Chapter 4
Sketching Distributed Data Provenance

Tanu Malik, Ashish Gehani, Dawood Tariq, and Fareed Zaffar

Abstract Users can determine the precise origins of their data by collecting detailed provenance records. However, auditing at a finer grain produces large amounts of metadata. To efficiently manage the collected provenance, several provenance management systems, including `SPADE`, record provenance on the hosts where it is generated. Distributed provenance raises the issue of efficient reconstruction during the query phase. Recursively querying provenance metadata or computing its transitive closure is known to have limited scalability and cannot be used for large provenance graphs. We present *matrix filters*, which are novel data structures for representing graph information, and demonstrate their utility for improving query efficiency with experiments on provenance metadata gathered while executing distributed workflow applications.

4.1 Introduction

The provenance of data is a description of how the data came into being or was derived. Provenance metadata is becoming increasingly useful in addressing a wide variety of issues, such as performance optimization, generating repeatable and reproducible scientific computation, security verification, and policy validation for checking regulatory compliance. Consequently, applications are being coupled with suitable provenance middleware that can audit events, read logs, and answer provenance-related questions.

Tanu Malik
University of Chicago, Illinois 60637, USA e-mail: `tanum@ci.uchicago.edu`

Ashish Gehani
SRI International, California 94025, USA e-mail: `ashish.gehani@sri.com`

Dawood Tariq
SRI International, California 94025, USA e-mail: `dawood.tariq@sri.com`

Fareed Zaffar
Lahore University of Management Sciences, Punjab 54792, Pakistan e-mail: `fareed.zaffar@lums.edu.pk`

We are particularly interested in provenance infrastructure that is used with applications that perform distributed computation. In this context, consider some examples that give rise to a variety of interesting issues: (i) scientific applications decompose data-intensive problems into subtasks and distribute them across a Grid through a workflow planner that may not track provenance; (ii) scientists who conduct distributed experimental analyses on a variety of research hardware, such as mass spectroscopes, DNA sequencers, or oscilloscopes, must maintain records of the combined analyses for reproducibility; (iii) when different users share data through network connections, the resulting information generated has distributed provenance that may be drawn from multiple, independent administrative domains.

A characteristic feature of such distributed applications is that they are often conducted in loosely controlled environments and use heterogeneous software platforms. It is therefore important to collect such provenance metadata in an application-agnostic manner. The Open Provenance Model (OPM) provides a specification that serves this purpose and allows provenance to be exchanged between systems through a generic vocabulary [27]. Tracking distributed computations at the operating system level allows coupling between the filesystem's state and the associated provenance metadata [32, 11]. A significant implication of this design choice, however, is that it results in large volumes of provenance metadata [12]. Nevertheless, a number of systems, including PASS and SPADE, support transforming such provenance records into OPM.

Provenance systems that audit at fine granularity employ various architectures and mechanisms to manage the resulting metadata. Several systems [32, 4, 36] collect provenance information in centrally managed databases, often referred to as provenance stores. Benefits of aggregating provenance information in central stores include the ease of maintenance and curation, storage efficiency, and access control [17]. These mechanisms, however, also introduce significant network overhead, with many provenance records being transferred to the central provenance store, although remote queries for them may never arise [12]. Accordingly, it is important for distributed applications to account for the location where provenance metadata is collected, processed, stored, and consumed.

Support for Provenance Auditing in Distributed Environments, `SPADE` [37] is a data provenance management system. `SPADE`v2 refers to the second generation of the system, which has modular components for gathering, integrating, filtering, storing, and querying data provenance. Except for the components that gather provenance, the rest are completely agnostic to the source domain. `SPADE` uses Reporter modules customized to the provenance domain to transform the specific semantics into an OPM compliant form. The domain can be a particular application, the operating system, or even manual curation. To manage the resulting provenance, `SPADE` embodies a decentralized model, with each distributed host maintaining the authoritative repository of provenance metadata collected on it. `SPADE`v2's modules for tracking operating system activity record not only data flow dependencies between files and processes but also data movement across systems via network connections. All provenance information is stored in a local database.

Distributed provenance management systems, such as SPADE, face a significant challenge when reconstructing data provenance that spans multiple hosts. The problem is often solved by tracing a path or recursively querying metadata that is manifested as a directed graph. Recursive querying is known to have poor response times for large provenance graphs [20]. In the case of distributed provenance, it is also expensive in terms of network operations since the provenance metadata is unlikely to be located where the data is stored, and the appropriate remote sources must be identified. The alternative to recursive querying is computing a transitive closure, which is computationally expensive. In addition, this requires global knowledge, which raises traditional distributed system challenges.

SPADE employs *provenance sketches* to address the problem of reconstructing distributed data provenance. Such provenance can be viewed as a collection of subgraphs, each from a different host, that interface through vertices corresponding to network connections between the hosts. The provenance sketches determine which network connections are relevant to a query, while locally computed transitive closures provide host-specific subgraphs that must then be stitched together. In our earlier work [24], provenance sketches summarized host-specific provenance subgraphs with Bloom filters [2]. In contrast, we now encode an entire provenance graph by organizing a set of Bloom filters into a new data structure that we term a *matrix filter*. Matrix filters, when propagated to other downstream hosts, determine in a single lookup the existence of a path between any two distributed hosts, which would previously have required contacting multiple hosts. If the path exists, the matrix filter can also be used to determine the specific remote hosts that contain the intermediate path. This allows us to contact the intermediate remote hosts in parallel to construct the full provenance path rather than building the path one remote host at a time. The parallel operation substantially improves the performance of distributed path queries.

We deployed SPADE to collect fine-grained provenance of workflows used in the NIGHTINGALE project [30]. The project uses heterogeneous machine learning algorithms to translate information from multiple languages so that monolingual users can query the content. The provenance of intermediate outputs is used when comparing the quality of competing approaches. We mapped the provenance metadata to distributed SPADE databases, and constructed representative provenance queries. SPADE was augmented with functionality to compute the provenance sketches needed for each host. Our experiments indicate that queries are answered accurately with the aid of matrix filters. Query response times remain constant even when the number of levels in the provenance increases.

The remainder of the paper is organized as follows. Section 4.2 describes provenance systems for distributed applications. Section 4.3 outlines the SPADE architecture and data model for auditing system-level provenance and storing it in distributed repositories. Section 4.4 describes sketches for encoding graphs. In particular, it describes the matrix filter and how it can be used for improving the latency of provenance queries in a distributed provenance system, such as SPADE. Section 4.5 reports our findings about the use of matrix filters to improve the efficiency of SPADE queries in a PlanetLab [31] distributed environment. Section 4.6 concludes.

4.2 Related Work

Distributed applications manage and query digital provenance in a variety of ways. Chimera [8] uses a virtual data catalog to store information about Grid data objects, transformation types, and applications. Swift extends concepts in Chimera to include a custom provenance data store with an SQL-like language [10]. ES3 [9, 25] and PASOA [14, 16] record the provenance of files in distributed services, but provide minimal query interfaces. Karma explores a service-oriented architecture for collecting provenance metadata about workflows. It employs basic recursive traversal to enhance query capabilities [36]. Service-oriented Grids also gather provenance at multiple locations using distributed protocols [15].

One primary issue that arises with distributed data artifacts is how they should be semantically described and referenced. OPM facilitates interoperability between systems by providing a common model for provenance. Several projects provide OPM-compliant provenance, such as `SPADE`, PASS [33], VisTrails [4], and Tupelo [40]. More recently, an OPM profile (which is a set of conventions) models aspects such as transactions in distributed systems [18].

