Provenance for the Cloud

Kiran-Kumar Muniswamy-Reddy, Peter Macko, and Margo Seltzer
Harvard School of Engineering and Applied Sciences

Abstract

The cloud is poised to become the next computing environment for both data storage and computation due to its pay-as-you-go and provision-as-you-go models. Cloud storage is already being used to back up desktop user data, host shared scientific data, store web application data, and to serve web pages. Today's cloud stores, however, are missing an important ingredient: ***provenance****.*

Provenance is metadata that describes the history of an object. We make the case that provenance is crucial for data stored on the cloud and identify the properties of provenance that enable its utility. We then examine current cloud offerings and design and implement three protocols for maintaining data/provenance in current cloud stores. The protocols represent different points in the design space and satisfy different subsets of the provenance properties. Our evaluation indicates that the overheads of all three protocols are comparable to each other and reasonable in absolute terms. Thus, one can select a protocol based upon the properties it provides without sacrificing performance. While it is feasible to provide provenance as a layer on top of today's cloud offerings, we conclude by presenting the case for incorporating provenance as a core cloud feature, discussing the issues in doing so.

1 Introduction

Data is information, and as such has two critical components: what it is (its contents) and where it came from (its ancestry). Traditional work in storage and file systems addresses the former: storing information and making it available to users. Provenance addresses the latter. Provenance, sometimes called lineage, is metadata detailing the derivation of an object. If it were possible to fully capture provenance for digital documents and transactions, detecting insider trading, reproducing research results, and identifying the source of system break-ins would be easy. Unfortunately, the state of the art falls short of this ideal.

Current research has demonstrated the feasibility of automatically capturing provenance at all levels of a system, from the operating system [18, 30] to applications [27]. Our goal is to extend provenance to the cloud.

Provenance is particularly crucial in the cloud, because data in the cloud can be shared widely and anonymously; without provenance, data consumers have no means to verify its authenticity or identity. The web has taught us that widely shared, easy-to-publish data are useful, but it has also taught us to be skeptical consumers; it is impossible to know exactly how updated or trustworthy data on the web are. We should solve the problem now while cloud services are still new and evolving. For example, Amazon's "Public Data Sets on AWS" provides free storage for public data sets such as GenBank [2], US census data, and PubChem [1]. If researchers are to make the most of these data sources, they must be able to accurately identify the process used to generate the data. Provenance, bound to the data it describes, provides the necessary information for verifying the process used to generate the data. Similarly, provenance can be used to debug experimental results and to improve search quality. We discuss these use cases in Section 2.2.

As both automatic provenance collection and cloud storage are relatively new developments, it is not obvious how to best record provenance in the cloud. We begin by identifying four properties crucial for provenance systems. First, *provenance data-coupling* states that when a system records data and provenance, they match – the provenance accurately describes the data recorded. Second, *multi-object causal ordering* states that ancestors described in an object's provenance exist, i.e., the objects from which another object is derived. This ensures that there are no dangling provenance pointers. Third, *data-independent persistence* states that provenance must persist even after the object it describes is removed. Fourth, *efficient query* states the system supports queries on provenance across multiple objects. We discuss these properties and the implications of violating them in Section 3.

Using these properties as a metric, we designed three alternative protocols for storing provenance using current cloud services. The protocols vary in complexity, the guarantees they make, and the distributed cloud components they involve. The first protocol is the simplest and uses only a cloud store. In turn, it is the weakest of the protocols. The second protocol satisfies a larger subset of the properties and uses a cloud store and a cloud database. The third protocol uses a cloud store, a cloud database, and a distributed cloud queuing service and satisfies all the properties. The database and

queue have the same availability, reliability, and scalability properties as the store. We discuss the protocols and the properties they satisfy in Section 4.3. We use a Provenance Aware Storage System (PASS) [30] augmented to use Amazon Web Services (AWS) [5] as the backend to build and evaluate the protocols for storing provenance. Based on our experience designing and implementing protocols for storing provenance on current cloud offerings, we discuss research challenges for providing native provenance support on the cloud.

The contributions of this paper are:

1. Definition of properties that provenance systems must exhibit.
2. Design and implementation of three protocols for storing provenance and data on the cloud, evaluating each protocol with respect to the properties we established.
3. Evaluation and comparison of the cost and performance of our three provenance storage protocols.

The rest of the paper is organized as follows. In the next section, we provide background on provenance and our provenance collection substrate, discuss use cases for provenance in the cloud, and introduce the cloud services that are most pertinent to this work. In Section 3, we present the desirable properties for storing provenance in the cloud. In section 4, we discuss the challenges unique to storing provenance on the cloud and present the architecture and implementation of our three provenance recording protocols. In section 5, we evaluate the protocols for overhead, throughput, and cost. We discuss related work in section 6. We discuss the challenges for providing native support for provenance in the cloud in section 7, and we conclude in section 8.

2 Background

Provenance can be abstractly defined as a directed acyclic graph (DAG). The DAG structure is fundamental and holds for all provenance systems irrespective of the software abstraction layer at which they operate. The nodes in the DAG represent objects such as files, processes, tuples, data sets, etc. The edges between two nodes indicates a dependency between the objects. Nodes can have attributes. For example, a process node has attributes such as the the command line arguments, version number, etc. A file node has name and version attributes. Each version of a file or process is represented by a distinct node in the DAG. The provenance graph, by definition, is acyclic as the presence of cycles would indicate that an object was its own ancestor.

2.1 Provenance Aware Storage System (PASS)

We use our PASS [30] system to collect provenance. PASS is a storage system that transparently and automatically collects provenance for objects stored on it. It observes application system calls to construct the provenance graph. For example, when a process issues a `read` system call, PASS creates a provenance edge recording the fact that the process depends upon the file being read. When that process then issues a `write` system call, PASS creates an edge stating that the file written depends upon the process that wrote it, thus transitively recording the dependency between the file read and the file written. For processes, PASS records several attributes: command line arguments, environment variables, process name, process id, execution start time, the file being executed, and a reference to the parent of the process. For all other objects (files, pipes, etc.), PASS records the name of the object (pipes do not have names). Prior to this work, PASS used local file systems and network attached storage as its storage backend; this work leverages PASS as a provenance collection substrate and extends its reach to using the cloud as the storage backend.

2.2 Cloud Provenance Use Cases

The following use cases illustrate the utility and need for provenance in the cloud.

Debug Experimental Results: The Sloan Digital Sky Survey (SDSS) [20] is an online digital astronomy archive consisting of raw data from various sources (e.g., imaging camera, photometric telescope, etc.). It also provides an environment for researchers to process and store data in personal databases. Since researchers use of the environment is bursty, one can imagine using cloud stores and virtual machines to provide this service. Consider a scenario where SDSS administrators upgrade the software distribution on the compute node images unbeknownst to the users. Suppose further that when users run their scripts, the resulting output is flawed. Without provenance, users are left to manually search for clues explaining the change in behavior. With provenance, users can compare the provenance of newly generated output with the provenance of older output to determine what has changed between invocations. For example, if a new JVM had been introduced, the difference in JVMs would be readily apparent in the provenance output.

Detect and Avoid Faulty Data Propagation: The SDSS processed data is produced by a pipeline of data reduction operations. A scientist using the data might want to ensure that she is using an appropriately calibrated data set. Without provenance, the scientist has no means to verify that she is using data processed by

the correct software. With provenance, the scientist can examine the data's provenance to verify that appropriate versions of the tools were used to process the data. In addition, provenance enables users to discover how far faulty data has propagated throughout a data processing pipeline.

