Making a Cloud Provenance-Aware

Making a Cloud Provenance-Aware

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

Abstract

The advent of cloud computing provides a cheap and convenient mechanism for scientists to share data. The utility of such data is obviously enhanced when the provenance of the data is also available. The cloud, while convenient for storing data, is not designed for storing and querying provenance. In this paper, we present desirable properties for distributed provenance storage systems and present design alternatives for storing data and provenance on Amazon's popular Web Services platform (AWS). We evaluate the properties satisfied by each approach and analyze the cost of storing and querying provenance in each approach.

1 Introduction

Scientific computing in fields such as astronomy, physics, climate science, etc. is highly data-intensive. Recently, large-scale scientific computing endeavors required scientists to acquire and administer large data repositories [16, 2, 17, 7] and scientists found themselves having to take on computing challenges in grid computing and distributed storage. The advent of cloud computing introduces the possibility to dramatically change this landscape. The vision of cloud computing for science is that scientists can focus on their core competencies (i.e., conducting scientific research) while large, online service providers focus on their core competency. Not only does this relegate tasks to the organizations best equipped to deal with them, it provides more graceful storage provisioning and growth. Scientists who cannot predict, apriori, their exact storage requirements can simply purchase more storage capacity from their providers, avoiding large initial investments or hard-to-predict stepwise cost functions. Ultimately, this reduces the per-byte cost to the user. Further, since clouds are designed to be highly available, scientists need not build their own highly available repositories nor worry about backing up their massive data sets.

The following is an example of this new architecture for data-intensive science. Data from the US Census databases are released on the cloud by US Census Bureau [1]. Scientists who wish to analyze this data for trends can download the data set to their local compute grid, process it, and then upload the results back to the cloud, easily sharing their results with fellow researchers. In this model, scientists need only enough local storage for their current experiments.

Data stored in the cloud has the potential to be a valuable shared resource. Its value, however, can be greatly enhanced if the cloud stores both data and the provenance of that data. For example, provenance that includes (or references) the complete ancestry of a data set reveals the source of the data as well as the exact processing applied to that source. Imagine that a researcher discovers that a particular version of a widely-used analysis tool is flawed. She can identify all data sets affected by the flawed software by querying the provenance. Similarly, if the community discovers that all data produced by a particular piece of hardware is corrupt, the provenance facilitates mapping that hardware into all affected data sets. Consider the efforts of one group attempting to reproduce the results of another research group. If the reproduction does not yield identical results, comparing the provenance will shed insight into the differences in the experiment. Our goal is to design and analyze alternatives for incorporating provenance directly into a cloud computing infrastructure.

Our vision has a parallel in grid computing in that the grid is designed to process data and has its own metadata services [6] that can be used to record provenance. The grid is different from the cloud as follows. In a grid, machines are scheduled in a batch mode based on availability, i.e., if systems are available, then the user can run his computing, else he has to wait for machines to be available. The cloud, on the other hand, is designed to be highly available as resources are created or allocated on demand. The difference in availability leads to different

fault tolerance and consistency semantics in grid applications and cloud applications, thus motivating the need for our work.

We focus our analysis on the cloud computing services offered by Amazon Web Services (AWS) as they are the most mature products on the market and are reasonably well documented in the literature. AWS services are designed to be highly available and scalable. However, they sacrifice perfect consistency and provide eventual consistency. Eventual consistency means that at any instant in time, data may be inconsistent across replicas, but over time, every update will, in fact, propagate to every replica. Eventual consistency makes it difficult to provide strong guarantees about provenance and the data it describes.

Our system model assumes that we use a Provenance Aware Storage System (PASS) [11] augmented to use AWS for backend storage. We present three design alternatives for extending the reach of PASS to the cloud. Our first architecture uses only the Amazon Simple Storage Service (S3). Our second architecture uses both S3 and Amazon SimpleDB. Our third architecture uses S3, SimpleDB, and the Amazon Simple Queueing Service (SQS). We begin by outlining the requirements of a provenance-aware cloud and then evaluate each architecture with respect to those requirements. We then analyze the cost of storing and querying provenance in each architecture.

The contributions of the paper are:

1. We make the case for provenance-aware cloud storage and define the required properties of such a system.
2. We present three provenance-aware cloud storage design alternatives all of which build upon Amazon's S3, evaluating each architecture with respect to the requirements we established.
3. We compare the three alternatives in terms of storage and query performance.

The rest of the paper is organized as follows. In section 2, we provide background on AWS services, PASS, and discuss our assumed usage model. In section 3, we discuss requirements for a provenance-aware cloud storage system. In section 4, we discuss three system design alternatives for constructing a provenance-aware cloud. In section 5, we analyze the three design alternatives. We discuss related work in section 6. We conclude in section 7.