Not all systems, however, provide a combined comprehensive recording and querying infrastructure. The PASS project developed the provenance query language (PQL) [21]. PQL, however, does not interact with the distributed provenance gathering system PA-NFS [33] that enhances NFS to record provenance in local area networks. ExSPAN [41] allows the exploration of provenance in networked systems. Both systems use provenance metadata to answer queries about the origin of data and how it was derived. The ExSPAN scheme extends traditional relational models for storing and querying provenance metadata, while `SPADE` supports both graph and relational database storage and querying. Queries in ExSPAN are not optimized for performance.

ProQL [22] is a query language for provenance graphs and presents a convenient way of exploring tuples and nodes, and the ability to isolate and request portions of the graph. Similarly, D-PQuery [17] allows fetching of portions of a provenance graph in a distributed setting. However, the efficiency of queries is not addressed. ExSPAN explores storage and query optimization techniques to reduce communication latency and bandwidth, and employs caching of provenance metadata to improve query performance [41]. Caching assumes locality over the incoming query pattern. `SPADE` employs summary data structures to improve the performance of each distributed query.

An issue related to distributed provenance querying is the identification of objects uniquely across different administrative boundaries. PASS describes global naming, indexing, and querying in the context of sensor data [34]. `SPADE` addresses the issue by using storage identifiers for provenance vertices that are unique to a host and requiring distributed provenance queries to disambiguate vertices by referring to them by the host on which the vertex was generated as well as the identifier local to that host.

Given the data-intensive nature of managing provenance metadata, providing adequate storage can be a challenge. PASS explores storing provenance in highly available fault tolerant environments, such as clouds [34, 35]. SPADE employs flexible provenance storage, including graph databases, installed on the hosts where the provenance is generated. Provbase [1] uses Hbase, an open-source implementation of Google's BigTable [5] to store and query scientific workflow provenance. Provenance metadata is exported to Hbase as RDF triples, and SPARQL is used to query Hbase using its native API. Storing provenance concisely has also been investigated elsewhere [41, 20] and remains an active area of research.

4.3 Tracking System-Level Provenance with SPADE

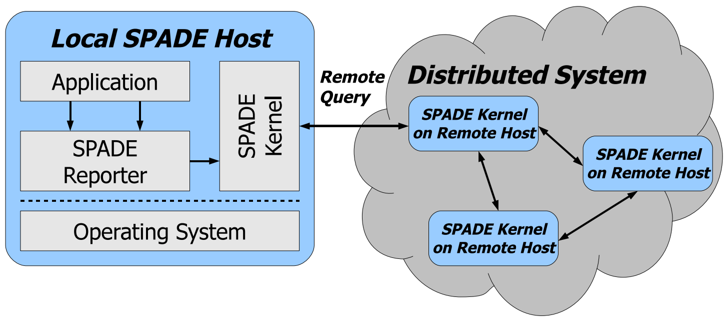

Fig. 4.1 The SPADE kernel provides an independent provenance middleware service on each host in the distributed system. A *SPADE Reporter* is a module that transforms records from a domain, such as operating system activity, into an OPM-compliant representation of data provenance. Distributed provenance queries are transparently handled by the local service, which contacts remote daemons as needed.

SPADEv1 refers to an initial implementation that enabled provenance questions about executed processes and files that were read and written by them. SPADEv2 is a second generation of the system and adopts the OPM model. It has a kernel, storage, querying, and filtering that is agnostic to the source of provenance, with domain-specific annotations created in *Reporter* modules, as illustrated in Figure 4.1. The rest of this paper focuses on the use of operating system Reporters that audit filesystem reads and writes, process execution, and TCP connections, and transform them to OPM.

In a distributed environment each computer has the freedom to maintain an independent filesystem and accompanying namespace, and yet data can be shared across organizational boundaries. The provenance recording infrastructure must overlay a coherent framework that facilitates reasoning about the origins of data in such a distributed environment. In particular, the infrastructure must track data flows within a host — that is, intra-host dependencies, and across hosts — that is, inter-host dependencies. We now describe how we record both intra-host and inter-host operating system dependencies with `SPADE`.

Recording provenance by tracking data flows requires the system to (i) identify the producers and consumers of each piece of data, and (ii) define the granularity at which a piece of data will be tracked. On a single host, the immediate source of a piece of data will be a process, which may in turn (recursively) have used data written by other processes that have executed on the same host. In addition to the data flowing within a single host, processes may have read data from other hosts through network connections. In such an event, the provenance of any data modified by a process must also include the provenance of the data read from the remote host. We adopt the convention of identifying data by both its location in the system and the time at which it was last modified.

The granularity at which we track the provenance of a data object affects the overhead that will be introduced in the system. The advantage of fine-grain auditing, at the level of assembly instructions or system calls, for example, is that information flow can be traced more precisely, allowing an output's exact antecedents to be ascertained by reconstructing the exercised portion of the control flow graph of the relevant process. The disadvantage is that the system's performance will perceptibly degrade and the monitoring will generate large volumes of provenance metadata. Since persistent data is managed at file granularity, a reasonable compromise on the level of abstraction at which to track data provenance is to define it in terms of files read and written.

4.3.1 Intra-host Dependencies

We utilize the following elements to model intra-host dependencies in a provenance graph:

- *Process vertices* are initialized when the auditing system first encounters a process. Each vertex contains a range of attributes, including the name of the process, its operating system identifier, owner, and group. Each vertex also records the parent process, the host on which the process is running, the creation time of the process, the command line with which it was invoked, and the values of environment variables. We do not version process vertices as the state changes (when an environment variable is updated, for example), although this could be useful for long-running processes such as server daemons.

- *File vertices* include various attributes associated with a file, including the host on which it resides, its pathname in the host's filesystem, the size of the file, the last time it was modified, and optionally a hash of the file's contents and a digital signature by the file's owner to attest the integrity of the hash. When the provenance of a file is being discussed, the *sink* of the associated provenance graph will be the vertex corresponding to the file. We adopt the convention of identifying a file using both its logical location and its last time of modification to disambiguate different versions of the same file, which avoids data dependency cycles in the provenance graph.

- *Edges* in a provenance graph are directed, signifying the direction of the data dependency. An edge to a file vertex indicates that the file was read, while an edge from a file vertex indicates that the file had been modified. Analogously, an edge from a process indicates that a read operation was performed by the process, while an edge to a process vertex reflects a write operation. Consequently, read and write operations to and from the filesystem by a process can be modeled by a provenance graph.

In the context of provenance, we define the semantics of a *primitive operation* to be an output file, the process that generated it, and the set of input files it read in the course of its execution. For example, if a program reads a number of data sets from disk, computes a result and records it in a file, a primitive operation has been performed. If a process modifies a number of files, a separate instance of the representation is used for each output file.

Primitive operations are combined into a *compound operation*. For instance, if the result of appending together several data sets (by a program such as UNIX *cat*) is then sorted into a particular order (using another program, such as UNIX *sort*, that executes as a separate process), then the combination of appending and sorting is a compound operation. Thus, the provenance of every file can be represented by a compound operation that is a directed graph, consistent with the model used by Grid projects [39].