Improving Text Search Results: Shah et. al. [39] showed that provenance can improve desktop search results. The provenance graph provides dependency links between files, similar to hyperlinks between webpages, that can be used to improve the quality of search results. Shah's scheme first uses a pure content-based search to compute an initial set of documents. Then, they traverse the provenance DAG of the initial document set P times. At each iteration of the traversal, they update the weight for each node based on the number of incoming/outgoing edges. After P runs, they re-rank the files and include new files to the list based on the weights computed.

Similarly, provenance can be used to improve search quality for data stored on the cloud. For example, consider a scenario where a user archives data on the cloud. Without any content-based indexing, searching that archived data requires downloading each file to the user's desktop. Content-based indexing reduces the number of files the user needs to download. Content-based indexing refined by provenance, such as inter-file dependencies, inputs, or command-line arguments from the program that generated the data, further reduces the effort required to locate a particular file.

2.3 Cloud Services

We next provide a brief description of the cloud services that are most pertinent to this work.

Object Store Service: A cloud object service allows users to store and retrieve data objects. Service providers generally provide a REST-based interface for accessing objects, with each object identified by a unique URI. The service allows users to *PUT*, *GET*, *COPY*, and *DELETE* objects. The PUT operation overwrites any previous versions of an object. With each object, clients can store some metadata, represented as <name,value> pairs. The PUT operation supports atomic updates to both data and metadata. The cost of using such services is based on the number of bytes transferred (both to and from), the storage space utilization, and the number of operations performed. Amazon *Simple Storage Service* (S3) [37] and Microsoft Azure *Blob* [6] are examples of object store services.

Database Service: A cloud database service provides index and query functionality. The data model is semi-structured, i.e., it consists of a set of rows (called items), with each row having a unique itemid and each item having a set of attribute-value pairs. The attribute-value pairs present in one item need not be present in another, and an item can have multiple attributes with the same name. For example, an item can have two phone attributes with different values. The database service provides the same reliability and availability guarantees as the data store. Amazon's *SimpleDB* [38] and Microsoft Azure's *Table* [8] are examples of such services. SimpleDB supports attribute names and values up to 1 KB, while Azure allows them to be up to 64KB. SimpleDB provides a traditional SELECT query interface, whereas Azure provides a LINQ [25] query interface.

Messaging Service: Distributed messaging systems provide a queuing abstraction allowing users to exchange messages between distributed components in their systems. Queues are typically identified by a unique URL. Users can perform operations such as *SendMessage*, *ReceiveMessage*, and *DeleteMessage*. The messaging service provides similar guarantees to that of the corresponding cloud store. Message delivery is generally best-effort, in-order message delivery. Amazon's *Simple Queueing Service* (SQS) [41] and Microsoft Azure *Queue* [7] are examples of such Messaging systems. Both SQS and Queue enforce an 8KB limit on messages.

2.3.1 Eventual Consistency

As with other distributed systems, building highly scalable cloud services involves making various choices in the design space. A number of recent systems that operate at the cloud scale have chosen to be provide high performance and high availability while providing a weaker form of data consistency, called eventual consistency. AWS is an example of an eventually consistent service suite. This implies that, for example, a client performing a GET operation on an S3 object immediately after a PUT on that object might receive an older copy of the object as S3 might service that request from a node that has not yet received the latest update. If two clients update the same object concurrently via a PUT, the last writer wins, but for a non-deterministic period of time after a PUT, a subsequent GET operation might return either of the two writes to the client. Azure services, on the other hand, are strictly consistent; a client is guaranteed to receive the latest version of an object. Eventual consistency dictates that clients must design appropriate mechanisms to detect inconsistencies between objects. We designed our protocols assuming eventual consistency, as it is the weaker form of concurrency; anything that works with eventual consistency will work trivially with stronger models.

3 Provenance System Properties

There are four properties of provenance systems that make their provenance truly useful. We motivate and introduce these properties.

Provenance Data Coupling The *data-coupling* property states that an object and its provenance must match – that is, the provenance must accurately and completely describe the data. This property allows users to make accurate decisions using provenance. Without data-coupling, a client might use old data based on new provenance or might use new data based on old provenance. In both of these cases, the user relying on the provenance is misled into using invalid data.

Systems that do not provide data-coupling during writes can detect data-coupling violations on access and withhold or explicitly identify objects without accurate provenance. For example, if the provenance includes a hash of the data, we can compute the hash of a data item to determine if its provenance refers to this version of that data. Detection is, at best, a mediocre replacement for data-coupling, because although users will not be misled, they cannot safely use available data when its provenance is wrong.

Given the eventual consistency model of existing cloud services and the fact that we cannot modify existing cloud services, we find a weaker form of the property, *Eventual data-coupling* practical. In eventual data-coupling, the data and its provenance might not be consistent at a particular instant, but are guaranteed to be eventually match. With eventual data-coupling, a system requires detection, since there may exist intervals during which an object and its provenance do not match.

Multi-object Causal Ordering This property acknowledges the causal relationship among objects. If an object, O, is the result of transforming input data P, then the provenance of O is the super-set of the provenance of P. Thus, a system must ensure that an object's ancestors (and their provenance) are persistent before making the object itself persistent. Multi-object Causal Ordering violations occur when the system writes an object to persistent store before writing all its ancestors, and the system crashes before recording those ancestors and their provenance. These violations produce dangling pointers in the DAG. Similar to eventual data-coupling, a weaker form of the property *Eventual Causal Ordering* is realizable. A system still requires detection to account for the intervals during which an object's provenance may be incomplete, because its ancestors and their provenance are not yet persistent or not available due to eventual consistency.

Data-Independent Persistence This property ensures that a system retains an object's provenance, even if the object is removed. As in the last section, assume that P is an ancestor of O. If P were removed, O's provenance still includes the provenance of P, so a system must make sure to retain P's provenance, even if P no longer exists. If P's provenance is deleted when P is deleted, parts of the provenance DAG will become disconnected. If P had no descendants, then a system might choose to remove its provenance, since it would no longer be accessible via any provenance chain. Another approach to solving this problem is to copy and propagate an ancestor's provenance to its descendants. This is inefficient in terms of space and can quickly become unwieldy.

Efficient Query Since provenance is created more frequently than it is queried, efficient provenance recording is essential. However, efficient query is also important as provenance must be accessible to users who want to access or verify provenance properties of their data. In scenarios where the number of objects are few or users already know the objects whose provenance they want to access, efficiency is not an issue. Efficiency matters, however, when the number of objects is sizeable and users are unsure of the objects they want to access. For example, users might want to retrieve objects whose provenance matches certain criteria. In scenarios such as this, if a system stores provenance, but that provenance is not easily queried, the provenance is of reduced value.

4 Protocol Design and Implementation

We begin this section by presenting the challenges unique to the cloud that guided our protocol design. Next, we present a high level architectural overview and implementation of our system. Finally, we describe each of our three protocols in detail. For each protocol, we discuss its advantages and limitations. For the rest of the paper, we use AWS as the cloud backend as it is the most mature product on the market.

4.1 Challenges

The cloud presents a completely different environment from the ones addressed by previous provenance systems. The cloud is designed to be highly available and scalable. None of the existing provenance solutions, however, account for availability or scalability in their design. The cloud is also not extensible, while all existing solutions required making changes to the operating system, the workflow engine, the application, or some other piece of software. Further, the long latency between users and the cloud presents different update and error models. These properties make managing provenance in the cloud different from managing it on local storage.

