ProvenanceLens: Service Provenance Management in the Cloud

Tao Li[1,2,3], Ling Liu[3], Xiaolong Zhang[1], Kai Xu[1], Chao Yang[4]

[1]School of Computer, Wuhan University of Science and Technology
[2]Hubei Province Key Laboratory of Intelligent Information Processing and Real -time Industrial System
[3]School of Computer Science, Georgia Institute of Technology, Atlanta, GA 30329, USA
[4]Computer Science and Information Engineering, Hubei University, Wuhan, China
litaowust@163.com, lingliu@cc.gatech.edu, xiaolong.zhang@wust.edu.cn, 476298823@qq.com, stevenyc@foxmail.com

***Abstract*—Service provenance can be defined as a profile of service execution history. Queries of service provenance data can answer questions such as when and by whom a server is invoked? which services operate on this data? What might be the root cause for the service failure? Most of the organizations today collect and manage their own service provenance in order to trace service execution failures, locate service bottlenecks, guide resource allocation, detect and prevent abnormal behaviors. As services become ubiquitous, there is an increasing demand for proving service provenance management as a service. This paper describes ProvenanceLens, a two-tier service provenance management framework. The top tier is the service provenance capturing and storage subsystem and the next tier provides analysis and inference capabilities of service provenance data, which are value-added functionality for service health diagnosis and remedy. Both tiers are built based on the service provenance data model, an essential and core component of ProvenanceLens, which categorizes all service provenance data into three broad categories: basic provenance, composite provenance and application provenance. In addition, ProvenanceLens provides a suite of basic provenance operations, such as select, trace, aggregate. The basic provenance data is collected through a light-weight service provenance capturing subsystem that monitors service execution workflows, collects service profiling data, encapsulates service invocation dependencies. The composite and application provenance data are aggregated through a selection of provenance operations. We demonstrate the effectiveness of ProvenanceLens using a real world educational service currently in operation for a dozen universities in China.**

Keywords. service provenance; service dependency; execution history, service profiling.

I. INTRODUCTION

Everything in the cloud is delivered as a service, from infrastructure and platform to software and applications. Service provenance is a description of service execution history. Capturing and understanding the dynamic workflows of service execution is critical to improve service quality, guide resource allocation, analyze causes of service failures, detect security vulnerabilites. Both Internet company (such as google, twitter) and software company pay more attention to service behavior monitoring [1,2], because service provenance can help answer questions such as who invoke this server, which set of data are accessed by the server, which service is bottleneck of the system. By examining the service runtime state statistics together with service invocation graph, we can find the root cause of a service failure. By analyzing the service invocation patterns, we can evaluate the side effects of a service failure. By collecting input and output of a service execution and its execution path, we can evaluate the correctness of a complex service execution. By surveying elapsed time of services and their execution paths, we can gain better understanding of where the system bottlenecks might be.

As cloud systems and applications become more distributed and big data centric, not only services are deployed in distributed compute nodes but the number of services involved in accomplishing a job also continues to increase. This raises more challenges for service provenance capturing and service provenance tracing with respect to service invocation, dynamic and complex execution dependencies among a large number of services. In addition, service provenance management also needs to deal with large service provenance data and determine which types of service provenance data should be stored persistently, which provenance data should be derived on demand based on stored provenance data, and how to efficiently derive new provenance data over stored service provenance. Thus, service provenance management in general consists of provenance capture, storage and analysis. By providing service provenance management as a service, we need to design the provenance management functionalities in such a way that they can be easily used by an application to monitor the runtime behavior of its services. Concretely, the following system requirements should be the core design objectives:

Automated and configurable provenance capturing. The service provenance management by design should take into account of heterogeneous services and application systems and identify the essential provenance data that are common to many applications and their cloud services. More importantly, the provenance collection process should be automated and easily configurable for provenance capturing and provenance analysis, while maintaining application transparency.

Light-weight. One of the key design goals for service provenance management is light-weighted. First, service provenance capturing should be exercised with little

COLLABORATECOM 2014, October 22-25, Miami, United States

DOI 10.4108/icst.collaboratecom.2014.257339

overhead that has negligible impact on the performance of routine executions of applications. Thus, optimization techniques should be employed at the right time for the right applications. For example, some application services may execute thousands of times in a short period of time. The provenance capturing modules should turn on selective sampling with adjustable intervals to allow provenance capturing to be zoomed out with larger monitoring intervals or selectively zoomed in with shorter intervals.

Scalable data management. As the number of services involved in cloud applications continues to increase, the dataset size of service provenance to be collected and managed continue to grow. Thus, a scalable service provenance management should support two core functionalities: (i) compact storage with the support of seamless scale out to a distributed provenance storage and (ii) efficient provenance analysis by providing a set of basic provenance operations, enabling querying, tracing and reasoning over large provenance data across multiple services within an application and across applications within a cloud hosting service.

In this paper, we present ProvenanceLens, a two-tier service provenance management framework, which is designed to meet the above requirements. The top tier is the service provenance capturing and storage and the next tier provides analysis and inference capabilities of service provenance data. Both tiers are built based on the service provenance data model, an integral part of ProvenanceLens. A unique feature of this provenance data model is to categorize service provenance into three broad categories: basic provenance, composite provenance and application provenance. We also introduce the provenance data structure and a suite of basic operations, such as select, project, aggregate, slice, dice, roll-up, drill-down, and trace. The basic provenance data is collected through a light-weighted provenance capturing system that monitors service execution workflows, collects service profiling data, encapsulates service invocation dependencies. A real-world cloud service in educational domain managing teaching services for a dozen universities is used as the use-case study to demonstrate the effectiveness of ProvenanceLens.

The rest of the paper is organized as follows. Section 2 reviews the architecture of ProvenanceLens and the use-case scenario. Section 3 describes the ProvenanceLens data model. Section 4 presents the ProvenanceLens capturing system. Tthe experimental evaluation of ProvenanceLens on the use-case study is given in Sections 5. We discuss related work in Section 6 and conclude the paper in Section 7.