2 Background

In this section, we introduce the Amazon Web Services (AWS) used in our design and analysis[1]. We then discuss PASS and conclude the section by discussing the usage model that we assume for the rest of the paper.

2.1 Simple Storage Service (S3)

S3 [12] is Amazon's storage service. It is an object store where the size of the objects can range from 1 byte to 5GB. Each object is identified by a unique URI. Clients access S3 objects using either a SOAP- or REST-based API. The object operations that are relevant to this work are: *PUT*, *GET*, *HEAD*, *COPY*, and *DELETE*. The PUT operation stores an object on S3, overwriting an object if it already exists. With each object, clients can store up to 2KB of metadata, which is also specified as a part of the PUT operation. A client can retrieve either complete S3 objects or partial objects (described by a byte range) via the GET API. The HEAD operation retrieves only the metadata part of an object. The COPY and DELETE operations, as the names suggest, create copies of objects or delete objects.

The cost of using S3 is based on the amount of data transferred to and from S3, the amount of storage space, and the number of operations performed. For example, it costs USD 0.15 per GB for the first 50 TB / month of storage used, USD 0.10 per GB for all data transferred in, USD 0.17 per GB for the first 10 TB / month for data transferred out, USD 0.01 for every 1,000 PUT, COPY, POST, or LIST requests, and USD 0.01 for 10,000 GET (and other) requests.

Amazon S3, like other AWS components, provides eventual consistency. If a client performs a GET operation on object right after a PUT, the client might receive an older copy of the object as the replica from which S3 chooses to return the object might not yet have received the latest update. If two clients update the same object concurrently via a PUT, the last PUT operation is retained. Again, for an ephemeral time after the PUT, a GET operation might return either of the two writes to the client as the replica that is chosen to process the GET might not have the last PUT operation propagated to it.

2.2 SimpleDB

Amazon SimpleDB [13] is a service that provides the functionality of indexing and querying data. SimpleDB's data model consists of items that are described by attribute-value pairs. Items are synonymous to rows in traditional databases. Each item can have a maximum of 256 attributes-value pairs and can have multiple attributes with the same name (for example, an item can have two phone attributes with different values). Each attribute name and value can be as large as 1KB. SimpleDB automatically indexes data as it is inserted. New data can be inserted into SimpleDB or ex-

isting data can be modified using the *PutAttributes* operation. SimpleDB provides three primitives for querying data: *Query*, *QueryWithAttributes*, and *SELECT*. Query returns items that match predicates specified in the query expression. QueryWithAttributes also returns the attribute-value pairs associated with the items that match the predicate. Further, one can also specify a subset of attributes that are to be returned with the results. SELECT provides functionality similar to QueryWithAttributes, with the main difference being that the queries are expressed in the standard SQL form. Attributes can be deleted using a *DeleteAttributes* call. All data, items and attribute-value pairs, are stored in a *Domain*. Each of the operations that manipulate items must specify the domain in which to perform the operations. Amazon provides REST and SOAP interfaces for these operations. The pricing model of SimpleDB is based on the amount of data transferred in and out, the amount of data stored, and the number of machine hours consumed for performing operations.

Like S3, SimpleDB provides eventual consistency. An item inserted might not be returned in a query that is run immediately after the insert. In addition, SimpleDB is idempotent, i.e., running PutAttributes multiple times with the same attributes or running DeleteAttributes multiple times on the same item or attributes will not generate an error.

2.3 Simple Queueing Service (SQS)

SQS [15] is a distributed messaging system that allows users to exchange messages between various distributed components in their systems. Clients can create queues that are identified by a URL that is unique among all of the user's queues. Clients can enqueue messages using the *SendMessage* operation. SQS imposes an 8KB limit on the size of the message and the characters in the message are restricted to Unicode characters. Clients can receive messages on the queue using *ReceiveMessage*. Clients can request a maximum of 10 messages to be returned as a result of a ReceiveMessage. Messages can be deleted from the queue using *DeleteMessage* with a handle returned on ReceiveMessage. Clients can interact with SQS using either an HTTP or a SOAP interface. The pricing model of SQS is based on the number of operations, the amount of data transferred in and out, and the amount of data stored, as in S3.