Extensibility: Most existing provenance systems assume the ability to modify system components. For ex-

ample, PASS uses either a file system or an NFS service as the storage backend. PASS defined new extensions to the VFS interface to couple data and provenance [28]. The Virtual Data Grid [17] and myGrid [42] workflow engines integrate provenance collection into the workflow execution environment. The PASOA [34] framework for recording provenance in service oriented architectures assumes the existence of a custom designed provenance recording service. In the case of the cloud, however, modifying or extending existing services is not possible.

Availability: One can imagine building a wrapper service that acts as a front to the cloud services and provides a cloud provenance storage service that satisfies the properties we identified. For the approach to be viable, however, the wrapper service has to match the availability of the cloud. If not, the overall availability is reduced to the availability of the wrapper service. Building such a highly available wrapper service is counterproductive as it requires a great deal of effort and infrastructure investment, defeating the very purpose of moving to the cloud. Hence, we design protocols that leverage existing services while satisfying the properties.

Scalability: In order to make the provenance queryable, most systems store provenance in a database. Hence, we considered storing the provenance in a database backed by an S3 object (e.g., a MySQL or Berkeley DB database stored in the S3 object). The provenance would then be queryable, but this approach would not scale. First, to avoid corrupting the database, clients need to synchronize updates between each other. A single global lock is a scalability bottleneck, and a distributed lock service would introduce the potential for distributed deadlock. Second, due to the update granularity of cloud stores, clients need to download the database object for every update, which also does not scale. One can, of course, use more sophisticated parallel database solutions. This is, however, expensive and hard to maintain and is against the pay-as-you-use model of the cloud. All this points to using a scalable cloud service such as SimpleDB to store provenance, as we do in two of our protocols (Section 4.3.2 and Section 4.3.3). Storing the provenance in a separate service opens the issue of coordinating updates between the database service and object store service, which we address while describing the protocols.

Some of the properties of the cloud, on the other hand, make storing provenance easier. For example, NFS and the file system have to ensure consistency in the face of partial object writes, while cloud stores deal only with complete objects. Hence cloud provenance does not have to consider partial write failures.

4.2 Architecture Overview

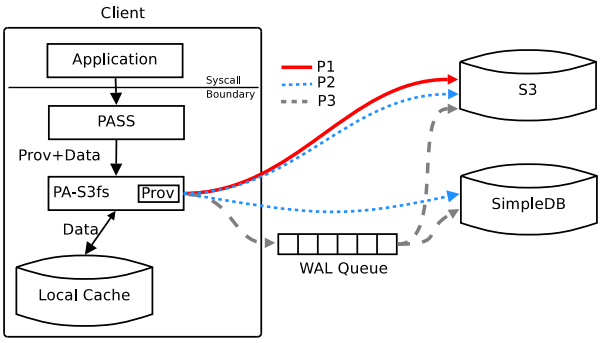


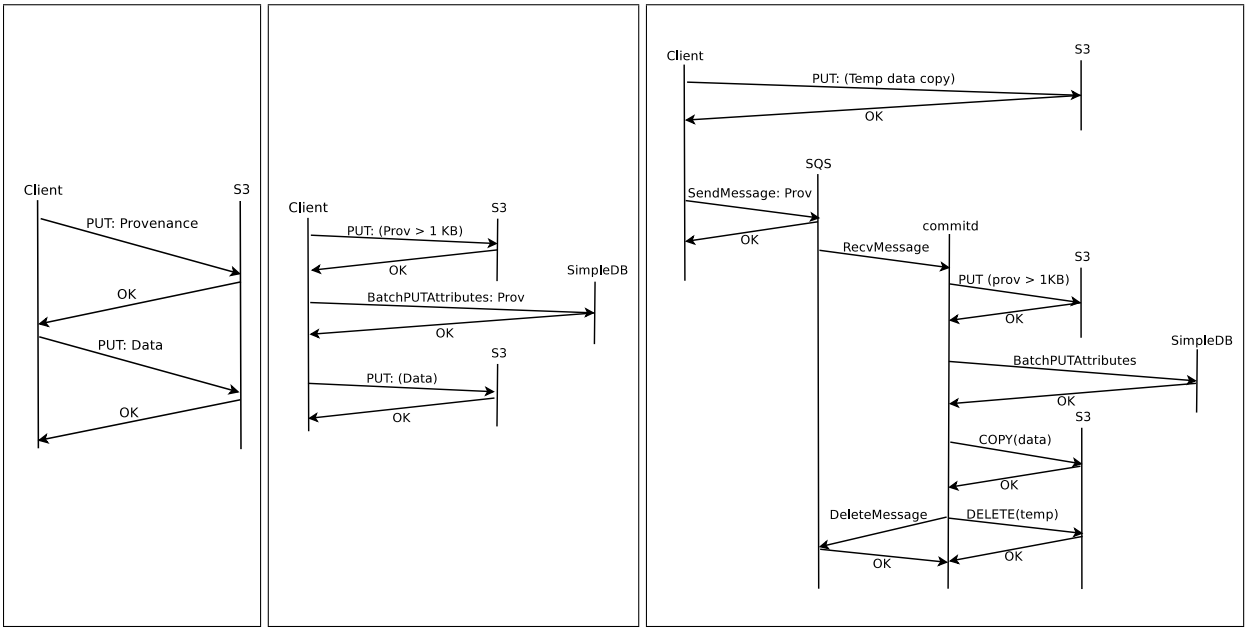
Figure 1: Architecture: The figure shows how provenance is collected and the cloud is used as a backend.

Figure 1 shows our system architecture. The system is composed of the client (compute node) and the cloud. The client is in turn composed of PASS and PA-S3fs. PASS monitors system calls, generating provenance and sending both provenance and data to Provenance Aware S3fs (PA-S3fs). PA-S3fs, a user-level provenance-aware file system interface for Amazon's S3 storage service, caches data and provenance on the client to reduce traffic to S3. PA-S3fs caches data in a local temporary directory and the provenance in memory. On certain events, such as file close or flush, it sends both the data and the provenance to the cloud using one of the protocols P1, P2, or P3, which we discuss in the next subsections. Further, PASS has algorithms built into it that preserve causality by carefully creating logical versions of objects when they are simultaneously updated by multiple processes at the same client [29]. The provenance recorded in the cloud by the protocols reflects this versioning.

Implementation PA-S3fs is derived from S3fs [36], a user-level FUSE [19] file system that provides a file system interface to S3. PA-S3fs extends S3fs by interfacing it to PASS, our collection infrastructure. PASS internally uses the Disclosed Provenance API (DPAPI) [28] to satisfy the properties specified in Section 3 and eventually stores the provenance on a backend that exports the DPAPI. Hence, extending S3fs to PA-S3fs translates to extending S3fs and FUSE to export the DPAPI.

4.3 Protocols

Table 1 summarizes our three protocols with respect to the properties in Section 3. Although we discuss the protocols in the context of moving data from users to the cloud, they can also be used while replicating data and provenance across different cloud service providers. Further, while our implementation is based on extending the file system interface to the cloud, the protocols are



(a) (b) (c)

Figure 2: Protocol 1 (a): Both provenance and data are recorded in a cloud object store (S3). Protocol 2 (b): Provenance is stored in a cloud database (SimpleDB) and data is stored in a cloud store (S3). Protocol 3 (c): Provenance is stored in a cloud database (SimpleDB) and data is stored in a cloud store (S3). A cloud messaging service (SQS) is used to provide data-coupling and multi-object causal ordering.

Property	P1	P2	P3
Provenance Data-Coupling	✖	✖	✔
Multi-object Causal Ordering	✔	✔	✔
Efficient Query	✖	✔	✔

Table 1: Properties Comparison. A check mark indicates that the property is supported, otherwise it is not.

independent of the storage model and applicable whenever provenance has to be stored on the cloud.