II. OVERVIEW

A. Architecture

The ProvenanceLens system by design consists of two subsystems: the provenance capturing subsystem and the provenance management and analysis subsystem. The provenance capturing subsystem performs service runtime profiling and derives execution dependency through execution workflow analysis over the basic provenance log created by different service profiling agents. Every profiling task is controlled by the provenance collector management. System administrator can start or terminate a service profiling task or set provenance collection strategy, such as where and what types of service provenance data should be collected, the interval of sampling, how long the provenance collection process should run, and the type of composite provenance data to be derived over the captured basic provenance data.

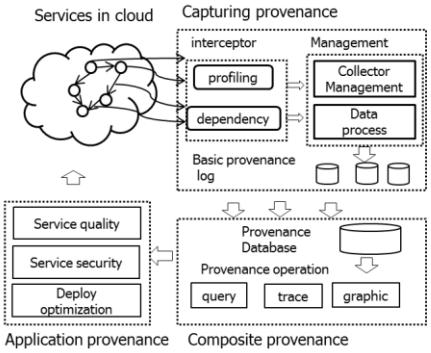


Fig. 1 ProvenanceLens System Architecture

The provenance query and analysis subsystem allows the system administrator to generate a selection of composite provenance data and store them in persistent provenance store for on-demand access. In the first prototype development, three types of provenance operations are provided to generate composite provenances: provenance query operation, provenance trace operation and graph based service correlation operation. In addition, the ProvenanceLens system will also provide the application level provenance data for application specific provenance analysis. An extensible application provenance library is provided to allow application-specific provenance analysis or functionality specific provenance analysis to be performed conveniently for service quality evaluation, service security compliance evaluation, and cloud deployment optimization.

B. Case Study

Figure 1 shows a teaching management system deployed in an IaaS cloud for 12 universities in P.R. China. Given that universities vary from one another in terms of both the number of students, the number of teaching faculties and the number of courses being offered within each semester or quarter, the service response time for requests coming from different universities can vary dramatically. Even for the same service, the peak time performance for the same university can differ significantly from normal time. These motivate the use of service provenance management on this operational cloud system.

By performing service profiling using the techniques developed in ProvenanceLens, we conclude with a number of interesting observations: First, the concrete quality of service (QoS) requirements can vary significantly from university to university. Second, with different business model for course and teaching management at different universities, the service runtime behavior for each university is different. Some services need more CPU resource, such as ArrangeCourse, TeachingTask. Other

services need more I/O resource, such as StudentProject management. Also some services may have clearly understood business peak-time performance, such as RegistCourse, LessonPlanManagement. The peak time behavior of these services depend on many factors: the number of students, the number of business processes, the type of user behavior. Thus it is hard to find the root cause of some failures of a service invoked by different universities.

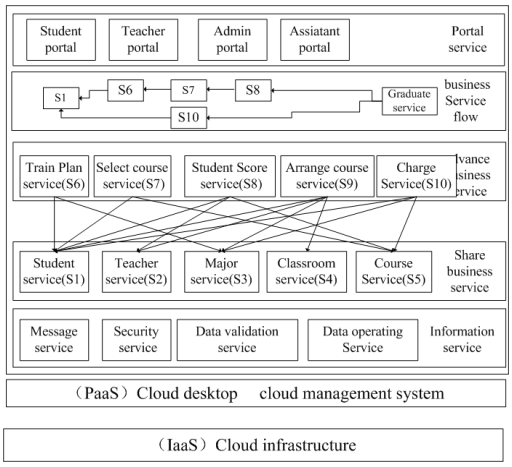
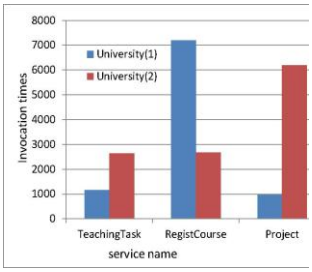


Fig. 2 A University-Teaching Management System deployed in an IaaS cloud

Figure 3 and Figure 4 show two example provenance data captured by the university service provenance analysis over the basic service provenance.



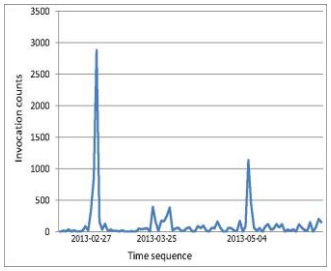


Figure3 service invocation times Figure4 service invocation time-sequence

In Figure2, we choose three typical services: TeachingTask, RegistCourse, Project service, which represent three different service types: computation intensive, concurrent execution, file operations. University(1) has high invocation frequency for RegistCourse service, it suggests to the system administrator that additional servers may be beneficial if the peak time performance should meet a specified QoS requirement such that all registration requests by students for a course are guaranteed. University(2) has a relatively hot TeachingTask service, which indicates the potential demand for more CPU resource. In Figure3, we show that LessonPlanManagement services in university(2) using time sequence. It indicates that the peak time workload for the file storage was experienced on 2013-02-27. These examples clearly show the importance of service profiling and service provenance in understanding the service runtime performance, detecting system bottlenecks, and providing cost-effective capacity planning. Figure 5 shows the invocation dependence for RegistCourse service. When the RegistCourse service encounters a runtime error and aborts suddenly, provenance data can help system administrator to detect possible root causes. For example, the RegisterCourse service has multiple runtime execution paths. In order to understand which execution path was the cause of this failure, we need to examine both the execution history to see if there are any other reported failures from relevant services. This leads us to find that the CheckConfilct service S_7 reported an error in the same duration of S_1. Then we need to perform composite service provenance analysis to trace the set of execution paths from S_1 to S_7 over the captured provenance data. By narrowing down the focus to the execution path of $S_1 \rightarrow S_2 \rightarrow S_7$ and $S_1 \rightarrow S_3 \rightarrow S_7$ and the fact that only UserRegister S_3 was active in the short interval prior to the occurrence of the failure, this leads us to infer with high probability that the CheckConfilct service would be the root cause of the failure of the registerCourse service. Clearly, by maintaining traces (different execution paths) as a composite provenance, this root analysis can be performed faster in the presence of complex application services, composed by many correlated services.