As all other AWS services, SQS is designed to be available and scalable and is eventually consistent. SQS samples a set of machines on a ReceiveMessage, returning only the messages on those machines. The clients need to repeat these requests until they receive all the necessary messages. SQS provides a best effort in-order delivery of messages. SQS provides a *GetQueueAttributes:ApproximateNumberOfMessages* operation that allows a client to estimate the approximate number of messages on a queue. Due to eventual consistency, the result of this operation is an approximation. Another important concept is that of the *visibility timeout*. When a message is returned to a client via ReceiveMessage, SQS blocks the message from other clients for a configurable amount of time. SQS does not delete the message after delivering a message to a client as there is no guarantee that the client actually received the message. However, this leaves the message open for other clients to simultaneously process it. In order to prevent such a situation, SQS blocks the message for a period of time. If the client processing the message does not call a DeleteMessage by the end of the visibility timeout, the message is visible again. Thus SQS ensures that there is only one client processing a message at a single point of time[2].

2.4 PASS

A Provenance-Aware Storage System (PASS) [11] is a storage system we built that collects provenance for objects stored on it. PASS observes system calls that applications make and captures relationships between objects. For example, when a process issues a `read` system call, PASS creates a provenance record stating that the process depends upon the file being read. When that process then issues a `write` system call, PASS creates a record stating that the written file depends upon the process that wrote it. Further PASS stores data and provenance records together, ensuring that the provenance is consistent with the data. Since persistent objects (files) are related to each other via data flows through transient objects like processes and pipes, PASS also records the provenance of the transient objects. PASS also versions objects appropriately in order to preserve causality.

2.5 Usage Model

Amazon's storage service, S3, is appropriate to store large files that are often read and rarely updated. This model is not appropriate for applications that concurrently update data, and precludes some of the problems associated with distributed systems and provenance systems (such as provenance cycles [4]). However, multiple clients can concurrently update different objects at the same time.

Amazon hosts public data sets for free as Amazon Elastic Block Store (Amazon EBS) snapshots [1]. EBS is a service that provides block level storage volumes for use with Amazon's virtual machine service (Elastic Compute Cloud (EC2)). The disadvantage of using EBS volumes is that users have to clone the whole EBS volume even if they are interested only in a part of the data

set. Making data available as S3 objects allows users to selectively copy the data they need. Furthermore, the EBS sharing model requires that users be able to access the file system format that was used in the EBS volume. S3 objects are more convenient for sharing as users can download data using HTTP. Hence, we assume that users share data directly as S3 objects. The discussions in the rest of the paper assume the model that we have established here.

3 Provenance System Properties

There are three properties of provenance systems that make their provenance truly useful. In this section, we introduce these properties and explain their value.

1. **Read Correctness** This property states that a read must return objects that have consistent provenance. A system should not return an object without provenance or with an out-dated provenance. This property allows users to verify the provenance of an object before using it. For example, a scientist might know that data sets generated from a particular experiment run were flawed. Before using any data, the scientist should verify that the data is not from the flawed run. Read Correctness in turn requires two sub-properties to hold: *atomicity* and *consistency*.

   (a) **Atomicity** The atomicity property states that provenance must be recorded atomically with the data it describes, i.e., both data and provenance should be recorded or neither should be recorded. Protocols that are not designed to be atomic violate the read correctness property. Consider the case where a client records data and crashes before recording the provenance. This violates read correctness property and may render the data set useless. Now consider the case where a client records provenance and crashes before the data on the server is updated. This can cause an old version of data to be interpreted as being a new version (because the provenance says it is). In current systems, atomicity is violated because they are designed to provide either storage or query, but not both. If we continue to use disparate systems to store provenance and data, the protocols must be designed to support atomicity.

   (b) **Consistency** The consistency property states that data and provenance must match. This differs from atomicity in that atomicity pertains to provenance storage, whereas consistency pertains to provenance and data retrieval. For example, consistency can be violated due to AWS's eventual consistency model. If the provenance and data are stored on different services (say S3 and SimpleDB), S3 might return an older version of data while SimpleDB returns the latest provenance (or vice versa), thus violating consistency. Consistency violation leads to read violations that can result in a scientist using the wrong data set.

2. **Causal Ordering** This property entails that the provenance and data of an ancestor object must be recorded in the provenance system. If the provenance of an object refers to an ancestor object and neither the provenance nor data of the ancestor have been recorded, then the object is disconnected from its provenance tree, and its provenance is therefore incomplete. Causal Ordering violations occur when the descendant and its provenance are evicted to persistent store before the ancestor, and the system crashes before the ancestor's provenance and data are recorded. We are designing systems to accept a weaker form of the property called *Eventual Causal Ordering* in which the ancestor object and its provenance might not be available at the same instant that a descendant becomes persistent, but will eventually be available.

3. **Efficient Query** Provenance must be accessible to users who want to verify properties of their data or simply be aware of its lineage. Thus, provenance systems must support efficient query. If a system stores provenance, but that provenance is not readily accessible, the provenance is of questionable value.