4.3.2 Inter-host Dependencies

We now consider a simple example where an operation spans multiple hosts. A user with identity **501** on the machine with IP address **10.12.0.55** uses *ssh* to connect to a remote host. The user runs the UNIX *cat* program to output the contents of the file **/var/log/remote_httpd.log**. The output is redirected into the file **/tmp/local_httpd.log** in the filesystem of the host where the *ssh* command was invoked. This effectively copies the contents of the remote file to the local file.

```
% ssh 501@10.12.0.55 cat /var/log/remote_httpd.log > /tmp/local_httpd.log
```

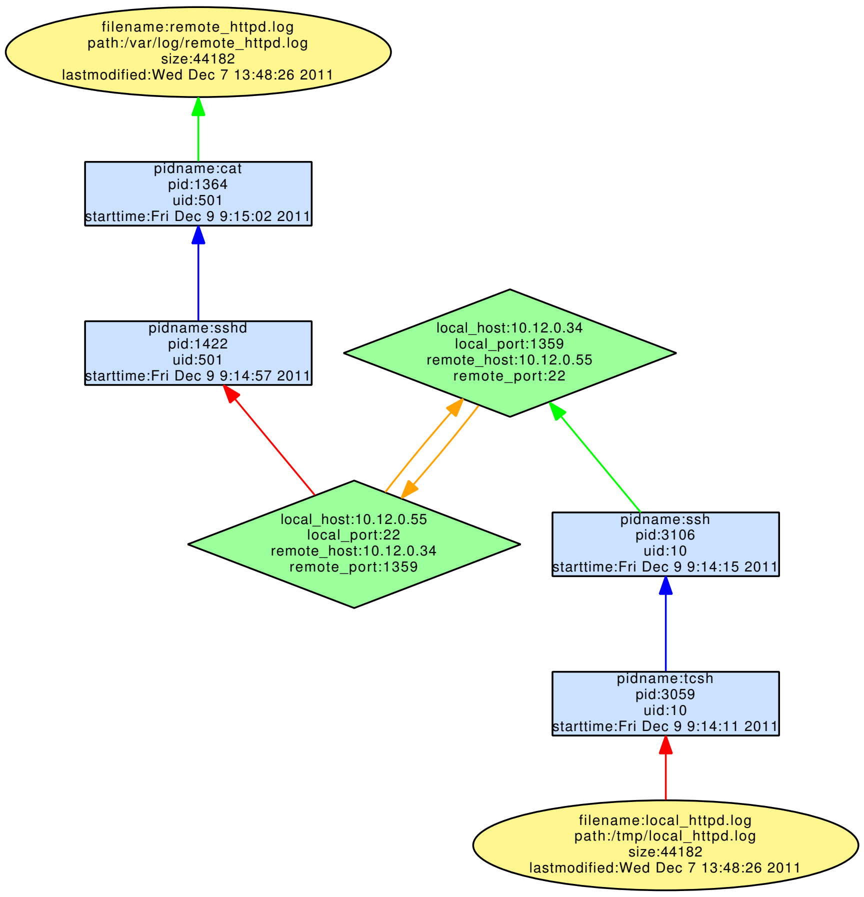

Fig. 4.2 A vertex shown with a rectangle represents the execution of a process, while a vertex shown with an ellipse represents a file that was read or written. A network vertex, depicted using a diamond, has the property that its attributes can *independently* be inferred at both ends of a connection. TCP connections used for protocols such as ssh, FTP, HTTP, or Java RMI allow the construction of such network vertices.

Similar commands and analogous file transfer utilities like *sftp*, FTP, or GridFTP are commonly used in large distributed computations to move input data to idle processors and to retrieve the results after the execution completes. If the provenance tracking was restricted to inter-host dependencies, queries about the provenance of the file **/tmp/local_httpd.log** would not be able to establish a relationship to the file **/var/log/remote_httpd.log** on the machine **10.12.0.55**.

One approach to addressing the gap described above is to record information about the host on which each process runs and where each file is located. Users can then be provided a mechanism for transferring the provenance metadata when a file moves from one computer to another. Records that refer to the part of the provenance graph that originated on a remote host will be explicitly disambiguated using the *host* attribute. While this scheme ensures that all provenance queries can be answered at the destination host, it incurs considerable storage overhead [11].

An alternate approach would avoid replicating the provenance records at the destination host to which the file is being transferred. Instead, the provenance store at the destination would be provided with a pointer back to the relevant provenance metadata on the source host. However, provenance queries at the destination would require the source hosts to be contacted, slowing the response time and decreasing reliability (since remote hosts may be unreachable).

In the above example, a distributed data flow takes the form of a file transfer. In practice, data may also flow through network connections directly from one process to another, as is the case in service-oriented architectures. In such systems, a series of HTTP calls is made from one host to another, each passing XML documents that include requests and arguments, and corresponding XML responses with return values.

- To model network flows, we introduce a fourth type of element in provenance graphs — the *network vertex* — that has the property that its attributes can *independently* be inferred in two or more processes. If the processes are being audited by different provenance middleware, the property ensures that each system can construct an equivalent network vertex without any explicit coordination. For example, equivalent network vertices associated with a TCP connection can be constructed at both endpoints using the local IP address and TCP port, remote IP address and TCP port, and timestamp (including the date), as illustrated in Figure 4.2.

Figure 4.2 depicts the provenance graph for the file **/tmp/local_httpd.log** that would arise after execution of the *ssh* command described earlier. (The graph is simplified for clarity.) The key point to note is that the provenance vertex for the network connection (between *ssh* and *sshd* in the example) can be independently constructed by both the hosts at the two ends of the network connection. This allows complete decentralization of the provenance recording in the distributed system, with each host's provenance infrastructure operating independently. At the same time, the provenance records generated can be pieced together to yield a coherent and complete reconstruction of the distributed data flows.

4.4 Querying Provenance

SPADE provides a query client that can be used to inspect the provenance metadata generated by the operating-system-level Reporters to ask questions about processes that ran, the files they read or wrote, and the network connections they initiated or handled. In particular, this can be used to answer the questions asked in the First Provenance Challenge of the International Provenance and Annotation Workshop (IPAW) [26]. SPADEv2 supports a variety of storage formats for the provenance metadata. This includes the default storage in the graph database Neo4j [29], the embedded relational database H2 [19] (and with minor changes, MySQL [28]). The client can interrogate each storage with the underlying database's query language, as well as custom provenance queries.

All of the domain-specific semantics are recorded as annotations of the OPM vertices. `SPADE`v2 requires the user to specify the hosts on which the known elements of the query are present. It maps these elements to globally unique provenance identifiers in an initial phase, and then uses its auxiliary data structures to operate on provenance graphs that are represented in terms of these identifiers.

The provenance queries in the IPAW challenge can be classified into those that require access to (i) the entire provenance graph of the output file, (ii) just a subgraph of the provenance of a vertex, or (iii) a path in the provenance graph between specific input and output vertices. These categories lead to the following query specifications: (a) given a vertex, request its entire provenance graph, (b) path expressions with vertex attributes that include process and file identifiers, (c) given a vertex, request its provenance subgraph up to k levels, and (d) check if a path exists between two vertices, s and t. We will focus on provenance path queries of type (d) as they are the most general and often have high latency.

Provenance path queries can be answered recursively – by following a pointer, corresponding to the direction from which data had flowed – or by computing the transitive closure over the entire graph. It has been shown experimentally that standard recursive graph traversal algorithms do not scale for large workflow processes and for large collections of data sets [20]. The alternative method of computing the transitive closure over the entire provenance graph is computationally expensive and has a large storage overhead [7]. When the graph is distributed, computing the transitive closure is a complex operation. An efficient method for computing the transitive closure has been described [20], but it is not clear how it translates to a decentralized scenario.

We adopt a hybrid approach for answering distributed provenance path queries. Across distributed hosts, the query is computed recursively. Within a host, the query is computed using the transitive closure, an operation that is natively supported if the storage used is a graph database.