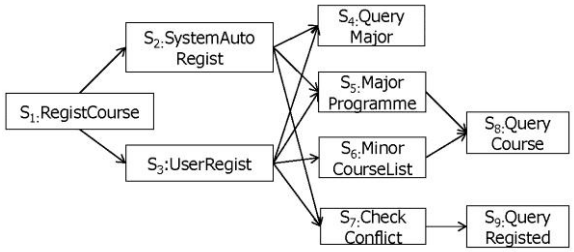


Fig. 5 execution dependency in registerCourse service

In ProvenanceLen, we address these requirements through the service provenance model which defines both the data structure and the provenance operations for both basic provenance captured directly by the provenance collector agents and the composite provenance derived through the use of basic provenance operations.

III. SERVICE PROVENANCE MODEL

In this section, we describe the service provenance data model developed in the ProvenanceLens system. The first feature of our provenance model is the separation of three types of service provenance data: the basic provenance, the composite provenance and the application provenance. The second feature of our provenance model is the support for three types of provenance operations: the basic provenance query operations, the provenance aggregation operation, and the provenance correlation graph operations. We use the case study in the previous section to illustrate the main concepts, data structure and operations.

A. *Basic Provenance*

Given a set of cloud services denoted as $S=\{S_1, S_{2,..} S_n\}$, we use invoke(S_i,S_j) or $S_i \rightarrow S_j$ to describe the service invocation relationship that S_i invokes S_j. Each invocation relationship is also associated with the provenance captured during the invocation and execution of services S_i and S_j, such as time, location, elapsed time, input, output. In general, for each service execution, we need to collect the seven types of provenance, which describe "what", "when", "where", "who", "which", "how" and "why" [8]. "What" means the invocation event. "when" describe the time of invocation, we record the request time and response time which can help us compute the elapsed time. In distributed execution environment, due to time synchronization in distributed nodes, we also compute the elapsed time based on local node clock. "Where" describe the service location ,such as IP address. "Who" describe the invocation initiator. "Which" describe invocation protocol and the service framework used, such as RPC, WSDL, SOAP. "Why" describes the reason of the invocation fault. "How" describes the path of invocation. "When", "where" and "who" can be captured directly. "How" can be computed by a series of calling service and invoked service. "Why" can be inferred by service status in the path.

TABLE 1. SERVIVCE PROVENANCE CONTEXT

Concept	**Provenance ontology**	**service provenance** (example)
what	Event	Invocation
when	Time	Invocation timestamps
Where	Space	Service IP address
who	Agent or other things	service invoking
Which		service invocation protocol
how	Action, input/output	Invocation path
why	Reason	Invocation fail reason

Table 2 shows the data structure of the basic provenance captured by the service profiling agents in the provenance capturing phase. We can view each basic provenance as a nine dimensional object.

TABLE 2. BASIC PROVENANCE

item	**Description**
Token	32 bit string which identify a relatime job
InvokingService	Service customer
ServiceInvoked	Service provider
Location	IP address
Elapsed time	Time cost of one invocation
Timestamp	Receive invocation request time
Input	Input parameter
Output	Output parameter
Status	1-sucessful 0-fail, time out

Table 3 shows an example of the basic service provenance records captured for the RegistCourse service described in Figure 4. T1, T2, T3, T4 are four different tokens, representing four different jobs. (T1, S1, S2, 35ms, true) describes that service S1 invoke service S2 with the elapsed time of 35ms. This ElapsedTime includes the execution time of S2.

TABLE 3 REGISTCOURSE EXAMPLE

token	Invoking	Invoked	ElapsedTime	status
Token1	S_1	S_2	35ms	True
Token1	S_2	S_5	20ms	True
Token2	S_1	S_3	50ms	True
Token2	S_3	S_5	30ms	True
Token3	S_5	S_8	12ms	True
Token4	S_3	S_6	---	false

B. *Composite Provenance*

For each service execution captured by the provenance collector agent, we separate those that capture the state of a service, such as success or failure status, active or inactive status, from those that capture invocation relationship and the execution dependency of a pair of services, such as $S_1 \rightarrow S_2$ and S_1 preceeds S_2. We call the former the *data-oriented provenance* and the later the *process-oriented provenance*. By utilizing data-oriented provenance, such as (service, status), we can figure out the exception rate of a service execution, denoted by ServiceExceptionRate(S_1, status). This can be computed using the number of failed execution divided by the total number of executions in a given time interval. We call such derived provenance the data oriented composite provenance. Similarly, by utilizing the process-oriented provenance, we can infer process-oriented composite provenance. An example is how frequently the invocation of a service fails, denoted by InvokeExceptionRate(S_1, S_2, status). Another example is which services are dependent on a given service, which can be derived by evaluating whether they counting on the number of services that are reachable from the given service through the invocation graph (Figure 4).

Composite provenance can be obtained by employing simple aggregate functions, such as count, avg, max, and min, over the basic provenance records. They can also obtained by using more complex provenance operations as those to be discussed in the subsequent section. For example, we can aggregate the time based provenance by using data cube summarization techniques, such as using coarse time granularity, such as hours, days, months, to summarize the service invocation frequency provenance over a period of day or week or month. This will allow us to answer questions such as which weekday the highest invocation frequency is observed for a given pair of services.

Given a finite set of basic provenance records captured by the provenance collector agents, we can derive a large collection of composite provenance. Consider the storage cost and the search space explosion, a wise decision is to carefully select a small set of composite provenance for persistent storage and compute the remaining composite provenance on-demand. The choice is often application dependent. For example, a system administrator may need certain composite provenance data on a subset of services for improving service availability.