We consider the three properties described above as requirements. Other properties, for example *idempotency* of operations, may become necessary depending on the system architecture. Some properties are precluded by the usage model. For example, read correctness also requires *isolation* to hold. Isolation implies that other processes cannot see data in an intermediate state. However, such scenarios do not arise because our usage model precludes concurrent access to the same object.

4 Design Alternatives

In this section, we present three architectures that explore the design space in storing provenance with data in S3. We start with an architecture that stores data and provenance directly in S3. We then present an architecture that stores the data in S3 and the provenance in SimpleDB to allow for efficient provenance query. Our last architecture stores the data in S3 and provenance in SimpleDB,

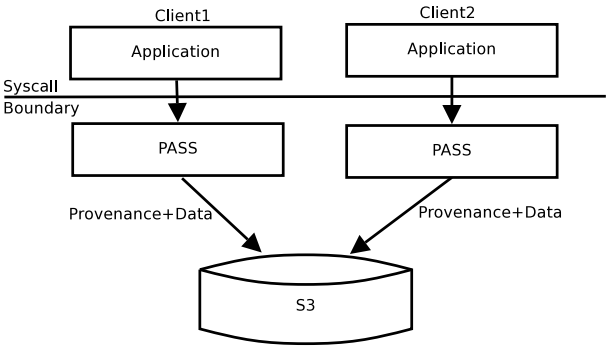

Figure 1: PASS with S3 as the storage substrate

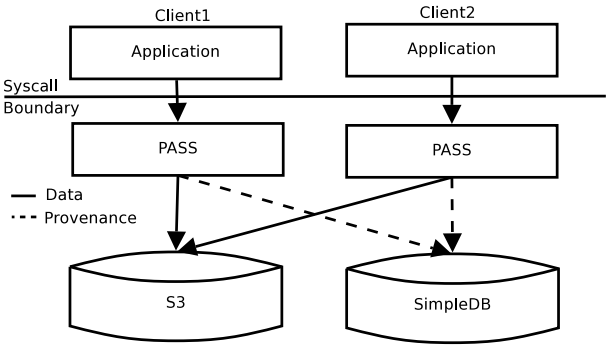

Figure 2: PASS layered on S3 and SimpleDB. Data is stored in S3 and provenance in SimpleDB.

but uses SQS to provide read correctness. For each architecture, we present a protocol that is used to store provenance and discuss the advantages and limitations of storing provenance using the architecture.

4.1 Standalone S3

Architecture: Figure 1 shows the S3-only architecture. We have a PASS system using S3 as the storage layer, storing both provenance and data. Each PASS file maps to an S3 object. We store an object's provenance as S3 metadata. We mirror the file system in a local cache directory, reducing traffic to S3. We also cache provenance locally in a file hidden from the user. When the application issues a `close` on a file, we send both the file and its provenance to S3.

We considered storing the provenance in a separate database (like MySQL or Berkeley DB) that exists on S3 instead of storing provenance records with each individual object. We discarded this approach for the following reason. For this approach to be feasible, we need to cache the provenance database at the client, else we need to download (using GET) and upload (using PUT) the entire database to and from the client for all provenance database updates. However, caching the database at the client is infeasible as the database can become corrupt if two clients pick up the same version of the database and update it independently.

Protocol: The protocol for storing provenance and data is simple. On a file close, we perform the following operations:

1. Read the data cache file and provenance cache file of the object.

2. Convert the provenance into attribute value pairs, as required by S3.

3. Initiate a PUT call that has as arguments, the object, the size of the object, and the provenance attribute value pairs.

Discussion: This architecture provides read correctness and maintains eventual causal ordering. Since the data and provenance are shipped to S3 using a single PUT call, provenance is stored atomically with the data, i.e., either both provenance and data are stored or they are both not stored. Provenance is consistent with data as the data and provenance are stored together using a single PUT call. This ensures read correctness. The eventual causal ordering is maintained as long as we send all the ancestors of an object and their provenance to the S3 before sending the provenance of an object to S3. Querying, however, can be inefficient. The only way to read provenance is by issuing a HEAD call on an object. However, if we do not know the exact object whose provenance we seek, then we might need to iterate over the provenance of every object in the repository, which is so inefficient as to be impractical. We are also limited by S3's 2KB metadata limit. This is a serious limitation in environments where the provenance of a process exceeds the 2KB limit (which we see regularly). We might address this problem by storing provenance overflowing the 2KB limit in separate S3 objects. However, this introduces read correctness challenges and only worsens the query problem.