We improve the efficiency of recursive querying with sketches that help reduce the number of hosts that must be contacted when constructing the response to a provenance path query. The sketches are space-efficient representations of graphs and are used in `SPADE` to track connections between network vertices. In the rest of this section, we describe how provenance sketches are constructed and how they are used in `SPADE` to efficiently answer provenance path queries.

4.4.1 Provenance Sketches

Consider a graph, such as the one at the top of Figure 4.3, that depicts the provenance of a piece of data on a single host in terms of identifiers for file and process vertices and edges representing the data dependencies. We introduced the notion of a provenance sketch [13] to allow such a graph to be succinctly represented. Depending on how the sketch is constructed, it can support a specific set of queries.

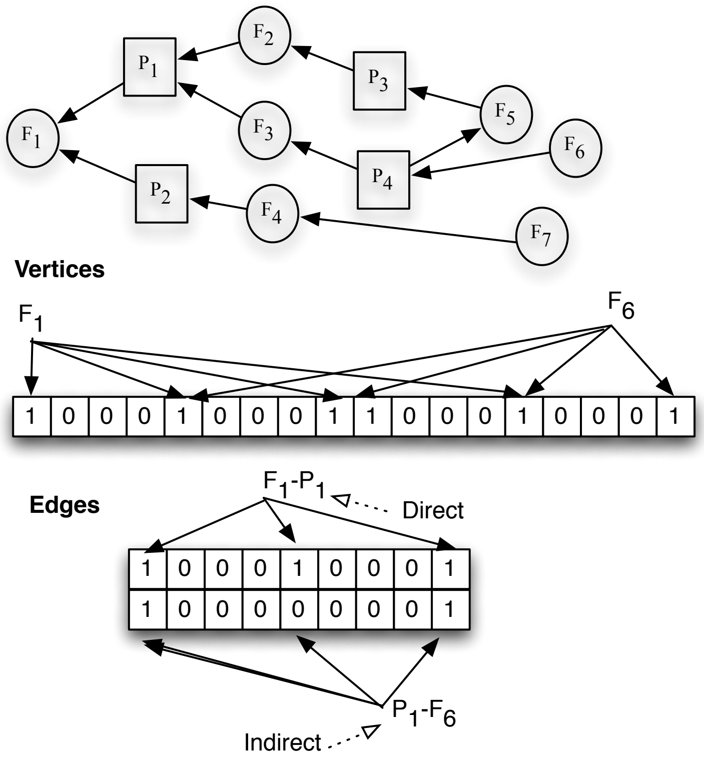

Fig. 4.3 A sketch of a provenance graph can created by inserting the hash of each vertex in a Bloom filter, the hash of each direct edge in a second Bloom filter, and the hash of each indrect edge in a third Bloom filter. However, this construction has limitations.

The sketches we will describe use Bloom filters as building blocks. A Bloom filter is a compact data structure that provides a probabilistic representation of a set. It supports membership queries – that is, queries that ask "Is element X in set Y?", denoted with the predicate $inSet(Y,X)$. Given a set $A = a_1,\ldots,a_n$ of n elements, a Bloom filter uses a vector v of m bits, with all bits initially set to 0, and k independent hash functions, $h_1,\ldots,h_k$, each with a range $1,\ldots,m$. For each element $a \in A$, the bits at positions $h_1(a),\ldots,h_k(a)$ in v are set to 1. Inserting an element only requires it to be hashed with the k functions. Each of the k outputs determines a bit in the vector v that should be set to 1 (if it is currently 0). Given a query for b, the bits at positions $h_1(b),\ldots,h_k(b)$ are checked. If any of them is 0, then certainly b is not in the set S. Otherwise, b is conjectured to be in the set with probability $(1-e^{-kn/m})^k$. The possibility of a false positive is what causes this probability to differ from 1.

We considered a number of methods to encode graphs with Bloom filters, each enabling a different set of queries. The simplest approach is to use the vertices of the graph as elements of the set S [13]. However, this only enables set membership queries, such as $inSet(S, F_1)$ and $inSet(S, P_1)$ where F_1 is a file vertex and P_1 is a process vertex. This cannot support queries about a path in a provenance graph, such as $P_1/F_2/\star/P_2/F_4$, which determines if file F_2 generated by process P_1 is part of the provenance of file F_4, generated by P_2. To enable path queries, an alternative approach is to store the edges of the graph as set members. This enables path queries in which all the vertices on a path are specified. However, it cannot handle regular expression path queries, which are among the provenance queries in the First Provenance Challenge.

A provenance sketch with a second filter that contains edges can answer some path queries but at the cost of topology-induced false positive [24]. To see why this occurs, consider the path query $P_1/F_2/\star/P_2/F_4$. A filter with edges as set members returns "true", since edges P_1–F_2 and P_2–F_4 are in the filter. However, file F_2 is not in the provenance of file F_4. To address this, we previously proposed an edge-based sketch for graphs. Two Bloom filters are maintained – corresponding to the sets of *direct* and *indirect* edges in the graph [24]. The set of indirect edges is obtained by computing the transitive closure of the provenance graph, as shown in Figure 4.3. Such an edge-based Bloom filter correctly answers all path queries (other than the false positives from the underlying Bloom filters).

The above provenance sketch construct does not capture all the ancestral relationships of a vertex – in particular, ancestors that are on other hosts are not encoded. To answer a distributed path query, the construct would require `SPADE` to maintain an additional table of cross-edges between network artifact vertices. Below we introduce a provenance sketch that encodes all the ancestral relationships of vertices and supports queries about whether a path exists between two vertices, regardless of whether they are on the same host or on different hosts.

4.4.2 Matrix Filter

We introduce the *matrix filter*, a new data structure to probabilistically represent graph connectivity (or any other data that can be stored in a matrix). Whereas the original matrix may have a size of $\mu \times \mu$ for an arbitrary μ, the filter only uses $O(m \times m)$ space for a fixed m, thereby providing a compact representation of the matrix.

A matrix filter consists of (a) a *row array* of m bits, (b) a *column array* of m^2 bits, and (c) k independent hash functions $\{h_1, \ldots, h_k\}$, each of which has a range of $\{1, \ldots, m\}$. Further, the i^{th} bit b_i of the row array defined in (a) is associated with the i^{th} set of m bits in the column array defined in (b).

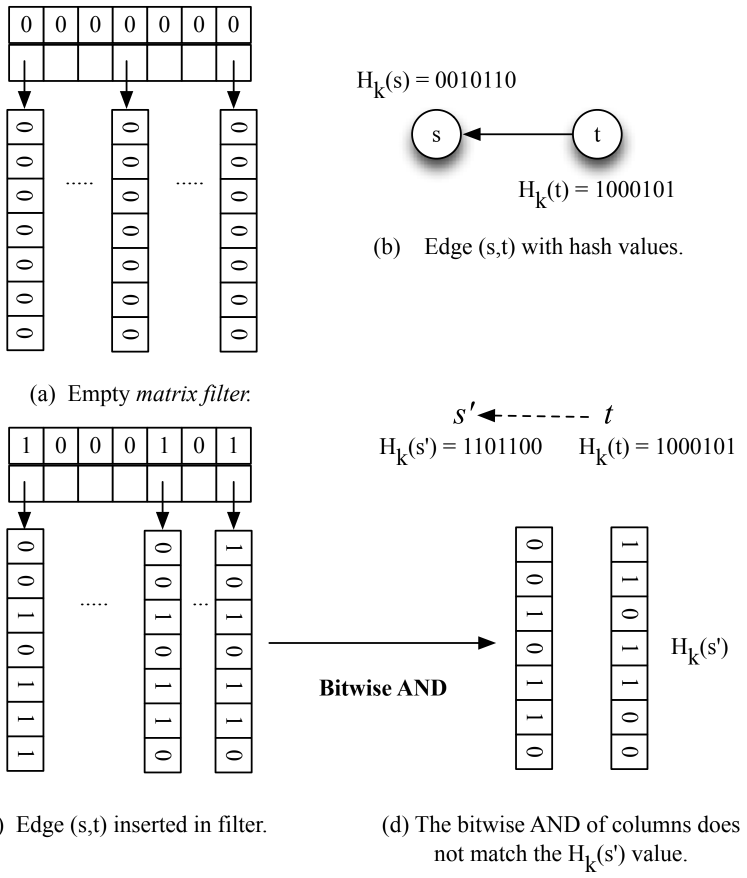
0 0 0 0 0 0 0
Hk(s) = 0010110
s
t
Hk(t) = 1000101
(b) Edge (s,t) with hash values.
(a) Empty matrix filter.
s' ← - - - - - - t
Hk(s') = 1101100 Hk(t) = 1000101
1 0 0 0 1 0 1
Bitwise AND
Hk(s')
(c) Edge (s,t) inserted in filter.
(d) The bitwise AND of columns does not match the Hk(s') value.