4.3.1 P1: Standalone Cloud Store

Storage Scheme: We map each file to an S3 object and store the object's provenance as a separate S3 object. It might seem attractive to record provenance as metadata of the object, but that introduces two problems. First, removing the object removes its provenance, violating provenance persistence. Second, most systems impose a hard limit on the size of an object's metadata. To address the deletion issue, one could truncate the data in the object and rename the object to a shadow directory on deletion. To address the metadata limit, one could store the extra provenance in the first n bytes of the object itself and on deletion, truncate the data part of the object. Instead, we create a primary S3 object containing the data and a second, provenance S3 object, named with a `uuid` and containing the primary object's provenance plus an additional provenance record containing the name of the primary S3 object. In the primary S3 object's metadata, we record a version number and the `uuid`, thus linking the data and its provenance. For objects that are not persistent, such as pipes and processes, we record only the provenance object with no primary object. For provenance queries, this scheme requires us to lookup the primary object and then retrieve the provenance whereas the previous scheme can avoid this. On deletions, however, the previous scheme requires the system to update all provenance referring to the object to point to the new name assigned on deletion. We chose to store provenance in a separate object, because provenance queries are infrequent relative to object operations, and updating provenance pointers on every delete can be expensive.

Protocol: Figure 2a depicts protocol P1. On a file close (or flush), we perform the following operations:

1. Extract the provenance of the file (cached by PA-S3fs). PUT the provenance into the S3 provenance object. If the provenance object already exists, GET the existing object, append the new provenance to it, and then issue a PUT.

2. PUT the data object with metadata attributes containing the name of the provenance object and the current version.

Before sending the provenance and data of an object, we need to identify the ancestors of the object and send any unrecorded ancestors and their provenance to ensure multi-object causal ordering. A client can, at best, assure a consistency model comparable to that of the underlying system; that is if the underlying system supports eventual consistency, then the best P1 can do is ensure *eventual multi-object causal ordering*. A reading client that wants to check multi-object causal ordering must use Merkle hash trees or some similar scheme to verify the property. If the property is not satisfied, the client should try refreshing the data until the objects do meet the multi-object causal ordering property.

Discussion: This protocol does not support data-coupling, but using version numbers stored both in the provenance object and the primary object's metadata, clients can detect provenance decoupled from data. P1 achieves eventual multi-object causal ordering if it sends all the ancestors of an object and their provenance to S3 before sending the object's provenance to S3. However, such an implementation can suffer from high latency. Querying is inefficient as we cannot retrieve objects by their individual provenance attributes; we can only retrieve all of an object's provenance via a GET call. If we do not know the exact object whose provenance we seek, then we need to iterate over the provenance of every object in the repository, which is so inefficient as to be impractical.

4.3.2 P2: Cloud Store with a Cloud Database

Storage Scheme: This scheme, which is already independently in use by some cloud users [13], stores each file as an S3 object and the corresponding provenance in SimpleDB. We store the provenance of a version of an object as one SimpleDB item (row in traditional databases). As in P1, we reference the provenance of an object by `uuid` assigned to the object at creation time. For example, assume that an object named *foo* has `uuid` 'uuid1', its version is 2, and it has two provenance records: (input, bar_2) and (type, file). P2 stores this in SimpleDB as:

```
ItemName=uuid1_2
attribute-name=name,attribute-value=foo
attribute-name=input,attribute-value=bar_2
attribute-name=type,attribute-value=file
```

The name attribute allows us to find an object from its provenance. We chose this one-row-per-version scheme instead of storing the provenance of all versions of an object as one SimpleDB item, as it allows users to distinguish the version to which the provenance belongs. We store provenance values larger than the 1KB SimpleDB limit as separate S3 objects, referenced from items in SimpleDB. As in P1, we store the object's current version number and uuid in its metadata.

Protocol: Figure 2b shows the second protocol. On a file close, we extract the provenance cached in memory and convert it to attribute-value pairs. We then group the attribute-value pairs by file version, construct one item for the provenance of each version of the file, and perform the following actions:

1. If any of the values are larger than 1KB, store them as S3 objects and update the attribute-value pair to contain a pointer to that object.
2. Store the provenance in SimpleDB by issuing *BatchPutAttributes* calls. SimpleDB allows us batch up to 25 items per call, hence we issue as many calls as necessary to store all the items.
3. PUT the data object with metadata attributes containing the name of the provenance object and the current version.

As in P1, P2 enforces multi-object causal ordering by recording ancestors and their provenance before sending the provenance and data of the new object.

Discussion: P2 is an improvement over P1 in that it provides efficient provenance queries, because we can retrieve indexed provenance from SimpleDB. Like P1, P2 does not provide data-coupling but can detect coupling violations and exhibits high latency to ensure multi-object causal ordering. Due to eventual consistency, we can encounter a scenario in which SimpleDB returns old versions of provenance when S3 returns more recent data (and vice versa). We detect this by comparing the version of the object in S3 and the version returned in the provenance. If they are not consistent, we can request the specific version of the provenance we need from SimpleDB.

4.3.3 P3: Cloud store with Cloud Database and Messaging Service

Storage Scheme and Overview: P3 uses the same S3/SimpleDB storage scheme as P2, but differs from P2 in its use of a cloud messaging service (SQS) and transactions to ensure provenance data-coupling. Each client has an SQS queue that it uses as a write-ahead log (WAL) and a separate daemon, the *commit daemon*, that reads the log records and assembles all the records belonging to a transaction. Once it has all the records for a transaction, the daemon pushes data in the records to S3 and provenance to SimpleDB. If the client crashes before it can log all the packets of a transaction to the WAL queue, the commit daemon ignores these records. One might be tempted to use a local log instead of an SQS

queue, but such an arrangement leads to data-coupling violations when a client crashes before the commit daemon has completely committed a transaction. By using SQS as the log, if the client running the commit daemon crashes during a commit, another machine can commit the partially completed transaction.

Messages on SQS (and Azure) cannot exceed 8KB, hence we cannot directly record large data items in the WAL queue. Instead, we store large objects as temporary S3 objects, recording a pointer to the temporary object in the WAL queue. The commit daemon, while processing the WAL queue entries, copies a temporary object to its real object and then deletes the temporary object. Both S3 and Azure do not currently support a rename operation. Hence the object has to be copied from the temporary name to the real object. One thousand copy operations cost 0.01 USD for S3 and 0.001 USD for Azure with no charge for the data transfer required to perform the copy. Hence the copy operation has minimal cost from a user's perspective. Once items are in the WAL queue, they are guaranteed to eventually be stored in S3 or SimpleDB, so the order in which we process the records does not matter.

We must, however, *garbage collect* state left over by uncommitted transactions. SQS automatically deletes messages older than four days, so we do not need to perform any additional reclamation (unless the 4-day window becomes too large) on the queue. However, temporary objects that have been stored on S3 must be explicitly removed if they belong to uncommitted transactions. We use a *cleaner* daemon to remove temporary objects that have not been accessed for 4 days.

Protocol: Figure 2c shows our final protocol. We divide the protocol into two phases: log and commit. The log phase begins when an application issues a `close` or flush on a file and consists of the following actions.

1. Store a copy of the data file with a temporary name on S3.
2. Allocate a uuid as a transaction id. Extract the provenance of the object. Group the provenance records into chunks of 8KB and store each of these chunks as log records (messages) in the WAL queue. The first bytes of each message contain the transaction id and a packet sequence number. The first message has the following additional records: A record indicating the total number of packets in the transaction, a record that has a pointer to the temporary object, and a record tagged with the transaction id and the object version.