C. *Application Provenance*

Application provenance refers to some high-level provenance data analysis that are application dependent. For example, one can use a graph structure to capture the invocation relationship among a collection of services. By modeling services as vertices and the invocation ElapsedTime as the edge weight, we can employ subgraph matching algorithms to find the common invocation patterns. We can also employ graph clustering algorithms to group services into K clusters such that services within the same cluster are densely connected.

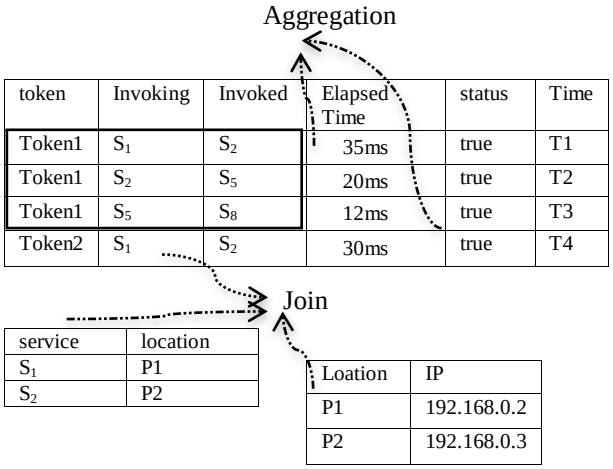
Given that different applications may require different application provenance based on both composite provenance and application specific service management requirements, in ProvenanceLens, we provide a library of graph analysis algorithms to facilitate the creation and derivation of application specific provenance from the basic provenance data and the composite provenance data.

D. *Service Provenance Storage*

We consider support three types of storage schema : relationship schema, document schema and graphic schema. In the first prototype implementation of ProvenanceLens, we choose relational database MySQL, the document database MongoDB, and the graph database Neo4j[18] to store both the basic provenance captured by our provenance capturing tool and the composite provenance derived from the basic and composite provenance data.

(1) Storing *Relationship based provenance data*

We store the invocation relationship information in relational table. Figure 6 shows an example of such relational table with six columns: token, invoking, invoked, elapsed time, status and time. The relationship between a service and the location where the service is executed and the relationship between the location and the IP address are stored separately in two different tables.



token	Invoking	Invoked	Elapsed Time	status	Time
Token1	S_1	S_2	35ms	true	T1
Token1	S_2	S_5	20ms	true	T2
Token1	S_5	S_8	12ms	true	T3
Token2	S_1	S_2	30ms	true	T4

service	location
S_1	P1
S_2	P2

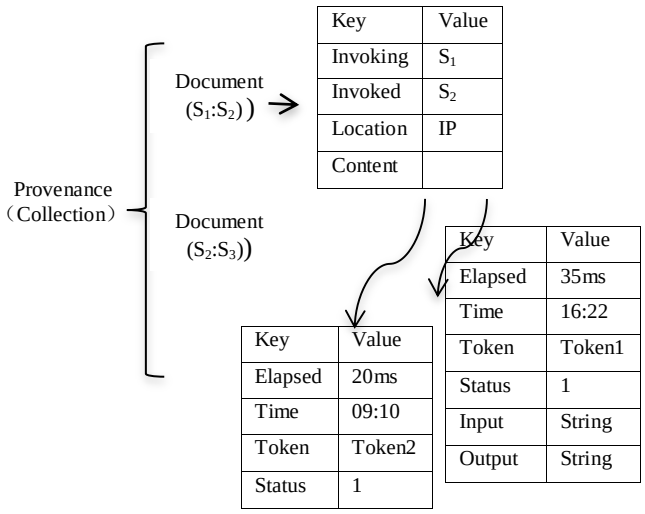
Loation	IP
P1	192.168.0.2
P2	192.168.0.3

Figure 6 Relational data storage for relationship based provenance

Aggregation operations. In order to support summarization of basic provenance records captured by our service profiling tool, we provide a suite of aggregation functions such as sum, count, average, sort, group. By employing these aggregation operations, we can compute how many times did S_1 invoke S_2 in the past 3 days, what is the failure rate of this invocation in the past 4 weeks. When the service invocation relationship table is large in size, the computation complexity of such aggregation operations can be significant. Performance optimization techniques can be beneficial for improving the real-time analytics.

Join operation: Join operation is essential for computing some of the composite provenance and application provenance. For example, consider the basic provenance stored in three different tables in Figure 6. If we want to get the invocation frequency grouped by location, we need to perform join on the invocation table and the service-location table. In addition, we need to use join operation to carry out the invocation path tracing through self-join on the invocation table by examining <invoking, invoked> relationship row by row. Such join operations can be expensive for long tables and indexing techniques can help speedup both relational selection and relational join operations.

(2) storing provenance as *documents*



Key	Value
Invoking	S_1
Invoked	S_2
Location	IP
Content	

Key	Value
Elapsed	35ms
Time	16:22
Token	Token1
Status	1
Input	String
Output	String

Key	Value
Elapsed	20ms
Time	09:10
Token	Token2
Status	1

Figure 7 Storing provenance as documents

Document based storage is a typical NO-SQL data schema. Figure 7 shows the storage schema for storing provenance data as documents. We choose Mongodb to store the basic provenance in the form of documents. Data store in one collection. Each service dependency is a document which include all the invocation related data between the two services. In each document, there are three key-value pairs, which describe invoking, invoked and content. The content is a sub-document and it describes the invocation detail, which includes elapsed time, invocation time, token, status, etc.