Fig. 4.4 A graph edge (s,t) is stored in a *matrix filter*. The filter is then used to determine that edge (s',t) is not in the graph.

Assume that $\mathscr{S}$ is the set of direct edges in the provenance graph $G = (V, E)$. During an initial setup phase, the edges in $\mathscr{S}$ are inserted into the filter. Consider the insertion of a direct edge $(s,t) \in E$ – that is, s is the parent of t – into the matrix filter, as showin in Figure 4.4(b). First k hash functions are applied to t and each resulting value sets the bits in the row array defined above in (a). Thus, $H_k(t)$ sets one or more of the m bits. For each bit in the row array that is set by $H_k(t)$, k hash functions are applied to s. Each resulting value sets one or more of the i^{th} set of m bits in the column array. Figure 4.4(c) shows that when edge (s,t) is inserted into the empty filter, $H_k(t)$ sets the bits in positions 1, 5, 7. Each of these bits is associated with a set of m bits, which is updated with the value of the hash of s, $H_k(s) = 0010110$.

To check if an edge, (s',t), is present in the graph, the k hash functions are applied to t. The bits indexed by $H_k(t)$ are checked in the row array. If they are all set to 1,

t is potentially in the graph. Each such bit is associated with a set of m bits in the column array. All these sets of m bits are combined by a bitwise AND. A vertex s' is potentially an ancestor of t if all the bits determined by the hashes $H_k(s')$ are in the bitwise AND. If any of the row array bits is set to 0, this indicates that t is not a child vertex. If any of the bits in the computed bitwise AND is 0 but was pointed to by one of the hashes of s', this indicates that s' is not a parent vertex of t. In Figure 4.4(d), the test for edge (s', t) fails and thus the edge is not in the graph.

In the matrix filter, vertices of an edge are hashed to distinct arrays – one vertex to the row array and the other to the column arrays indexed by the set row array bits. By encoding the ancestry relationship of the edge into different arrays, the matrix filter can compute if a path exists between any two vertices of the graph without computing the graph's transitive closure. We demonstrate this property of the matrix filter through examples:

- A completely specified path query, such as $s/t/u/v$, can be checked by issuing set membership subqueries (s,t), (t,u), and (u,v). If the set membership test is true for all subqueries, then the path exists.

- A path query specified with a regular expression $s/*/t$ can be performed by first checking if t is in the row array of the matrix filter. If t is present, then a check is done to see if s is in the bitwise AND of the corresponding subsets of the column array of the matrix filter.

4.4.3 SPADE's Use of Matrix Filters

To collect the provenance of data created by a distributed application, SPADE is deployed on all the hosts where the application executes. Each instance operates independently, creating its own matrix filter to represent the provenance graph for the host on which it resides. Detailed provenance metadata in the form of process, file artifact, and network artifact vertices are collected in a local provenance store. Intra-host provenance queries can be resolved using these provenance stores. What remains then is to determine the cross-host provenance relationships, which are captured by the set of network artifact vertices in a distributed provenance graph. Hence, the matrix filter on each host is used to store the set $\mathscr{S}$ of edges between network artifact vertices and their ancestor vertices that are also network artifacts. Each network artifact is stored in the row array of the host's matrix filter. Its ancestor network artifact vertices are stored in the corresponding column array locations. This includes ancestors on the same host as well as on other hosts. Figure 4.5 shows the sets $\mathscr{S}$ of edges between network artifacts that the hosts insert in their respective matrix filters.

Provenance path queries whose end points reside on different hosts must determine the exact hosts through which their path traverses. Otherwise, computing them will result in a commensurate number of (high latency) network connections on several hosts. Since network artifacts connect hosts, we observe that storing edges

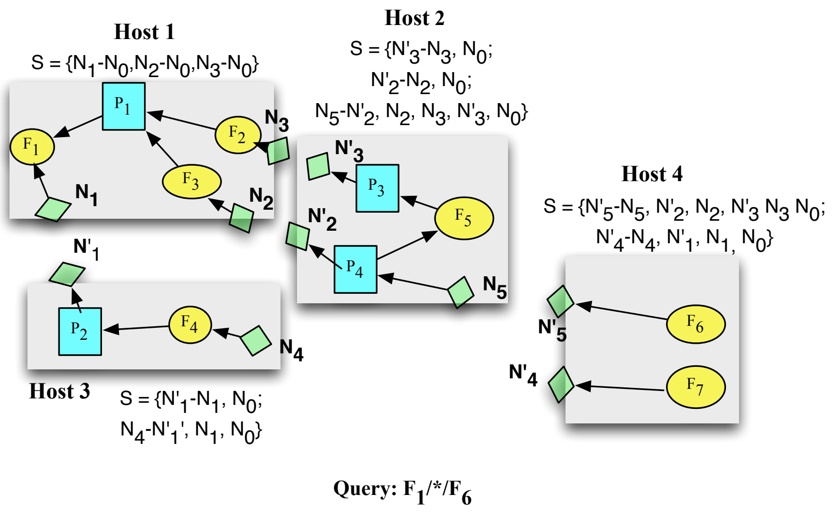
Host 1
S = {N1-N0,N2-N0,N3-N0}
Host 2
S = {N'3-N3, N0; N'2-N2, N0; N5-N'2, N2, N3, N'3, N0}
Host 3
S = {N'1-N1, N0; N4-N'1', N1, N0}
Host 4
S = {N'5-N5, N'2, N2, N'3 N3 N0; N'4-N4, N'1, N1, N0}
P1
P2
P3
P4
F1
F2
F3
F4
F5
F6
F7
N1
N2
N3
N4
N5
N'1
N'2
N'3
N'4
N'5
Query: F1/*/F6

Fig. 4.5 A provenance graph can be distributed across multiple hosts. Query $F_1/*/F_6$ arrives at host 4. The list of all hosts that must be contacted is completely determined locally using host 4's matrix filter.

only between network artifacts is sufficient to answer distributed provenance path queries. We elucidate this observation with a concrete example in the next subsection.

When a network artifact is generated on a host, SPADE adds it to the provenance store. If the new vertex is associated with an incoming network connection, the SPADE server on the remote host is contacted and that host's matrix filter is retrieved. The set of ancestor network vertices of the new vertex is extracted from the remote host's matrix filter. The new vertex and the set of ancestor vertices are added to the local matrix filter. (A performance optimization was also implemented where remote matrix filters are locally cached to avoid repeated retrieval at the cost of losing distributed consistency.) Consequently, the matrix filter of a host includes the ancestral network vertices from all upstream hosts.