4.2 S3 with SimpleDB

Architecture: Figure 2 shows the combined S3 and SimpleDB architecture. PASS stores each file as an S3 object and sends the corresponding provenance to SimpleDB. We considered storing both the data and the provenance in SimpleDB, but that is not feasible, because SimpleDB values are limited to 1 KB. The

provenance of each version of a file is stored as a SimpleDB item. The item name is the concatenation of the object name and the version. Each provenance record is converted to an attribute-value pair that describes the item. For example, if version 2 of an object named *foo* has two provenance records (input, bar:2) and (type, file). This is represented as follows: (ItemName=foo_2;attribute-name=input,attribute-value=bar:2;attribute-name=type,attribute-value=file). We store any provenance values larger than the 1KB SimpleDB limit as separate S3 objects, referenced from SimpleDB.

As in the previous architecture, we cache files and the provenance in a local cache directory. When the application issues a `close` on a file, we send the data to S3 and the provenance to SimpleDB. Apart from the provenance that has been collected by PASS, we store one additional record in SimpleDB: the MD5sum of the file contents concatenated with a nonce (the nonce is typically the file version). We use the MD5sum and the nonce to ensure consistency of the provenance and data.

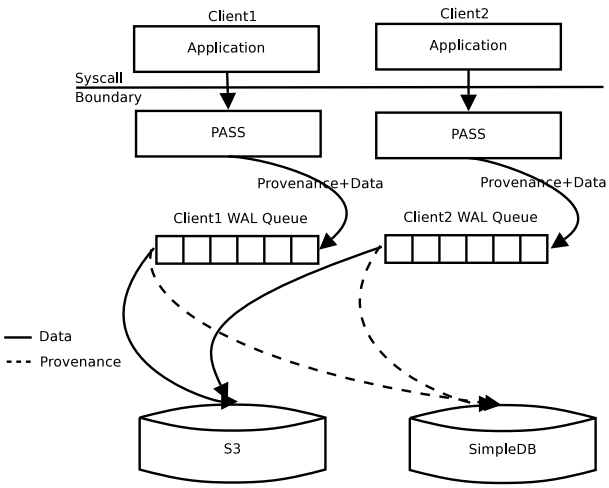

Figure 3: PASS layered on S3 and SimpleDB. SQS is used to provide atomicity.

Protocol: On a file close, the following operations are performed:

1. Read the data cache file and provenance cache file of the object.

2. Convert each provenance record into attribute-value pairs and construct one big provenance record for the file. If any of the values are larger than the 1KB limit that SimpleDB imposes, store the object in S3 and update the attribute-value pair to contain a pointer to the S3 object. We add an additional attribute-value pair that has the MD5sum of the data file concatenated with a nonce.

3. Store the record in SimpleDB by issuing a *PutAttributes* call. Since SimpleDB allows us to store only 100 attributes per call, we might have to issue multiple PutAttributes calls to store all the provenance.

4. Initiate a PUT call to S3 that stores the object. In the same PUT call, set as metadata, the nonce that was used in computing the MD5sum record stored in SimpleDB.

Discussion: This architecture provides efficient provenance queries and maintains causal ordering. It however, does not provide read correctness. Efficient provenance query results from: First, our ability to retrieve provenance at finer granularity from SimpleDB. Second, SimpleDB's automatic indexing of the records stored in it. This architecture maintains eventual causal ordering, in the same manner as the previous architecture.

However, this architecture violates Read correctness as there is no mechanism to maintain atomicity, although we do retain consistency via the MD5sums. 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 can detect this, however, using the MD5sum provenance value in the database, comparing it to the MD5sum of the data and the nonce. If they are not consistent, we can reissue the query, retrieving data from S3 until we get consistent provenance and data. The MD5sum of the data itself (without the nonce) is sufficient to detect inconsistency in most cases, except when a file is overwritten with the same data. In such cases, new provenance will be generated but the MD5sum of the data will be the same as before.

However, there are scenarios that violate atomicity. Consider the following scenario: A client crashes after storing the provenance of object on SimpleDB but before storing the object on S3. Clearly atomicity is violated here as provenance is recorded but not the data. On restart, the client could recover by scanning SimpleDB for "orphan provenance" and remove provenance of objects that do not exist. However, this is an inelegant solution as it involves a scan of the entire SimpleDB domain to clean up the orphans. The next architecture offers a better solution.

4.3 S3 with SimpleDB and SQS

Architecture: Figure 3 shows our final architecture. As in the previous architecture, we store data in S3 and provenance in SimpleDB. We retain the MD5sum and nonce from the previous architecture. We also cache files and the provenance in a local cache directory. This architecture differs from the previous one in that it uses SQS queues and transactions to ensure atomicity and as a result, read consistency. Each client has an SQS queue that it uses as a write-ahead log (WAL). The technique of using SQS queues to ensure atomicity is inspired by the work of Branthner et. al. [3] that explores using S3 as a backend for a database.