In the commit phase, the commit daemon assembles the packets belonging to transactions and once it receives all the packets of a transaction, performs the following actions.

1. Store any provenance record larger than 1KB into a separate S3 object and update the attribute-value pair to contain a pointer to the S3 object.
2. Store the provenance in SimpleDB by issuing *BatchPutAttributes* calls. SimpleDB allows us batch up to 25 items per call, hence we issue as many calls as necessary to store all the items.
3. Execute an S3 *COPY* method to copy the temporary S3 object to its permanent S3 object, updating the version as part of the COPY.
4. Delete the temporary S3 object using the S3 *DELETE* method. Delete all the messages related to the transaction from the WAL queue using the SQS *DeleteMessage* command.

We include all not-yet-written ancestors of an object in the object's transaction in order to obtain multi-object causal ordering. This ensures that we maintain multi-object causal ordering even if we send packets in parallel to SQS. In contrast, the previous protocols required that we carefully order ancestors and their descendants.

Discussion: The protocol satisfies eventual provenance data-coupling. We cannot provide a stronger guarantee due to the eventual consistency model of the services and due to the fact that we cannot modify the underlying services. Applications that are sensitive to provenance data-coupling can detect inconsistency and can retry again on detecting inconsistency. In prior work, we discuss provenance-aware `read` and `write` system calls [28], which provide an interface that can perform these checks on behalf of the application. Similar to the previous protocols, this protocol maintains eventual multi-object causal ordering, but provides better throughput. Further, queries are executed efficiently as SimpleDB provides rapid, indexed lookup.

5 Evaluation

The goal of our evaluation is to understand the relative merits of the different protocols and their feasibility in practice. To that end, our evaluation has three parts: first, we quantify the storage utilization and data transfer of the protocols independent of the provenance collection framework (Section 5.1), second, we evaluate the efficacy, performance, and cost of the protocols under various workloads (Section 5.2), and third, we evaluate the query performance of the protocols (Section 5.3).

We used the following software configurations for the evaluation:

- **S3fs**: S3fs on a vanilla Linux 2.6.23.17 kernel.
- **P1**: Provenance-Aware S3fs on a PASS kernel (Linux 2.6.23.17 kernel with appropriate modifications), with both provenance and data being recorded on S3.
- **P2**: Provenance-Aware S3fs on a PASS kernel with provenance stored on SimpleDB.

- **P3**: Provenance-Aware S3fs on a PASS kernel with provenance on SimpleDB, with an SQS queue used as a log.

To maximize performance, we implemented the protocols to upload the data objects, their provenance, and ancestral data and provenance in parallel (this violates multi-object causal ordering for P1 and P2).

We used Amazon *EC2 Medium* [15] instances running Fedora 8 to run the benchmarks. The medium instance configuration at the time we ran the experiments was a 32-bit platform with 1.7 GB of memory, 5 EC2 Compute Units (2 virtual cores with 2.5 EC2 Compute Units each), and 350 GB of instance storage. Since one cannot install a custom kernel on EC2 instances, we run the workload benchmarks (Section 5.2) that use the vanilla Linux kernel and the PASS kernel as User Mode Linux (UML) [14] instances with 512MB of RAM on EC2 machines. We had to use medium EC2 machines as the small instances proved to be insufficient to run the PASS kernel as a UML instance. We also ran the benchmarks from one of our local machines. Both the usage models, i.e, running the workloads on local machine and storing data and provenance on the cloud or running the workloads on EC2 machines and storing the data and provenance on the cloud are valid as our protocols are agnostic to the usage model.

We used the following three workloads in our evaluation. Each of the three workloads represents provenance trees of different depths.

CVSROOT nightly backup This workload simulates nightly backups of a CVS repository by extracting nightly snapshots from 30 days of our own repository, creating a `tarball` for each night, and uploading the 30 snapshots to AWS. The provenance tree for this workload is nearly flat with just the program `cp` as the ancestor of the stored archives. The workload is IO intensive, has negligible compute time, and S3fs performs 240 operations under this workload.

Blast This is a biological workload representative of scientific computing workloads. Blast is a tool used to find protein sequences that are closely related in two different species. This workload simulates the typical Blast job observed at NIH [12]. The provenance tree of the workload has a depth of five. The workload has a mix of compute and IO operations and S3fs performs 10,773 operations under this workload.

Challenge This is the workload used in the first and second provenance challenge [35]. The workload simulates an experiment in fMRI imaging. The inputs to the workload are a set of new brain images and a single reference brain image. First, the workload normalizes the images with respect to the reference image. Second, it transforms the image into a new image. Third, it averages all the transformed images into one single image. Fourth, it slices the average image in each of three dimensions to produce a two-dimensional atlas along a plane in the third dimension. Last, it converts the atlas data set into a graphical atlas image. The challenge workload graph is the deepest with maximum path length of eleven. Similar to blast, the workload has a mix of compute and IO operations and S3fs performs 6,179 operations.

We ran each workload at least 5 times for each configuration. The elapsed times we present do not include the commit daemon times for P3 as it operates asynchronously, thus not affecting the elapsed times.

Our evaluation results are AWS-specific as it is currently the only mature cloud service that also provides all the services we need (Note that SimpleDB, as of January 2010, is in public beta). Further, we find that AWS performance is highly variable due to a variety of factors that are not under our control, such as the load on the services, WAN network latencies, and the version of the software used for the service. Further, upgrades to the services seem to continually improve performance over time, thus making reproducibility harder. Due to the variance, we find that results from different days are not comparable. We found that we needed to execute the benchmarks at the same time or within a short time period for the results to be comparable. Even so, we find that at a given time, any of the protocols can perform well due factors such as relative load on the service, proximity of the replica chosen to service requests, etc. We have run a large number of experiments between August 2009 and January 2010. The results we present are those that are most representative of the behavior we observed and best illustrate the trends that we observed repeatedly.

5.1 Microbenchmarks

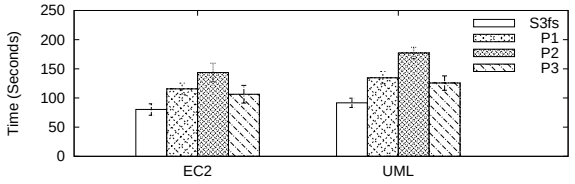


Figure 3: Elapsed times for the microbenchmark on an EC2 instance and on an UML machine running on an EC2 instance.

Our microbenchmarks quantify the throughput obtained by each protocol relative to S3fs. To isolate the protocol throughput from the application and provenance collection overheads, we ran the Blast benchmark

on a unmodified PASS system and captured the provenance. We then built a tool that uploaded the data objects and their provenance to the cloud using each protocol. We ran the microbenchmark on an EC2 instance. Further, to demonstrate that the results in the following section are not an artifact of using UML, we also ran the microbenchmark on a UML instance running on EC2. Figure 3 shows the microbenchmark results.

On EC2, P3, the protocol that best satisfies our properties, also exhibits the lowest overhead (32.6%) and P1 dominates P2. As there is no application time in this microbenchmark, the overheads are relatively high for all the protocols, ranging from 32% for P3 to 78.9% for P2. The UML microbenchmark results follow the pattern we see in the EC2 microbenchmark results, indicating that UML does not change the relative performance of the protocols.

	S3	**SimpleDB**	**SQS**
Time (s)	324.7	537.1	36.2

Table 2: Time taken to upload 50MB of provenance to each of the services.