A matrix filter construction also requires a judicious choice of the value of k and m. In general, a smaller k is preferred since it reduces the amount of computation [2]. The appropriate choice of m depends on the number of network connections being made. For applications with limited network connectivity, a smaller m in the range 10-50 provides low falses. Similarly, a higher m in the range of 100-500 provides low false positives for more network intensive distributed applications.

4.4.4 Querying Provenance Across Multiple Hosts

To illustrate how SPADE handles queries about provenance metadata that spans multiple hosts, we consider the case when the path is a regular expression that includes vertices *s* and *t* from different hosts. SPADE tackles distributed provenance path queries in two stages.

```
compute_query_hosts($N_s, N_t, M, C$)
begin
  $H \leftarrow \{\}$
  foreach n $_s \in N_s$
    foreach n $_t \in N_t$
      if inFilter($n_s, n_t, M$)
        print($Path\ exists$)
        mark($n_s$)
      fi
    end
  end
  foreach n $_s \in N_S$
    if isMarked($n_s$)
      foreach c $\in C$
        if $n_s \in c$
          $H \leftarrow H$ U host($c$)
        fi
      end
    fi
  end
end
```

Fig. 4.6 *M* is the matrix filter and *C* is the cache of sketches, both at the host where *t* is located. *H* is the set of hosts that will be contacted for path fragments. In the first stage, every network vertex n_s is marked if there is a descendant network vertex n_t in the matrix filter. In the second stage, a list is built of the hosts with sketches that contain any of the marked network vertices.

In the first step, the SPADE daemon on the host where the query arrives inspects its local matrix filter. The daemon determines if a path exists between the network artifacts that are the descendants and ancestors of vertices *s* and *t*, respectively. If such a path exists, SPADE determines the list of potential intermediate hosts that the path traverses. In the second step, the SPADE daemon on each of the hosts in the list is contacted to retrieve a part of the path that satisfies the query specification.

Assuming the existence of a provenance graph with components distributed across multiple hosts, each of which has its own matrix filter, we can perform a distributed provenance path query $s/*/t$ as follows:

1. Query the host where *s* is located to determine N_s, the set of network artifacts that are the descendants of *s*, and N_t, the set of network artifacts that are ancestors of *t*.

2. Execute the algorithm in Figure 4.6 to determine the list of hosts that need to be contacted.

If a path exists between the two distributed vertices, the provenance subgraphs corresponding to the query specification must be obtained. To determine the subgraphs, queries are sent to relevant remote hosts in parallel. The daemon that initiated the query receives the path fragments in response and assembles them into a single path from s to t. False positives from a sketch may result in extra path fragments in the response, necessitating careful selection of the parameter m when initializing the matrix filters.

Figure 4.5 illustrates the process of determining which hosts need to be contacted to obtain path fragments in response to a query. In this example, the query $F_1/*/F_6$ arrives at *Host 4*. The daemon on *Host 4* queries the daemon *Host 1* to obtain $N_{F_1} = \{N_1, N_2, N_3\}$, the network artifacts that are the descendants of F_1, and locally determines $N_{F_6} = \{N'_5\}$, the network artifacts that are ancestors of F_6. Using the local sketch, a check is performed to see if any path in the set $\{N_1/N'_5, N_2/N'_5, N_3/N'_5\}$ exists. The paths $\{N_2/N'_5, N_3/N'_5\}$ are present. The cached sketches are used to decide which hosts have network vertices in N_{F_1} as ancestors. If the sketch produces false positives, a small number of unnecessary network connections may still be made.

4.5 Experimental Results

Our experiments are conducted on provenance metadata that was gathered by using `SPADE` to monitor the workflow of a large distributed application in SRI International's Speech Technology and Research Laboratory [38]. The application workload originated as part of the NIGHTINGALE project [30], which allows monolingual users to query information from newscasts and documents in multiple languages. The objective of the NIGHTINGALE project is to produce an accurate translation. NIGHTINGALE aims to achieve this with a workflow that specifies the tools that will transform the inputs using automatic speech recognition algorithms, machine translation between languages, and distillation to extract responses to queries. However, there is no canonical algorithm for each of these steps, necessitating a choice between a variety of tools. The speech scientists use accompanying metadata to estimate which combination of available tools will produce the most accurate result. This is further complicated by the fact that the tools have multiple versions and are developed in parallel by experts from 15 universities and corporations. Finally, the choice of which specific version of a tool to use depends on the outcome of previous workflow runs.

A representative application workload executed for roughly half an hour with SPADE collecting provenance metadata about the processes that ran and files that were accessed and modified. The resulting provenance graph had 5256 file vertices, 5948 process vertices, 35948 edges, and a depth of 24 levels. Since the workflow was obtained from a single site (at SRI), we divided it to correspond to a distributed execution over eight geographically diverse hosts with matching network connection entries. These were PlanetLab [31] hosts located at SRI, University of Washington, and Princeton University.

To divide the workload into subgraphs corresponding to a distributed workflow, we used hMETIS [23], a graph partitioning tool. When running hMETIS, we used a high *UBfactor* (in the range of 40 to 50), which specifies that a large imbalance is allowed between partitions during recursive bisection. Since hMETIS partitions the workload by recursively creating bisections, the resulting topology over the distributed system has a tree structure, similar to what would result from a workflow planner. This also results in fewer edges between partitions, consistent with the goal of a distributed workflow planner, such as Pegasus [6]. The longest network path length in this tree was 4.

The resulting partitioned provenance graphs were then deployed on each of the eight PlanetLab hosts as SPADE databases. In addition, we constructed a synthetic workload that consists of a single linear path through nine PlanetLab hosts. The second workload provides a control to understand the effect of the sketches on the network latency of provenance queries, independent of the graph characteristics of the workload (since it consists of a series of network connections sequenced through all the hosts). In both workloads, there are 16 network vertices.

We use matrix filters with 20 bit row filters, 20 bit column filters, and 4 hash functions each. The matrix filters are built by inserting all network vertices created on a host, as well as ancestor network vertices from matrix filters of upstream hosts.

A client requests the provenance through a query, which may originate at any host in the system. A query is considered to be local to a host if the end vertex is on that host. Otherwise, the query is transported to the host where it is local. Matrix filters are used to both determine if a path exists as well as to locate the hosts that may contain parts of the provenance related to a query.

4.5.1 Reduction in Network Latency

The dominant cost of answering distributed provenance path queries comes from the network connections. To measure the reduction in network latency, we undertook the following experiment. The x axis of Figure 4.7 shows the actual number of hosts that need to be contacted to respond to a query asking for all the paths from one distributed vertex to another. The time taken to complete the request with and without the use of sketches is shown along the y axis. In particular, note that when using sketches, the latency does not increase when an increasing number of hosts must be contacted. This is true because the sketches allow all the remote hosts to be contacted in parallel, each with a suitable query, corresponding to the piece of the distributed path from that host.