The basic protocol is as follows: When an application at the client issues a `close` on a file, the client starts a transaction, recording the data and provenance on the WAL queue. We tag each record with the transaction id. Once the client successfully records provenance and data on the WAL queue, it issues a commit record to indicate that it has successfully completed the transaction.

A separate daemon on the client, the *commit daemon*, reads the log records from transactions that have a commit record and pushes them to S3 and SimpleDB appropriately. After transmitting all the operations for a transaction, the commit daemon deletes the log records in the WAL queue. If the client crashes before it can log all the information to the WAL queue, i.e., before it can commit a transaction, the commit daemon ignores these records.

The messages on SQS cannot exceed 8KB, hence we cannot directly record large data items on the WAL queue. We could split large objects into 8KB chunks and store them on the WAL log, but this is quite inefficient. Instead, we store the file as a temporary S3 object, recording a pointer to the temporary object in the WAL queue. The commit daemon, while processing the WAL queue entries, copies the temporary object to its correct name and then deletes the temporary file. Once items are in the WAL queue, they will eventually be stored in S3 or SimpleDB, so the order in which we process the records does not matter.

Another important property is *idempotency*. That is, the result does not change even if the operation executes multiple times. Idempotency is important because a commit daemon might process transactions whose records have already been stored on S3/SimpleDB. For example, a commit daemon might crash after storing the data and provenance but before deleting the messages from the WAL queue. Upon system restart, the commit daemon will process these records once again, since they are still on the WAL queue. Re-issuing those operations will not cause any errors as S3 and SimpleDB operations are idempotent.

It is important to COPY the temporary objects to their permanent locations before deleting them to maintain idempotency. If we instead rename the temporary object to its permanent name and the client crashes before it can process provenance records from the WAL queue, it cannot re-run the operations on system restart. Finally, we should *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). However, the 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: We divide the protocol into two main phases: log and commit.

1. **Log** The log phase begins when an application issues a `close` on a file.
   (a) Read the data cache file and provenance cache file of the object.
   (b) Compute the number of records that will be necessary to log both the provenance and the data. Create a transaction, allocate it a unique transaction id, and record a *begin* record that has both the id and the number of records in the transaction on the WAL queue.
   (c) Store a copy of the data file with a temporary name on S3. Enqueue a record that has a pointer to this temporary object. The record is tagged with the transaction id and a nonce. We use the nonce in conjunction with the data to compute an MD5sum that is used to verify consistency between the data and provenance.
   (d) Group the provenance records into chunks of 8KB and store each of these groups as log records on WAL queue. Tag each of these records with the transaction id. Enqueue an additional provenance record that has the MD5sum of the data and the nonce.
   (e) Finally, we tag the commit record with the transaction id and enqueue it.
2. **Commit**
   The commit daemon executes the commit phase. The daemon periodically monitors the WAL queue for the number of messages using the *GetQueueAttributes:ApproximateNumberOfMessages* call. Once it exceeds a threshold, the daemon executes the commit phase.

(a) Receive as many messages from WAL queue as possible using the SQS *ReceiveMessage* method. Assemble messages into transactions. Due to SQS's eventual consistency, there may be times where the daemon receives the commit record of a transaction but does not receive all rest of the records for the transaction. In those cases, we need to execute ReceiveMessage until we receive all the missing pieces.

(b) For data records, execute an S3 *COPY* operation and copy the object from its temporary name to its real name. Update the MD5sum of the data and nonce as part of the COPY.

(c) Store the provenance records in SimpleDB by issuing a *PutAttributes* call. Since SimpleDB allows us to store only 100 attributes per call, we might have to issue multiple PutAttributes calls to store all the provenance.

(d) Delete all the messages related to the transaction from the WAL queue using the SQS *DeleteMessage* method. Also delete the temporary copy of the data file using the S3 *DELETE* method.

Discussion: The architecture provides read correctness, causal ordering, and allows for queries to be executed efficiently. We maintain consistency by using the MD5sum of the data and the nonce, as in the previous architecture. Queries are efficient as SimpleDB provides rapid, indexed lookup. Similar to the previous architectures, this architecture maintains eventual causal ordering.

Table 1 presents a summary of the provenance properties satisfied by each of the design alternatives.

5 Analysis

In this section, we estimate the cost of storing and querying provenance in each of the three architectures. We generated provenance for three workloads on a PASS system: a Linux compile, a Blast workload [11], and the Provenance Challenge Workload [10]. We use the combined provenance generated from all three benchmarks as one single dataset for the rest of the discussion. Using the provenance generated, we extrapolate the quantity of data and provenance that would be transferred to AWS in the three alternatives and the number of bytes that would be transferred out of AWS for three representative queries. These numbers provide a reasonable basis for comparison as Amazon charges for its services based on the amount of data transferred in and out, the amount of data stored, and the number of operations performed. SimpleDB currently does not charge for number of operations but instead charges based on the number of machine hours that were consumed. To compare the architectures using uniform metrics, we estimate the number of operations performed on SimpleDB instead of machine hours consumed in our comparisons.