To understand why the protocols exhibit this relative performance, we ran another benchmark where we uploaded, in parallel, the first 50MB of provenance generated during a Linux compile to each of S3, SimpleDB, and SQS. Table 2 shows the results of this experiment. We find that SQS is dramatically faster than either S3 or SimpleDB and that S3 is significantly faster than SimpleDB. We tried to find the maximum possible throughput by varying the number of concurrent connections to each service. We found that S3 and SQS scaled well as the number of connections increased (we stopped at 150) while SimpleDB peaked at around 40 concurrent connections from a single client host. The numbers in Table 2 used 150 concurrent connections for S3 and SQS and 40 concurrent connections for SimpleDB. Thus, P1 leverages the better parallelism in S3 relative to SimpleDB and outperforms P2. P3 exhibits the best performance as it bundles all its provenance into 8KB chunks uploading them to SQS, the fastest service.

Table 3 shows the data and operation overheads. The data overheads are negligible – all under 1%. In contrast, the overhead in terms of number of operations is quite large, because all the protocols are at least doubling their work, writing both provenance and data. But, as we will see in the next section, operations are not very expensive.

5.2 Workload Overheads

Figure 4 shows the elapsed times for the workload benchmarks run from EC2 instances and from a local machine. We present results collected during September 2009 (Figure 4a) and during December and January 2009-2010 (4b). The Figure consists of 12 sets of results, with each set consisting of 3 individual results that measure the individual protocol overhead relative to S3fs.

	Data Transmitted (MB)	**Operations**
S3fs	713.09	617
P1	715.31 (0.31%)	2287 (270.7%)
P2	716.11 (0.42%)	1235 (100.2%)
P3	716.32 (0.45%)	1337 (116.7%)

Table 3: Data transfer and operation overheads for the protocols. The overheads, shown in parentheses, are relative to S3fs. Protocol P3 numbers do not include the commit daemon. The operation count in the microbenchmark are reduced as we only upload the final results of the computation.

Overall, we observe that the overheads are reasonable – less than 10% for 29 of the 36 individual results shown above. Of the remaining 7 results, 5 of them have an overhead less than 20%. The maximum overhead is 36% for P2 for the challenge workload benchmark run in December/January on EC2. For the same scenario in September, P2 has an overhead of 24.3%.

Incorporating application time into the equation reveals that the relative performance of the different protocols is comparable. At first blush, P3 seems to be the fastest protocol as it performs the best in 8 out of the 12 result sets. However, the error bars on the graphs indicate that the difference is not statistically significant.

We expected the elapsed time for the benchmarks to be greater in the local machine case than in the EC2 case. This was borne out for the nightly backup and challenge workloads. However, the Blast workload ran faster on the local machine than on EC2. We hypothesized that this was caused by an interaction between Blast's memory accesses and the UML's small 512MB memory (512MB is the maximum UML instance memory). We confirmed this by running Blast and the nightly backup benchmark on a native (not UML) EC2 instance. The I/O time for the nightly benchmark increased from 419s on a raw EC2 machine to 528s on a UML EC2 instance. For Blast, the corresponding number increases from 650s to 1322s. The dramatic difference between native EC2 and UML EC2 for the Blast workload was highly suggestive.

Finally, we observe that the elapsed times for all benchmarks except for the nightly local case, have reduced between 4% to 44.5% from September 09 to December 09/January 10. We also observe that P1's performance approaches that of P3 in many of the application benchmarks. As we stated earlier, this is due to various factors that are beyond our control.

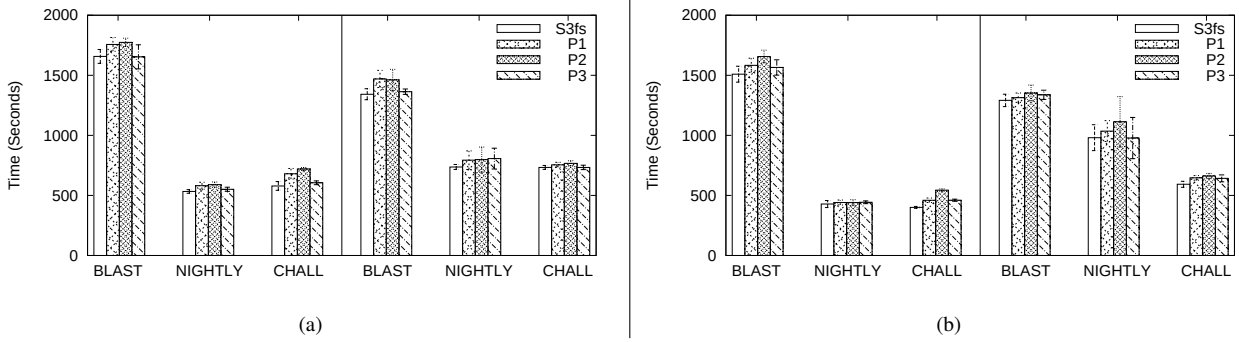


Figure 4: Elapsed times for workload benchmarks. Figure 4a shows the results for the benchmarks from September 2009. Figure 4b shows the results for the benchmarks from December 2009/January 2010. In both graphs, the left half shows elapsed times when the benchmark runs on EC2 instances. The right half shows the elapsed time when running on a local machine.

	Nightly	**Blast**	**Challenge**
S3fs	$1.05	$0.37	$0.27
P1	$1.05	$0.39	$0.29
P2	$1.05	$0.38	$0.29
P3	$1.06	$0.40	$0.30

Table 4: Cost for each benchmark (includes commit daemon cost).

Table 4 shows the cost in USD for each protocol. Overall, we observe the following relationship between protocols: $P3 > P1 >= P2 >= S3fs$. The extra cost required to store provenance in each of the protocols is minimal (compared to S3fs). As expected, P3 is the most expensive due to the operations it performs to log provenance on SQS and then upload provenance to SimpleDB. The cost for P1 and P2 are similar for Nightly and Challenge workloads. For Blast, P2 is cheaper than P1, because P1 needed more operations to store the provenance on S3 than P2 required to store the same provenance on SimpleDB.

5.3 Query performance

To evaluate query performance, we ran the following four queries on the Blast workload provenance:

Q.1 Retrieve all the provenance ever recorded.
Q.2 Given an object, retrieve the provenance of all versions of the object.
Q.3 Find all the files that were directly output by *Blast*.
Q.4 Find all the descendants of files derived from *Blast*.

We chose these queries as they represent varying levels of complexity. The first query is a simple dump of all the provenance. The second query uses an object handle to retrieve all of its provenance but requires no search. The third involves a lookup and a single-level descendant query. The fourth is a full descendant query. Table 5 shows the query results. There are only two different sets of results as P1 uses S3 objects to store provenance, and P2 and P3 use SimpleDB to store provenance, thus having identical query capabilities and performance.

We implement Q.1 in S3 by fetching the list of all S3 provenance objects and then performing a GET for each. Since there are no ordering constraints on when the GET requests are executed, i.e., it is not necessary for any GET to wait for the completion of another request, parallelizing these operations greatly improves performance (as we can see in the Table 5).

In SimpleDB, we execute "SELECT *" to retrieve all the provenance. We implement this as a single request that, due to the limits imposed by SimpleDB, has to be decomposed into several sequential operations, where one operation has to complete before the next one can start, so this request cannot be parallelized. However, the number of SimpleDB round-trips is smaller than in S3, and the query thus executes much more quickly.

In Q.2, the performance is comparable for both S3 and SimpleDB. We implement this query by first issuing a HEAD operation on the object to determine the uuid used to reference its provenance. In S3, we then issue a GET on the provenance object, while in SimpleDB we perform an appropriate SELECT operation. Note that these two operations must be performed sequentially, so the query cannot benefit from parallelism. Because both S3 and SimpleDB perform the HEAD operation, the performance is comparable.