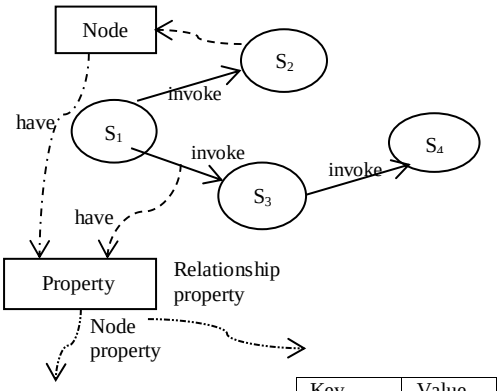
Comparing with the relational model for storing the basic provenance, the document model has a number of advantages. First, by using a key-value store to host the basic provenance, one can extend the structure of the basic provenance data easily for any given key, such as invoking and invoked by simply adding the new content as the new value by following the key-value structure.

Document based storage model can also improve query and insertion performance. Given a service S_1, in order to query how many services are invoked by S_1, we just look up S_1 related data, which are stored in one document with key S_1, instead of querying the entire table.

(3) Storing basic provenance in Graph stores

Although relational mode and document model based storage systems can reduce query space and improve storage efficiency, they cannot implement trace operation easily because the trace operation needs to perform iteration operation according to the service invocation dependency. By storing the service invocation provenance data in a graph store, say neo4j[18], we can easily compute the path of service execution.

Figure 8 shows an example of storing some of the basic provenance records in the graph database, where the vertices of the graph represent services and the edges represent the relationship of the service invocation. The property refers to both node and relationship property, which are represented and stored as key-value pairs. The node property describes the name, location, url of the service and the relationship property describes the elapsed time, the invocation time, token, status, input and output of services.



Key	Value
Name	SelectCourese
Location	IP
URL	Http://service

Key	Value
Elapsed	30ms
Time	16:22
Token	Token1
Status	1
Input	String
Output	String

Figure 8 An example of storing provenance in a graph database

By storing the basic provenance records in a graph database, we can conveniently provide complex path operations, such as shortest path, connected components, and so forth.

IV. SERVICE PROVENANCE CAPTURING

In this section, we introduce the framework for provenance capturing in ProvenanceLens. We first describe and how to capture and store provenance data. First, we illustrate capture point—where does we capture provenance, we compare white-box and black-box based provenance capturing methods and analyze related feature. Second, Then we discuss how to capture service dependency and how to reduce capturing cost by sampling strategy.

A. Capturing Methods

The first provenance capturing method is referred to as the white box method. It is suitable to capture disposed provenance, namely the provenance data about the services and their runtime execution at application level or user level and thus we as the provenance consumers know about the detail and can make more semantic annotatations. Typical examples are the elapse time of a service invocation. Generally, this approach carries rich semantic information about services and their execution environment, the service developers or service providers can use API to capture such disposed provenance. In comparison, the black-box method is used to capture observed provenance, such as those captured in Dapper[1] and PASS[3]. These types of provenance data are collected by the system automatically and transparently at low level of the system and the provenance users may not know more information about these provenance data to add rich semantic annotations. In comparison to the white-box methods, the provenance information collected using the black box method is relatively poor in quality and often depends on the specific system, framework or protocol.

In ProvenanceLens, a hybrid approach of white-box and black-box method is employed. The interceptor is deployed in the framework manually when it first invoked and run. The interceptor will first print basic provenance log, then all the provenance data can be collected by the system. Compare with the black box method and the white box method, this hybrid approach implements automated provenance collection, because the collector can contain different framework interceptors or collect provenance logs which are produced by other external tools, framework or invocation protocols. Because all the invocations must be executed through either the framework or the protocol, while keeping the balance between the transparency and flexiblility. In fact, the transparency depends on where an interceptor is deployed. If an interceptor is designed based on low level (library), it will have lower time cost and lower level of transparency. If an interceptor is designed based on high level application, it is not transparent, but it has rich high level semantics.

We collect service provenance data by service interceptors. Usually there are some interceptors according to different platforms. For example, JAVA EE has provided an interceptor, which can be used to logging, auditing, and profiling. We have developed an example interceptor based on CXF[10], an open web service framework. The interceptor could be deployed in the service customer and the service provider when the service is being published.

B. Capturing Service Dependency

The core problem of capturing service dependency is related to transmitting tokens which identify a job between a pair of services (consumer and provider). We introduce how to capture service dependency between a service provider and a service customer. (1) Token is produced by the service customer (invoker) in the entry of a transaction and stores in the local context variable of the thread. (2) When invocation happens, the interceptor will insert basic provenance into the head of the respective service invocation protocol. (3) When the service provider receives the invocation request, the interceptor would exact the token information from the head of the protocol, then put the token into local context variable of the thread. (4) Service customer receives the invocation respponse, and have the interceptor would print the basic provenance.

C. Sampling

Some hot service, such as GetUseName, RegistCourse may be invoked several thousands of times in a short period of time. Provenance data capturing agent will spend large resource costs, from CPU, memory to disk I/O and network I/O cost and storage space cost. Thus, similar to [1], ProvenanceLens system will use sampling as a valid method to address this problem. There are two commonly used sampling methods (i) Uniform sampling rate based method, in which all services will adopt the same sampling rate. But lower frequency service may miss important events. If we increasing sampling rate, it will affect the performance. (ii) Adaptive sampling rate based method can make adaptive sampling decisions such that low frequency service will trigger the increase of sampling rate automatically, while high frequency service will need to reduce the sampling rate to some extent. In addition, the high frequency service using low sampling rate for provenance capturing should set the provenance capturing interval to ensure no loss or minimum loss of important provenance information.

Choosing appropriate sampling rate parameter is important to enable the provenance-capturing agent to adaptively tune the setting of the parameter. For example, if the time for running the interceptor is 1 ms for every service invocation, and the throughput for serving a request will not exceed 1000 per second for common services in our use-case, because the services that handle higher request rates will be implemented as a distributed service running on a cluster of compute nodes. In ProvenanceLens, the sampling rate is set by default to 0.1% to 1% for popular services.