While Figure 4.7 shows that using provenance sketches significantly reduces the time to answer a path query, this is seen even more dramatically in Figure 4.8 with the synthetic workload that demonstrates the effect as the query is scaled to a depth

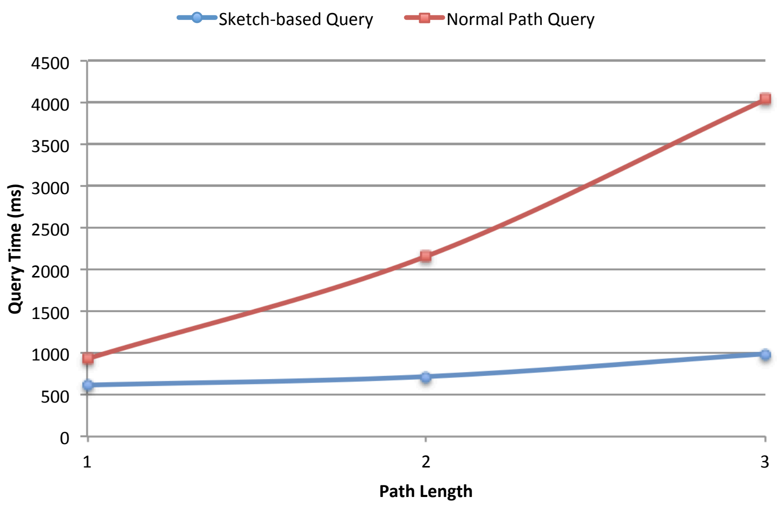

Fig. 4.7 The latency of path queries with and without the use of provenance sketches as a function of the number of hosts that must be contacted. The provenance is of a speech processing workflow.

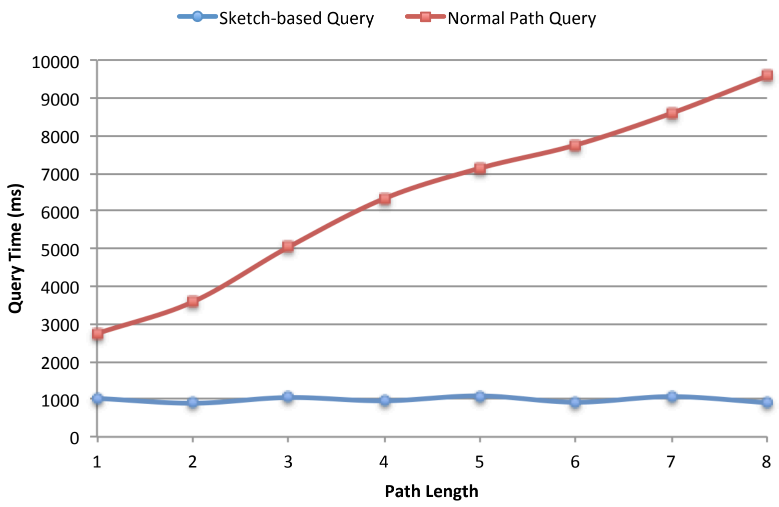

Fig. 4.8 The latency of path queries with and without the use of provenance sketches as a function of the number of hosts that must be contacted. The provenance is of a synthetic workload, consisting of a sequence of network connections through consecutive hosts.

of nine distributed hosts. The latency for answering queries remains constant regardless of the number of remote hosts that must be contacted when sketches are utilized.

4.5.2 Sketch Robustness

To understand the efficacy of our sketches as the provenance graph grows in size, we measured the number of false positive answers to queries about whether an edge between two network vertices exists in a matrix filter. Since the sketch contains provenance metadata of an increasing number of hosts as it propagates along a distributed path in the system, it is expected to provide an increasing number of false positive responses. Figure 4.9 shows that the rate of such false positives is very low, ensuring that the sketches are robust as the provenance grows. Figure 4.10 shows that this is not an issue even in the case of the more strenuous synthetic workload with longer path lengths.

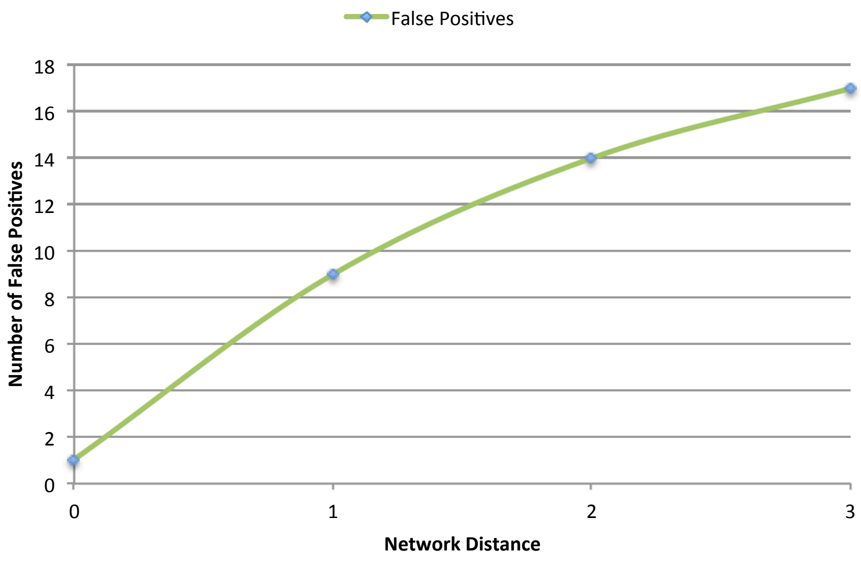

Fig. 4.9 The number of false positive responses in 100,000 random provenance queries. Each query checks if an edge exists between two network vertices using the matrix filter sketch of a host. The provenance is of a speech processing workflow.

4.6 Conclusion

`SPADE` is a system for auditing, recording, and querying the provenance of distributed applications. Domain-specific (such as operating system level) activity is transformed into an OPM-compliant record by `SPADE`v2 modules. Each host maintains the authoritative repository of its data provenance. The distributed model

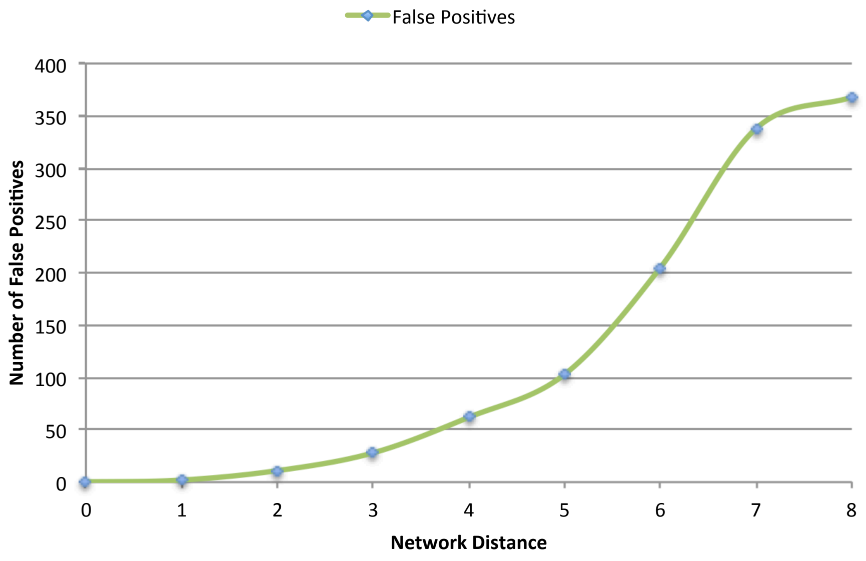

Fig. 4.10 The number of false positive responses in 100,000 random provenance queries checking if an edge exists between two network vertices using the matrix filter sketch of a host. The provenance is of a synthetic workload, consisting of a sequence of network connections through consecutive hosts.

of `SPADE` introduces the problem of reconstructing provenance during querying. `SPADE` uses novel provenance sketches to improve the performance of querying such provenance metadata. The provenance sketches determine which past network connections were relevant to a query, allowing all appropriate hosts to be contacted in parallel. Host-specific provenance subgraphs are computed using local transitive closures between network vertices. These subgraphs are retrieved in parallel and stitched together. `SPADE` is currently deployed within the NIGHTINGALE [30] project for auditing distributed workflows. Using real provenance data, we have shown the efficiency of `SPADE`'s novel matrix filter summary data structure.