In Q.3 and Q.4, we need to first find records (items) of processes that correspond to the multiple executions of *Blast*. This translates into looking up all items that satisfy a certain property. In S3, this requires a scan of all provenance objects. We implemented these two queries in S3 by retrieving all provenance objects and then processing the query locally. SimpleDB is more ef-

Query	S3 (P1)				SimpleDB (P2, P3)			
	Time (s)		MB Transferred	Ops.	Time (s)		MB Transferred	Ops.
	Sequential	Parallel			Sequential	Parallel		
Q.1	48.57	7.04	2.95	1671	0.83	–	2.05	13
Q.2	0.060	–	0.0015	2	0.037	–	0.008	2
Q.3	48.57	7.04	2.95	1671	0.82	0.34	0.11	37
Q.4	48.57	7.04	2.95	1671	1.86	0.72	0.19	87

Table 5: Query performance. The table shows the time taken to complete the queries, the total data transferred, and the total number of executed operations. The table shows the times for both sequential and parallel execution of the query. In both cases, the number of operations and the data transferred was the same. For Q.2, the values shown are the average time taken per object.

ficient for Q.3 and Q.4 as it indexes all the attributes in the database. Hence, for Q.3 and Q.4 in SimpleDB, we first issue a SELECT to find all items corresponding to *Blast*. We then issue a set of SELECT queries to find the names of all the items that reference the *Blast* items retrieved in the previous call. For Q.4, we have to repeat the second step recursively until we have located all the descendants. As we can see from the results, SimpleDB is an order of magnitude faster as it can retrieve data more selectively. Further, the performance gap between S3 and SimpleDB is bound to grow larger as more objects are involved.

5.4 Summary

All three protocols have low cost and data transfer overheads. The workload overheads were less than 10% over S3fs for all protocols in the majority of the cases. Our microbenchmarks show that P3, our most robust protocol, is the best performing. But, when application overheads are included, all protocols are within statistical error. Thus users can select the best protocol best suited for their needs, without performance penalty.

6 Related Work

Provenance in distributed workflow-based and grid environments has been explored by several prior research projects [11, 17, 21, 40]. There are also systems that track application-specific data to be able to regenerate data [23] or reproduce experiments [16]. All prior work assumes the ability to alter the underlying system components, as opposed to having to make due with a given infrastructure as we do here. We develop a provenance solution atop an infrastructure over which we have no control. However, we complement this prior work, and our protocols can be used to move the provenance collected by the above frameworks to the cloud.

Branthner et. al. [9] explore using S3 as a backend for a database. They use SQS as a log to ensure atomic updates to the database, similar to the mechanism we use in P3. While the mechanisms are similar, this work and Branthner et. al. address different research questions. Brantner et. al. use the mechanism to coordinate updates to a single service. We use the mechanism to provide consistency between two services, S3 and SimpleDB.

In prior work [31], we explored the challenges of storing provenance in the cloud, outlined protocols, and performed a rudimentary analysis of the protocols. This work follows on where that work left off, i.e., we implement and evaluate the protocols. Some tweaks were necessary to realize the protocols in practice. For example, for P1, we had originally intended to store the provenance as metadata of the S3 object, but this does not satisfy the data independent persistence property.

Hasan et. al. [22] discuss cryptographic mechanisms to protect provenance from tampering. Juels et. al [24] and Ateniese et. al. [4] present schemes that allow users to efficiently verify that a provider can produce a stored file. These research projects are complementary to our work and we can leverage them to verify that malicious users and servers have not tampered provenance on the cloud.

7 Native Cloud Provenance: Research Challenges

This work has focused on storing and accessing provenance on current cloud offerings. In the current scheme where provenance and data are stored on separate services, however, providers have no means to link the provenance of an object to its data. Providing native support for provenance on cloud stores enables providers to relate provenance to its data, allowing the providers to leverage the provenance for their benefit [32]. For example, the graph structure in provenance can provide service providers with hints for object replication. As more data moves to the cloud, providers will need to provide search capabilities to users. As outlined previously (Section 2.2), provenance can play a crucial role in improving search quality. Cloud providers could also allow users to chose between storing data and regenerating data on demand, if the provenance of data were available to them [3].

Building native support for the cloud presents a number of challenges in addition to the issues that arise in

building large scale distributed systems. We discuss some of these research challenges next.

System Architecture A native provenance store has to support both the object storage requirements of data and the database functionality requirements of provenance. The simplest approach is to obviously store the provenance and the data in two separate services. However, one needs to to co-ordinate updates across the two services. To provide strong provenance data-coupling using an external co-ordination service, the underlying services have to export a transactional interface. However, a fully transactional system is not feasible at the scales at which the cloud operates. Finding a middleground between the two extremes and the cost of each approach (the naive approach, fully transactional, and a possible middleground) is an open research challenge.

Security Provenance can potentially contain sensitive information. The fundamental issue is that provenance and the data it describes do not necessarily share the same access control. For example, consider a report generated by aggregating the health information of patients suffering a certain ailment. While the report (the data) can be accessible to the public, the files that were used to generate the report (the provenance) must not be. Provenance security is an open problem that is being explored by multiple research groups [10]. Providers need to take these issues into consideration while extending their service to support provenance.

Provenance Storage The semi-structured data model, imported by SimpleDB and Azure Table, is appropriate for storing provenance graphs. These services, however, are not necessarily optimized to store provenance graphs. Recently, databases such as Neo4j [33], have been designed from the ground-up for storing graphs. Exploring if a data service designed from the ground-up for storing provenance is more efficient in terms of performance and cost compared to a generic database service is an interesting avenue for future work.

Learning Models As we stated above, cloud providers can take advantage of provenance in a variety of ways. However, for each particular application, a particular subset of provenance has to be extracted or a particular type of generalization has to be made across all objects. For some applications, a simple pattern matching approach might be sufficient and for other applications, sophisticated machine learning mechanisms might be necessary. The models necessary to extract the necessary data for each application is an open question.

Processing Provenance Graphs The models above need to process the provenance graph to extract the necessary information. However, there are currently no general purpose graph processing systems available. MapReduce is one mechanism that is generally used to process graphs. Pregel [26], based on Bulk Synchronous Parallel model, is another approach that is currently being developed. How the two mechanisms compare with each other for graph workloads is a study worth undertaking.

Transparent Provenance Collection This work expects and trusts users to supply provenance. The provenance graph supplied by users is rich as it consists of process information. Without support from users, the cloud can automatically infer diluted provenance, i.e., provenance minus process information. In this provenance graph, all the processes from a single host will be represented by a single node representing the host. What subset of the provenance applications can be driven by this diluted graph?

Economics Providing native support for provenance increases the cost to the provider in terms of storage, CPU, and network bandwidth. Prior to embarking on building a native cloud store, an economic analysis that justifies that the extra cost of provenance is necessary. To this end, we need to design appropriate economic models and evaluate the cost of storing provenance.

8 Conclusions

The cloud is poised to become the next generation computing environment, and we have shown that we can add provenance to cloud storage in several ways. Our evaluation shows that all three protocols have reasonable overhead in terms of time to execute and minimal financial overhead. Further, our most robust protocol, which provides all the properties we outline, performs as well, if not better, than the other protocols, making it one of those rare occasions where we need not make compromises to achieve our objectives. We can construct a fully functional and performant provenance system for the cloud using off-the shelf cloud components.