D. Scalability

When a large number of services are deployed and invoked concurrently, ProvenanceLens should provide seamless scalability such that it can handle provenance capturing, provenance storage, provenance reasoning in real time and on demand. In ProvenanceLens, we provide three levels of support for improving the system scalability.

(1) Logical Independence between the inteceptor, the provenance collection, storage and analysis. This loose coupling basd abstraction enables each component to be modified or extended with no impact or minimum impact on the functionality of other components.

(2) Distributed Architecture for Provenance Capturing and Collection, which enables the new servers to be added to the existing infrastructure of the system seamlessly, enabling scalability of the ProvenanceLens with the increasing number of services and increasing amount of provenance data to be captured.

(3) Strategic Refinement Management, which allows the provenance capturing strategies, such as priority, scale, frequency, to be refined and revised. For example, we can set high priority for a given subset of services due to the demand for emergency based performance tuning.

In Figure 9, we show a sketch of the capturing tools and the framework of the provenance capturing agent, which is responsible for monitoring the collector and transfer configuration information to the relevant parties. We represent and store the provenance capturing strategies as collecting rules. Data flow refers to the process of provenance capturing, which includes the data check and append-only database.

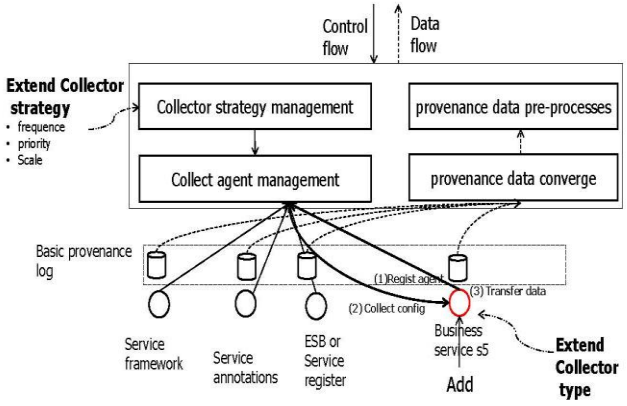


Figure 9 The Framework of Provenance Collector

In Figure 9, provenance log is designed to keep loose coupling between the provenance capturing system and the provenance analysis system. The provenance log can be produced by the inteceptors through annotation and ESB log. If provenance log is different from the basic provenance records we obtained, we provide a mapping tool to establish the one to one mapping between them. In fact, such mapping tools can be built by extending some existing schema-mapping tools that support heterogeneous data mapping, such as MapForce[17].

V. EVALUATION

In order to verify the performance of our service provenance management system ProvenanceLens, we design some experiments to evaluate the performance of service provenance capturing, usage and operation consumption based on the real world use-case outlined in Section II. The provenance server used in this measurement study has 8G RAM, CPU Xeon 2.5GHz, and the client node has CPU AMD 2.8G with 2G RAM.

A. *Capture Tools: Time Complexity*

We measure the runtime of the inteceptor by varying the output size and invocation frequency. The service used in this measurement study is the common query service, called UserInformation service, which responds to each query by returning the UserName, Password, department information. The intercepted output data from 100 to 1000 rows. Because the cost of running interceptor is low, so every intercept executes 1000 times. In Table 4, we record time (unit: ms) of the service without using interceptor (NoProv), then measure the time of the same service with interceptor (Proxv). The results in Table 4 show that the cost of running interceptor will increase with the increase of the output size. Especially, when output size exceeds 1000 rows, the runtime cost grows drastically.

Table 4 intercept time cost with output (time:ms)

OutSize	No Interception	With Interception	Intercept
100	11.09	11.39	0.30
200	16.46	16.93	0.47
400	27.55	27.89	0.34
600	40.17	41.05	0.88
800	51.35	52.12	0.77
1000	60.09	65.05	4.96

Next, we evaluate the impact of the invocation frequency on the interceptor performance. In this set of experiments, the service will be invoked at varying frequency, such as 100, 200, 1000. Table 5 shows that as the invocation frequency increases, the interceptor runtime cost is relatively stable. Thus, the invocation frequency does not have significant impact on the interceptor performance. The average time cost of interceptor is less than 1 ms. Also the small difference between no interception and with interception is the cost of initialization of interceptor.

Table 5 intercept time cost with frequency (time:ms)

OutSize	No Interception	With Interception	Intercept
100	1588	1681	0.93
200	3247	3408	0.81
400	6497	6644	0.37
600	9787	10183	0.66
800	13042	13326	0.36
1000	16362	16656	0.29

Figure10 measures the interception runtime by varying the number of output rows and the invocation frequency. It shows that when the number of output rows exceeds 1000, the inteceptor runtime is drastically increased. On the other hand, as the increase of invocation counts, the average time of every invocation is relatively stable.

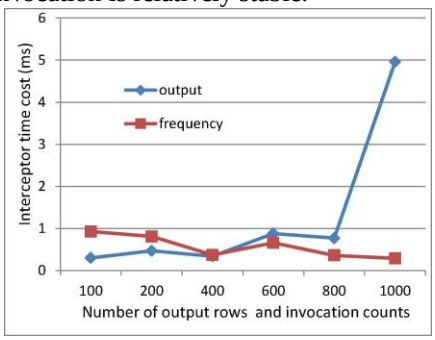


Figure10 inteceptor time cost of frequency and output

Figure 11 shows the interception runtime for 40 different invocations of the inceptor and we can see that the interceptor runtime cost is very low, which shows that the provenance capturing agent in ProvenanceLens is light weight and non-intrusive.

