databases, file servers, in-band encoding, and file headers, along with their strengths and weaknesses. Next we outlined hybrid approaches for using an overloaded namespace so legacy applications can transparently transfer lineage metadata, trade the level of replication of lineage records for the likelihood of query completion, and use of a distributed database to provide global visibility of the lineage metadata. Finally, we framed the problem of communication-aware consolidation of lineage records across all files in a distributed system, and provided an approximation scheme for optimizing the replication in a Grid cluster.

References

[1] M. Nedim Alpdemir, Arijit Mukherjee, Norman W. Paton, Alvaro A. A. Fernandes, Paul Watson, Kevin Glover, Chris Greenhalgh, Tom Oinn, and Hannah Tipney, Contextualised workflow execution in myGrid, Proceedings of the European Grid Conference, Lecture Notes in Computer Science, Volume 3470, Springer-Verlag, 2005.

[2] M. Charikar, V. Guruswami, and A. Wirth, Clustering with Qualitative Information, 44th Annual IEEE Symposium on Foundations of Computer Science, 2003.

[3] D. G. Clarke and D. M. Clark, Lineage, Elements of Spatial Data Quality, 1995.

[4] Document Type Definition, http://www.w3.org/TR/REC-xml/

[5] Filesystem in Userspace, http://fuse.sourceforge.net

[6] Global Autonomous Language Exploitation, http://www.speech.sri.com/projects/GALE/

[7] Ashish Gehani and Ulf Lindqvist, Bonsai: Balanced lineage authentication, 23rd Annual Computer Security Applications Conference (ACSAC), IEEE Computer Society, 2007.

[8] Ashish Gehani and Ulf Lindqvist, VEIL: A system for certifying video provenance, Proceedings of the 9th IEEE International Symposium on Multimedia, 2007.

[9] Hadoop Distributed File System, http://hadoop.apache.org/core

[10] Hypertable Query Language, http://code.google.com/p/hypertable/wiki/HQLTutorial

[11] Hypertable Distributed Database, http://www.hypertable.org

[12] H. V. Jagadish and F. Olken, Database management for life sciences research, SIGMOD Record, Vol. 33, 2004.

[13] Aditya Kashyap, File system extensibility and reliability using an in-kernel database, Master's Thesis, State University of New York, Stony Brook, 2004.

[14] Steve R. Kleiman, Vnodes: An architecture for multiple file system types in Sun UNIX, Proceedings of the USENIX Summer Conference, 1986.

[15] Sander Klous, Jamie Frey, Se-Chang Son, Douglas Thain, Alain Roy, Miron Livny, and Jo van den Brand, Transparent access to Grid resources for user software, concurrency and computation: Practice and experience, Volume 18(7), 2006.

[16] MySQL, http://www.mysql.com/

[17] Kiran-Kumar Muniswamy-Reddy, David A. Holland, Uri Braun, and Margo Seltzer, Provenance-aware storage systems, Proceedings of the USENIX Annual Technical Conference, 2006.

[18] J. L. Romeu, Data quality and pedigree, Material Ease, 1999.

[19] Lineage File System, http://crypto.stanford.edu/~cao/lineage.html

[20] Y. L. Simmhan, B. Plale, and D. Gannon, A survey of data provenance in e-science, SIGMOD Record, Vol. 34(3), 2005.

[21] Stefan Stonjek, Morag Burgon-Lyon, Richard St. Denis, Valeria Bartsch, Todd Huffman, Elliot Lipeles, and Frank Wurthwein, Using SAM datahandling in processing large data volumes, UK e-Science All Hands Meeting, 2004.

[22] C. Swany, Correlation Clustering: Maximizing agreements via semidefinite programming, ACM Symposium on Discrete Algorithms, 2004.

[23] Martin Szomszor and Luc Moreau, Recording and reasoning over data provenance in Web and Grid services, International Conference on Ontologies, Databases and Applications of Semantics, Lecture Notes in Computer Science, Volume 2888, Springer-Verlag, 2003.

[24] TeraGrid, http://teragrid.org/

[25] Erez Zadok, Ion Badulescu and Alex Shender, Extending file systems using stackable templates, Usenix Technical Conference, 1999.

[26] J. Zhao, C. A. Goble, R. Stevens, and S. Bechhofer, Semantically linking and browsing provenance logs for E-science, ICSNW, 2004.