Table 2 shows storage comparison estimates. We estimate the amount of storage space required to store provenance in S3 by converting each provenance record generated by PASS into the S3 metadata format. The provenance takes up 121.8MB (9.3% overhead over raw data). In general, storing provenance in S3 requires no additional operations as we store provenance as part of the data PUT operation. The only exception occurs when provenance grows larger than S3's 2KB metadata limit. To avoid this, we store any record larger than 1KB in a separate S3 object. There are 24,952 such records that result in an equal number of additional PUT operations.

We estimate the space required for the second architecture (S3+SimpleDB) by converting the provenance records into the SimpleDB data model format. We store in each SimpleDB item, all the provenance of an object version using attribute value pairs. SimpleDB limits values to 1KB, so we store any objects larger than this in a separate object in S3. This architecture requires 177.9MB for provenance storage. We compute the number of operations to store provenance as:

$$N_{SimpleDBitems} + N_{provrecs>1KB}$$

where N stands for number. The number of operations is less than the total number of provenance records, as all the provenance records of a version are stored as attributes of one item, using a single PutAttributes call.

The estimate for space required for the last scheme (S3+SQS+SimpleDB) is given by the following equation:

$$2 \times S_{SQS} + S_{SimpleDB}$$

where S stands for storage space. We store each provenance record once in SQS, then read it once from SQS for processing and storage in SimpleDB. The size of provenance stored in SQS is the same as the size of provenance stored in the first architecture (S3). The fact that we have to first store data in a temporary file and then copy the data over to its real name does not impose any data overhead as the COPY operation is not billed for data transfer.

We estimate the number of operations required to store provenance in this scheme as follows: First, we have to store each object as a temporary name and then copy it to the final name, resulting in $2 \times N(S3objects)$ operations. We store provenance records larger than 1KB as separate

Architecture	Read Correctness		Causal Ordering	Efficient Query
	Atomicity	Consistency		
S3	✔	✔	✔	✖
S3+SimpleDB	✖	✔	✔	✔
S3+SimpleDB+SQS	✔	✔	✔	✔

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

	Raw	S3	S3+SimpleDB	S3+SimpleDB+SQS
Data	1.27GB	121.8MB (9.3%)	167.8MB (13.6%)	421.4MB (32.2%)
ops	31,180	24,952 (0.8x)	168,514 (5.4x)	231,287 (7.41x)

Table 2: Storage cost comparison. The Raw column shows the amount of data that will be stored in S3 and number of ops that will be needed to store the data in S3 without any provenance. The other columns show the amount of space that will be occupied by provenance and the estimated number of ops required to store the provenance. We show in brackets, the overhead over the Raw column for storage and operations.

S3 objects. We store the remaining provenance records in groups of 8KB in SQS and later read them from SQS, resulting in $2 \times prov/8KB$ SQS operations. Finally, we store all these records in SimpleDB. Hence, the number of operations required to store provenance is given by:

$$2 \times [N_{S3objects} + \frac{provsize}{8KB}] + N_{SimpleDBitems} + N_{provrecs>1KB}$$

The results indicate that all the properties can be satisfied at a reasonable space overhead of 32% (22.9% over the first architecture). At first blush, the number of operations seems excessive. However, operations are much cheaper (in USD) than storage in the AWS pricing model.

Query	S3		SimpleDB	
	Data	ops	Data	ops
Q.1	121.8MB	56,132	51.24MB	71,825
Q.2	121.8MB	56,132	2.8KB	6
Q.3	121.8MB	56,132	13.8KB	31

Table 3: Query comparison. The data column shows the estimate of the amount of data returned by executing the queries. The ops column shows the number of operations estimated to be executed to return the results

Table 3 shows the query results. The query results are the same for the last two architectures (as they both query SimpleDB), hence we omit the results for the third architecture. We show results for the following three queries.

Q.1: Given an object and version, retrieve the provenance of that object version. The table shows the result for performing the query Q.1 on all objects. We chose to perform the query on all objects as the query results for one object are insufficient to differentiate the two methods.

Q.2: Find all the files there were outputs of *blast*.

Q.3: Find all the descendants of files derived from *blast*.

The three queries represent varying levels of complexity. In Q.1, we already know the object and therefore, the query requires only a lookup on the object (filtering out unnecessary information). In Q.2, we do not know the object, so we need to lookup all objects that satisfy a given property. In Q.3, we need to lookup all files that satisfy a property and then try to find the descendants of such a file.