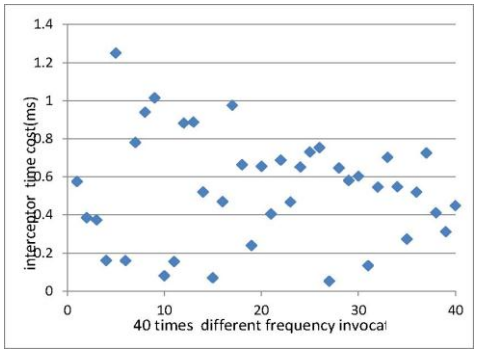


Figure 11 inteceptor runtime cost of 40 different service invocations

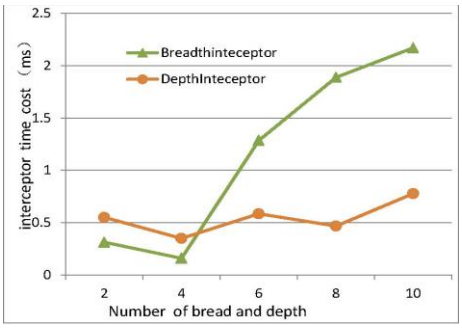


Figure12 Inteceptor runtime cost with varying breadth and depth of invocation

The next set of experiments evaluates how the interception runtime is impacted by both depth and breath t of service dependency. Depth here refers to the sequence of invocations. Breadth refers to the number of services being invoked at same time by a service. In this set of experiments, the service being monitored will invoke 2, 4, 6, 8, 10 other services. Figure 12 shows that both breadth and depth of the service invocation. It is observed that the depth of the service invocation do not have much impact on the runtime

cost of the interception when the number of invoked services varies from 2 to 10. However, the breadth of the service invocation may impact the runtime performance of the interceptor. When the invocation breadth increases from 2 to 10 services, the runtime cost grows as well.

B. Resource Utlization

We measure the percentage of CPU and memory resource utilization when varying the number of tasks triggered for provenance capturing. Figure 13 shows the measurement results, where 32tps means that there are 32 collection tasks per second. For the popular services, we collect the provenance data with high frequency, thus it incurs high CPU cost. We observe that the CPU utilizaiton is 35 percent when the collector frequency is at 388 tasks per second. On the other hand, the memory utilization is low and relatively stable, showing that the amount of provenance data captured is relatively small.

.

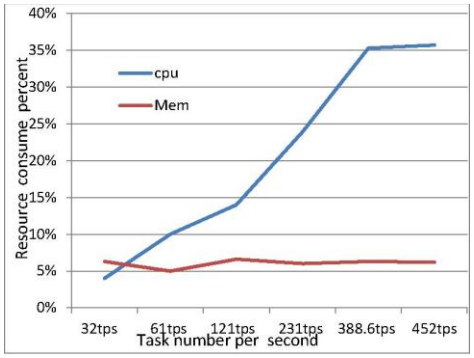


Figure13 Resource Utilization

C. Runtime of Three Different Provenance Stores

Figure14 measures the runtime cost of three different provenance stores by inserting a varying amount of provenance data in MySQL, MongoDB, Neo4j respectively with 100, 200, 500, 1000, 2000 thousand rows. MySQL spent more time doing insertion than MongDB and Neo4j. With the increasing rows of insertion data, the operation runtime increases quickly for MySQL, while the run time of Neo4j increases slightly and the runtime of Mongodb is very stable. By inserting 2000 thousands of rows of provenance data, MySQL runtime is 192.81 seconds, Mongodb runtime is 6.234s and Neo4j is 38.781 seconds.

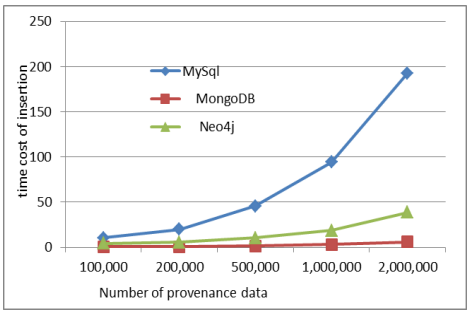


Figure14 Runtime of insertion in three stores

Figure15 compares the database size of MySQL, Mongodb and Neo4j with 10,100, 200, 500, 1000, 2000 thousands of rows inserted into the provenance store. As the number of rows increases, the database size in Neo4j grows slowly than the other two stores. However, when the database is relatively small (10,000 rows or less), Neo4j spends more storage space than the other two stores due to the graph structure maintained in Neo4j. In comparison, as the number of provenance records increases, MongoDB spends largest space than the other two stores

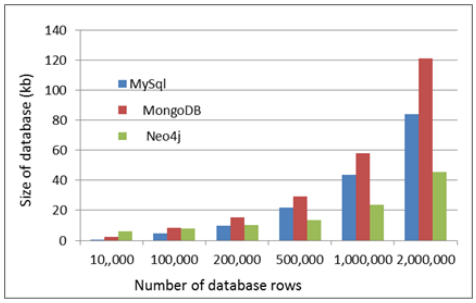


Figure15 Storeage cost with three stores

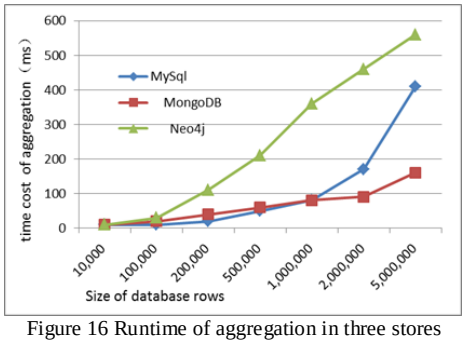


Figure 16 Runtime of aggregation in three stores

Figure16 measures the runtime of a typical aggregation by varying the size of the provenance records captured and stored in the three stores. We query how many times did service S_1 invoke S_2. With different provenance database sizes, the result shows that MongoDB has the most stable performance as the provenance database grows in size, whereas MySQL spends more time than MongoDB when the data size reaches 1,000,000 and Neo4j takes much longer to perform this aggregation query compared to MySQL and MongoDB when the size of the provenance database grows.