The web, which is the most widely used medium for sharing data, does not provide data provenance. The cloud, however, is still in its infancy, and can easily incorporate provenance now. We can deploy these kinds of services with systems today, but it is worth investigating the cost, efficacy, and feasibility of offering provenance as a native cloud service as well.

Acknowledgments We thank Kim Keeton, Bill Bolosky, Keith Smith, Erez Zadok, James Hamilton, and Nick Murphy for their feedback on early drafts of the paper. We thank Matt Welsh for discussions at early stages of the project. We thank Jason Flinn, our shepherd, for repeated careful and thoughtful reviews of our paper. We thank Kurt Messersmith from Amazon Web Services for providing us with credits to run the experiments in the paper. We thank the FAST reviewers for the valuable feedback they provided. This work was partially made possible thanks to NSF grant CNS-0614784.

References

[1] Pubchem. http://pubchem.ncbi.nlm.nih.gov/.

[2] Genbank. *Nucleic Acids Research 36 (Database Issue)* (January 2008).

[3] ADAMS, I., LONG, D. D. E., MILLER, E. L., PASUPATHY, S., AND STORER, M. W. Maximizing efficiency by trading storage for computation.

[4] ATENIESE, G., BURNS, R., CURTMOLA, R., HERRING, J., KISSNER, L., PETERSON, Z., AND SONG, D. Provable data possession at untrusted stores. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 598–609.

[5] Amazon Web Services. http://aws.amazon.com.

[6] Windows Azure Blob. http://go.microsoft.com/fwlink/?LinkId=153400.

[7] Windows Azure Queue. http://go.microsoft.com/fwlink/?LinkId=153402.

[8] Windows Azure Table. http://go.microsoft.com/fwlink/?LinkId=153401.

[9] BRANTNER, M., FLORESCU, D., GRAF, D., KOSSMANN, D., AND KRASKA, T. Building a database on S3. In *SIGMOD '08: Proceedings of the 2008 ACM SIGMOD international conference on Management of data* (New York, NY, USA, 2008), ACM, pp. 251–264.

[10] BRAUN, U., SHINNAR, A., AND SELTZER, M. Securing Provenance. In *Proceedings of HotSec 2008* (July 2008).

[11] CHEN, Z., AND MOREAU, L. Implementation and evaluation of a protocol for recording process documentation in the presence of failures. In *Proceedings of Second International Provenance and Annotation Workshop (IPAW'08)*.

[12] COULOURIS, G. Blast benchmarks. http://fiehnlab.ucdavis.edu/staff/kind/Collector/Benchmark/Blast_Benchmark.

[13] DAGDIGIAN, C. Plenery Keynote: Bio.IT World. http://blog.bioteam.net/wp-content/uploads/2009/04/bioitworld-2009-keynote-cdagdigian.pdf.

[14] DIKE, J. User-mode linux. In *Proceedings of the 5th Annual Linux Showcase & Conference* (Oakland, California, USA, 2001).

[15] Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2.

[16] EIDE, E., STOLLER, L., AND LEPREAU, J. An experimentation workbench for replayable networking research. In *4th USENIX Symposium on Networked Systems Design & Implementation* (2007).

[17] FOSTER, I., VOECKLER, J., WILDE, M., AND ZHAO, Y. The Virtual Data Grid: A New Model and Architecture for Data-Intensive Collaboration. In *CIDR* (Asilomar, CA, Jan. 2003).

[18] FREW, J., METZGER, D., AND SLAUGHTER, P. Automatic capture and reconstruction of computational provenance. *Concurrency and Computation: Practice and Experience 20* (April 2008), 485–496.

[19] Filesystem in userspace. http://fuse.sourceforge.net/.

[20] GRAY, J., SLUTZ, D., SZALAY, A., THAKAR, A., VANDENBERG, J., KUNSZT, P., AND STOUGHTON, C. Data Mining the SDSS SkyServer Database. Research Report MSR-TR-2002-01, Microsoft Research, January 2002.

[21] GROTH, P., MOREAU, L., AND LUCK, M. Formalising a protocol for recording provenance in grids. In *Proceedings of the UK OST e-Science Third All Hands Meeting 2004 (AHM'04)* (Nottingham, UK, Sept. 2004). Accepted for publication.

[22] HASAN, R., SION, R., AND WINSLETT, M. The Case of the Fake Picasso: Preventing History Forgery with Secure Provenance. In *FAST* (2009).

[23] HEYDON, A., LEVIN, R., MANN, T., AND YU, Y. *Software Configuration Management Using Vesta*. Monographs in Computer Science, Springer, 2006.

[24] JUELS, A., AND KALISKI, JR., B. S. Pors: proofs of retrievability for large files. In *CCS '07: Proceedings of the 14th ACM conference on Computer and communications security* (New York, NY, USA, 2007), ACM, pp. 584–597.

[25] The LINQ project. http://msdn.microsoft.com/en-us/vcsharp/aa904594.aspx.

[26] MALEWICZ, G., AUSTERN, M. H., BIK, A. J., DEHNERT, J. C., HORN, I., LEISER, N., AND CZAJKOWSKI, G. Pregel: a system for large-scale graph processing. In *PODC '09: Proceedings of the 28th ACM symposium on Principles of distributed computing* (New York, NY, USA, 2009), ACM, pp. 6–6.

[27] MARGO, D. W., AND SELTZER, M. The case for browser provenance. In *1st Workshop on the Theory and Practice of Provenance* (2009).

[28] MUNISWAMY-REDDY, K.-K., BRAUN, U., HOLLAND, D. A., MACKO, P., MACLEAN, D., MARGO, D., SELTZER, M., AND SMOGOR, R. Layering in provenance systems. In *Proceedings of the 2009 USENIX Annual Technical Conference*.

[29] MUNISWAMY-REDDY, K.-K., AND HOLLAND, D. A. Causality-Based Versioning. In *Proceedings of the 7th USENIX Conference on File and Storage Technologies* (Feb 2009).

[30] MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., BRAUN, U., AND SELTZER, M. Provenance-aware storage systems. In *Proceedings of the 2006 USENIX Annual Technical Conference*.

[31] MUNISWAMY-REDDY, K.-K., MACKO, P., AND SELTZER, M. Making a cloud provenance-aware. In *1st Workshop on the Theory and Practice of Provenance* (2009).

[32] MUNISWAMY-REDDY, K.-K., AND SELTZER, M. Provenance as first-class cloud data. In *3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware (LADIS'09)* (2009).

[33] Neo4j, the graph database. http://neo4j.org/.

[34] Provenance aware service oriented architecture. http://twiki.pasoa.ecs.soton.ac.uk/bin/view/PASOA/WebHome.

[35] The First Provenance Challenge. http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge.

[36] RIZUN, R. S3fs: FUSE-based file system backed by Amazon S3. http://code.google.com/p/s3fs/wiki/FuseOverAmazon.

[37] Amazon Simple Storage Service (Amazon S3). http://aws.amazon.com/s3.

[38] Amazon SimpleDB. http://aws.amazon.com/simpledb.

[39] SHAH, S., SOULES, C. A. N., GANGER, G. R., AND NOBLE, B. D. Using provenance to aid in personal file search. In *Proceedings of the USENIX Annual Technical Conference* (2007).

[40] SIMMHAN, Y. L., PLALE, B., AND GANNON, D. A framework for collecting provenance in data-centric scientific workflows. In *ICWS '06: Proceedings of the IEEE International Conference on Web Services* (2006).

[41] Amazon Simple Queue Service (SQS). http://aws.amazon.com/sqs.

[42] ZHAO, J., GOBLE, C.AND GREENWOOD, M., WROE, C., AND STEVENS, R. Annotating, linking and browsing provenance logs for e-science.