Q.1 is an easy query for both S3 and SimpleDB. Since we ran the query on all objects, S3 has to effectively retrieve the metadata of all objects in the store. SimpleDB can be more selective and retrieve only the pertinent items. SimpleDB, however, needs to execute around 72K operations to retrieve the items. This is because there is no way for SimpleDB to generalize the query and needs to issue one query per item.

Q.2 is executed in two phases: First, retrieve all objects that correspond to instances of `blast`. Second, retrieve all objects that have a blast object in their ancestry. For each phase, S3 has to scan the provenance all objects as it has no search capabilities (the second phase can, of course, be executed from a cache). SimpleDB does much better as it only needs to execute one query corresponding to each phase. First, execute a QueryWithAttributes to retrieve all objects that are blast instances. Then execute a second QueryWithAttributes to retrieve all objects that have as ancestor, objects in the result of the first query. Thus, SimpleDB is much more selective in terms of the objects it needs to lookup.

For Q.3, S3 is inefficient as in Q.2. SimpleDB performs worse than in Q.2 as it does not support recursive queries or stored procedures. Hence, for ancestry queries, it has to retrieve each item with a QueryWithAttributes, then examine each item for its ancestors and

then lookup further ancestors. Despite this limitation, SimpleDB performs an order of magnitude better than S3 for Q.3.

In summary, maintaining all the properties discussed in section 3 imposes a reasonable overhead for storage and proves to be extremely efficient for queries.

6 Related Work

Several prior research projects have explored the area of capturing provenance in workflow-based and grid computing [14, 9, 8]. The prior work is quite different from that presented here in that it assumes the ability to control and alter the components in the system, while we are developing a provenance solution atop an infrastructure over which we have no control. Additionally, with the exception of recent work on provenance store failure analysis [5], previous work does not consider provenance collection in the face of failure. Since we are presenting designs that build atop highly available and reliable components, our system is fault tolerant.

7 Conclusions and Future Work

We have identified the properties that need to be satisfied for storing provenance in the cloud: read correctness, causal ordering, and efficient query. We presented various design and protocols for storing provenance and data on the Amazon cloud. We analyzed the different architectures using bandwidth and storage costs. Our analysis show that the architecture satisfying all the properties poses a reasonable storage overhead compared to a strawman architecture (22.9%) while performing orders of magnitude better on the query overhead.

We are currently engaged in implementing the alternatives to confirm the conclusions of our analysis. Furthermore, while the impact of the extra operations on the cost – in terms of USD is low, a prototype will allow us to measure the impact of the extra operations on elapsed time. AWS is currently agnostic of the metadata. The provenance stored with the data presents AWS cloud with many hints about the application storing the data. In the future, we plan to investigate how a cloud might take advantage of this provenance.

References

[1] Public data sets on Amazon Web Services (AWS). `http://aws.amazon.com/publicdatasets`.

[2] BARCLAY, T., GRAY, J., AND CHONG, W. Terraserver bricks - high availability cluster alternative. Tech. rep., 2004. MSR-TR-2004-107.

[3] 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.

[4] BRAUN, U., GARFINKEL, S., MUNISWAMY-REDDY, K.-K., HOLLAND, D. A., AND SELTZER, M. Issues in automatic provenance collection. In *Proceedings of the 2006 International Provenance and Annotation Workshop* (May 2006).

[5] 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)*.

[6] DEELMAN, W., SINGH, G., ATKINSON, M., CHERVENAK, A., HONG, N., KESSELMAN, C., PATIL, S., PEARLMAN, L., AND SU, M. Grid-based metadata services. In *Scientific and Statistical Database Management (SSDBM)* (June 2004).

[7] Ensembl. `http://www.ensembl.org/index.html`.

[8] 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).

[9] 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.

[10] MOREAU, L., ET AL. The First Provenance Challenge. *Concurrency and Computation: Practice and Experience*. Published online. DOI 10.1002/cpe.1233, April 2008.

[11] 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*.

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

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

[14] 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).

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

[16] SZALAY, A. S., KUNSZT, P. Z., THAKAR, A., GRAY, J., SLUTZ, D., AND BRUNNER, R. J. Designing and mining multi-terabyte astronomy archives: the Sloan Digital Sky Survey. *SIGMOD Rec. 29*, 2 (2000), 451–462.

[17] WLCG Worldwide LHC Computing Grid. `http://lcg.web.cern.ch/LCG/public`.

Notes

[1] AWS constantly updates their services with new features. The features discussed in the paper are a snapshot from January 2009.

[2] Thus, messages can also be used as a distributed locking mechanism.