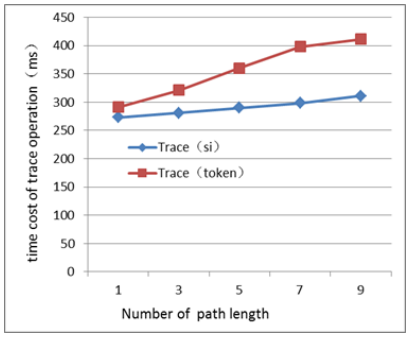


Figure17 runtime cost of trace operation

Figure17 measures the runtime cost of the trace operation by varying the length of path from 1, 3, 5, 7 to 9 services in one invocation path. Trace(token)(return the path according to token value) consumes more runtime than trace(s_i).

VI. Related Work

Provenance has been used in many domains [2], such as biology, Chemical Sciences, Earth Sciences. From the cloud perspective, there are various categories of services, ranging from business services, middleware services, database services, network services to application services. Existing work on capturing and managing provenance can be broadly divided accordingly. For example, Dapper[1] is dedicated for provenance management in application services, though its service trace tools are limited to homogeneous platform and do not support heterogeneous systems. PASS (provenance-aware storage systems)[3] designed for capturing and managing provenance in file systems, PASS implements the file provenance atuo-collection,and storage management by modifying the Linux kernel in order to intercept the system call of file operations. DTaP[4] is managing provenance in network. Trio[5] is collecting provenance in database, and provenance in databases [14,15,16] focus on view update, data tracing. Moreau [7] proposes web data provenance. Karma2 [6] is dedicated for provenance in workflow. Some workflow systems [11,12,13] collect the provenance data from the logs of workflow engines. To the best of our knowledge, this work is the first one considering provenance management as a service and provides a systematic framework for service provenance management in the Cloud. .

VII. Conclusion

We have presented ProvenanceLens, a two-tier service provenance management framework. The top tier is the service provenance capturing and storage subsystem and the next tier provides analysis and inference capabilities of service provenance data. Both tiers are built based on the service provenance data model, which categorizes all service provenance data into three broad categories: basic provenance, composite provenance and application provenance. ProvenanceLens also provides a suite of basic provenance operations, such as select, trace, aggregate. The basic provenance data is collected through a light-weight service provenance capturing subsystem. The composite and application provenance data are aggregated through a selection of provenance operations. We demonstrate the effectiveness of ProvenanceLens using a real world educational service currently in operation for a dozen universities in China.

Acknowledgement. The first author performs this work while visiting DiSL in Georgia Institute of Technology. Tao Li, X.Zhang and Kai Xu are partially supported by the National Natural Science Foundation of China (Grant Nos. 61273225), Humanities and Social Sciences Foundation of Education Ministry of Hubei province (Grant Nos. 2012D111). Ling Liu is partially supported by NSF under Grants IIS-0905493, CNS-1115375, IIP-1230740.

References

[1]. Benjamin H. Sigelman, Luiz Andr′ e Barroso, Mike Burrows, Pat Stephenson,Manoj Plakal, Donald Beaver, Saul Jaspan and Chandan Shanbhag, "Dapper, a Large-Scale Distributed Systems Tracing Infrastructure", Google Technical Report dapper-2010-1, April, 2010.

[2]. Yogesh L. Simmhan,Beth Plale and Dennis Gannon, "A Survey of Data Provenance in e-Science" ,SIGMOD,VOL 34.NO.3.Sept.2005

[3]. Kiran-Kumar Muniswamy-Reddy,David A. Holland, Uri Braun,and Margo Seltzer,"Provenance-Aware Storage Systems", 2006 USENIX Annual Technical Conference (USENIX'06) , June 2006

[4]. Wenchao Zhou, Micah Sherr, Tao Tao, Xiaozhou Li, Boon Thau Loo, and Yun Mao, "Efficient Querying and Maintenance of Network Provenance at Internet-Scale", SIGMOD ,Jun 2010

[5]. P. Agrawal, O. Benjelloun, A. Das Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. "Trio: A System for Data, Uncertainty, and Lineage. Proc". 32nd Intl. Conference on Very Large Data Bases , pages 1151-1154, Seoul, Korea, September 2006.

[6]. Yogesh L. Simmhan, Beth Plale, Dennis Gannon, "Karma2: Provenance Management for Data Driven Workflows", International Journal of Web Services Research , Vol.X, No.X, 200X

[7]. Luc Moreau, "The Foundations for Provenance on the Web", Foundations and Trends in Web Science Vol. 2, Nos. 2-3 (2010) 99-241

[8]. Sudha Ram and Jun Liu，"A New Perspective on Semantics of Data Provenance", SWPM-2009, 2009.10

[9]. Linton C.Freeman,"Centrality in social networks conceptual clarification", Social Networks}, Volume 1, Issue 3, Pages 215-239,1978

[10]. cxf,http://cxf.apache.org/

[11]. Susan B. Davidson, Juliana Freire, "Provenance and scientific workflows: challenges and opportunities", SIGMOD, 2008

[12]. Rajendra Bose and James Frew. " Lineage retrieval for scientific data processing: A survey", ACM Computing Surveys, 28 March 2005.

[13]. J. Zhao, C. A. Goble, R. Stevens, and S. Bechhofer, "Semantically Linking and Browsing Provenance Logs for Escience,"in ICSNW, 2004.

[14]. Y. Cui and J. Widom, "Lineage tracing for general data warehouse transformations," VLDB Journal , vol. 12, 2003.

[15]. Peter Buneman,Adriane P.Chapman,James Cheney,"Provenance Management in Curated Databases", SIGMOD, 2006

[16]. Laura Chiticarius James Cheney and Wang-Chiew Tan. "Provenance in databases: Why, how, and where", Foundations and Trends in Databases (4):379-474, 2009.

[17]. Altova MapForce 2009, MapForce graphical data mapping, Conversion & Intergration tool.Retrieved May 8, 2009, from http:// www. altova.com /documents/mapForcedatasheet.pdf

[18]. www.neo4j.org