Acknowledgements This material is based upon work supported by the National Science Foundation under Grant OCI-0722068 and IIS-1116414. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. J. Abraham, P. Brazier, A. Chebotko, J. Navarro, and A. Piazza, Distributed storage and querying techniques for a semantic web of scientific workflow provenance, IEEE International Conference on Services Computing, 2010.

2. B. Bloom, Space/time tradeoffs in hash coding with allowable errors, *Communications of the ACM*, Vol. 13(7), 1970.
3. A. Broder and M. Mitzenmacher, Network Applications of Bloom Filters: A Survey, 2002.
4. S. Callahan, J. Freire, E. Santos, C. Scheidegger, C. Silva, and H. Vo, VisTrails: Visualization meets data management, ACM SIGMOD International Conference on Management of Data, 2006.
5. F. Chang, J. Dean, S. Ghemawat, W. Hsieh, D. Wallach, M. Burrows, T. Chandra, A. Fikes, and R. Gruber, BigTable: A distributed storage system for structured data, 7th USENIX Symposium on Operating Systems Design and Implementation, 2006.
6. E. Deelman, J. Blythe, Y. Gil, C. Kesselman, G. Mehta, S. Patil, M. Su, K. Vahi, and M. Livny, Pegasus: Mapping scientific workflows onto the Grid, Grid Computing, 2004.
7. G. Dong, L. Libkin, J. Su, L. Wong, Maintaining transitive closure of graphs in SQL, International Journal of Information Technology, Vol. 5, 1999.
8. I. Foster, J. Vockler, M. Wilde, and Y. Zhao, Chimera: A virtual data system for representing, querying, and automating data derivation, 14th International Conference on Scientific and Statistical Database Management, 2002.
9. J. Frew, D. Metzger, and P. Slaughter, Automatic capture and reconstruction of computational provenance, Concurrency and Computation, Vol. 20(5), 2008.
10. L. Gadelha Jr., B. Clifford, M. Mattoso, M. Wilde, and I. Foster, Provenance management in Swift, Future Generation of Computer Systems, Vol. 27(6), 2011.
11. A. Gehani and U. Lindqvist, Bonsai: Balanced lineage authentication, 23rd Annual Computer Security Applications Conference, IEEE Computer Society, 2007.
12. A. Gehani, M. Kim, and J. Zhang, Steps toward managing lineage metadata in Grid clusters, 1st USENIX Workshop on the Theory and Practice of Provenance, 2009.
13. Ashish Gehani and Tanu Malik, Efficient Querying of Distributed Provenance Stores, 8th Workshop on the Challenges of Large Applications in Distributed Environments, 2010.
14. P. Groth, Recording Provenance in Service-Oriented Architectures, Report, University of Southampton, 2004.
15. P. Groth, M. Luck and L. Moreau, A protocol for recording provenance in service-oriented grids, International Conference on Principles of Distributed Systems, 2004.
16. P. Groth, On the Record: Provenance in Large Scale, Open Distributed Systems, Thesis, University of Southampton, 2005.
17. P. Groth, A Distributed Algorithm for Determining the Provenance of Data, eScience, 2008.
18. P. Groth and L. Moreau, Representing distributed systems using the Open Provenance Model, Future Generation Computer Systems, Vol. 27(6), 2011.
19. H2, `http://www.h2database.com`
20. T. Heinis and G. Alonso, Efficient lineage tracking for scientific workflows, ACM SIGMOD International Conference on Management of Data, 2008.
21. D. Holland, U. Braun, D. Maclean, K. Muniswamy-Reddy, and M. Seltzer, Choosing a data model and query language for provenance, 2nd International Provenance and Annotation Workshop, 2008.
22. G. Karvounarakis, Z. Ives, and V. Tannen, Querying data provenance, ACM SIGMOD International Conference on Management of Data, 2010.
23. G. Karypis, R. Aggarwal, V. Kumar, and S. Shekhar, Multilevel hypergraph partitioning: Applications in VLSI domain, 34th Design and Automation Conference, 1997.
24. T. Malik, L. Nistor, and A. Gehani, Tracking and sketching distributed data provenance, 6th IEEE International Conference on e-Science, 2010.
25. S. Miles, E. Deelman, P. Groth, K. Vahi, G. Mehta, and L. Moreau, Connecting scientific data to scientific experiments with provenance, 3rd IEEE International Conference on e-Science and Grid Computing, 2007.
26. L. Moreau, B. Ludaescher, I. Altintas, R. Barga, S. Bowers, S. Callahan, G. Chin Jr, B. Clifford, S. Cohen, S. Cohen-Boulakia, S. Davidson, E. Deelman, L. Digiampietri, I. Foster, J. Freire, J. Frew, J. Futrelle, T. Gibson, Y. Gil, C. Goble, J. Golbeck, P. Groth, D. Holland, S. Jiang, J. Kim, D. Koop, A. Krenek, T. McPhillips, G. Mehta, S. Miles, D. Metzger, S. Munroe,

J. Myers, B. Plale, N. Podhorszki, V. Ratnakar, E. Santos, C. Scheidegger, K. Schuchardt, M. Seltzer, and Y. Simmhan, The First Provenance Challenge, Concurrency and Computation: Practice and Experience, Vol. 20(5), 2007.

27. L. Moreau, B. Clifford, J. Freire, J. Futrelle, Y. Gil, P. Groth, N. Kwasnikowska, S. Miles, P. Missier, J. Myers, B. Plale, Y. Simmhan, E. Stephan, and J. Van den Bussche, The Open Provenance Model core specification (v1.1), Future Generation Computer Systems, 2010.
28. MySQL, `http://www.mysql.com`
29. Neo4j, `http://neo4j.org`
30. Novel Information Gathering and Harvesting Techniques for Intelligence in Global Autonomous Language Exploitation, `http://www.speech.sri.com/projects/GALE/`
31. PlanetLab, `http://www.planet-lab.org`
32. K. Muniswamy-Reddy, D. Holland, U. Braun, and M. Seltzer, Provenance-aware storage systems, USENIX Annual Technical Conference, 2006.
33. K. Muniswamy-Reddy, U. Braun, D. Holland, P. Macko, D. Maclean, D. Margo, M. Seltzer, and R. Smogor, Layering in provenance systems, USENIX Annual Technical Conference, 2009.
34. K. Muniswamy-Reddy, P. Macko, and M. Seltzer, Making a Cloud provenance-aware, 1st USENIX Workshop on the Theory and Practice of Provenance, 2009.
35. K. Muniswamy-Reddy, P. Macko, and M. Seltzer, Provenance for the Cloud, 8th USENIX Conference on File and Storage Technologies, 2010.
36. Y. L. Simmhan, B. Plale, D. Gannon, and S. Marru, Performance evaluation of the Karma provenance framework for scientific workflows, 1st International Provenance and Annotation Workshop, 2006.
37. Support for Provenance Auditing in Distributed Environments, `http://spade.csl.sri.com`
38. Speech Technology and Research, SRI International, `http://www.speech.sri.com`
39. D. Thain, T. Tannenbaum, and M. Livny, Condor and the Grid, Grid computing: Making the global infrastructure a reality, John Wiley, 2003.
40. Tupelo project, NCSA. `http://tupeloproject.ncsa.uiuc.edu/node/2`
41. W. Zhou, M. Sherr, T. Tao, X. Li, B. Loo, Y. Mao, Efficient querying and maintenance of network provenance at Internet-scale, ACM SIGMOD International Conference on Management of Data, 2010.