SGProv: Summarization Mechanism for Multiple Provenance Graphs

Daniele El-Jaick, Marta Mattoso, Alexandre A. B. Lima

Federal University of Rio de Janeiro (PESC/COPPE/UFRJ), Brazil
{deljaick, marta, assis}@cos.ufrj.br

Abstract. Scientific workflow management systems (SWfMS) are powerful tools in the automation of scientific experiments. Several workflow executions are necessary to accomplish one scientific experiment. Data provenance, typically collected by SWfMS during workflow execution, is important to understand, reproduce and analyze scientific experiments. Provenance is about data derivation, thus it is typically represented in the form of a directed acyclic graph. For each workflow execution, a provenance graph is generated. Numerous graphs are generated after several workflow runs, exploring different parameters. The resulting provenance database requires considerable storage space and querying it involves handling a large volume of graphs. Typical provenance queries process many graphs to get data derivation paths (lineage). This article proposes SGProv, a summarization mechanism for provenance graphs, using a graph database to store and query them. The goal is to generate a single small summary graph that represents all provenance graphs generated during an experiment, eliminating redundant data. This summarization approach aims to reduce the processing time of provenance queries by using only the summary graph to answer them without the need for rebuilding the original graphs. Results of provenance queries on the summary graph, from typical workflow executions, show performance improvements without data loss on query results.

Categories and Subject Descriptors: H.2 [**Database Management**]: Miscellaneous

Keywords: Graph, Provenance, Summarization

1. INTRODUCTION

Scientific experiments assisted by computers are based on simulations typically performed by chains of programs. Such chains are commonly represented by means of scientific workflows, which can be defined as formal specifications of the steps performed in scientific experiments [Deelman et al. 2009]. The exploratory nature of scientific experiments demands many executions of the same workflow on different scenarios [Taylor et al. 2007]. A scientist may need to analyze the results of running the workflow with different programs, input data or parameter combinations, like in a parameter sweep [Abramson et al. 2011]. In another scenario the scientist may widen or narrow the range of data analyzed by the workflow, or even change criteria of similarity [Ocaña et al. 2011b] or convergence of the experiment [Guerra et al. 2012]. Based on analyzes of these various workflow executions, the scientist may also exchange some of the programs, on the workflow specification, by others that are more in line with the data flow behavior that is being generated [Santos et al. 2013]. Again, this new workflow has to be executed for the same combination of parameters. Thus, for a single experiment, numerous executions of workflows are typically performed [Gil et al. 2007; Mattoso et al. 2010].

Each workflow execution generates provenance data, a trace that describes all data artifacts it used and produced as well as the transformations they suffered [Freire et al. 2008]. Provenance traces in workflows are based on objects (data and programs) and their relationships (dependencies) [Moreau and Missier 2011], being typically represented in the form of a Directed Acyclic Graph (DAG)

This article was partially funded by CNPq, Capes and Faperj

[Aggarwal and Wang 2010]. Regardless of workflow complexity, the resulting provenance graph is a DAG. The presence of conditional execution and parallelism, for example, affects the complexity of the workflow, but not the provenance DAG complexity. A provenance database is thus composed of several DAG whose nodes have varying structures that often are not known in advance. Nodes may have many dependencies between them. These characteristics bring significant challenges to provenance data storage and querying [Anand et al. 2010]. Storing and querying all DAG produced allow comparing the results obtained by exploiting the numerous variations of a workflow. Thus, scientists can identify, for example, the parameters of a particular execution and results of the workflow that used an alternative program. Typical provenance queries include getting the workflow execution "lineage" (path derivation) of results (final or intermediate). Running them efficiently (with short response time) and obtaining, as a result, a graph, or a set of graphs, that preserves the objects "lineage" relationships to answer provenance queries, are challenging [Woodman et al. 2011]. The approach often used in the literature to address the problem of provenance data storage and querying is to use a Relational Database Management System (RDBMS) [Huahai and Singh 2008; Gadelha et al. 2011; Ogasawara et al. 2011; Anand et al. 2012]. The main problems with these approaches is query specification complexity and query processing time, which, in most cases, involve many joins between tables, especially in a large set of graphs with many edges between nodes. The rigid schema of the Relational model also represents a problem for its use, since provenance data usually have flexible schemas [Davidson and Freire 2008].

This article examines the use of a Graph Database System (GDB) to store and query provenance graphs. Its data model consists of nodes, edges, and attributes. Edges have types and it is possible to have several different edges, one of each type, between the same pair of nodes. There is no rigid schema that previously defines the attributes for each node or edge, making the data storage very flexible. Almost all GDB natively provide implementations of classical graph algorithms such as traversals, breadth-first and depth-first search, and shortest path determination [Angles and Gutierrez 2008; Sadalage and Fowler 2012; Robinson et al. 2013; Haichuan and Kitsuregawa 2013], for example. Thus, using a GDB to traverse and retrieve data from various provenance graphs makes queries simpler and more efficient when compared to a RDBMS, since it does not involve joins between relations. Even with the use of a GDB, there remains the problem of handling large amounts of provenance data, which may affect query processing time. One approach to treat large graphs is summarization. Summaries can significantly reduce the size of a graph by grouping nodes and edges, but maintaining its relevant structure and enough information to answer queries applying classical graph search algorithms [Tian and Patel 2010]. Navlakha et al. [2008] propose a graph summary technique generated from the summarization of dense areas of the graph, complete bipartite subgraphs and cliques. Tian and Patel [2010] propose a summary graph where: a super-node is formed by nodes that have the same attributes with the same values; a super-edge is created between two super-nodes s_i and s_j if, in the original graph, each node of s_i has at least one edge to a node of s_j. Liu and Yu [2011] propose a summary graph where: nodes of a super-node must have attributes in common, with the same values (or similar values) and the same number (or similar number) of edges to neighboring super-nodes. However, existing solutions are dedicated to summarization of a single and generic graph with many nodes and edges. Thus, the summarization is based on graph theory or on complex similarity functions. Moreover, the data model semantic is not known in advance. On the other hand, provenance queries are usually executed on a high volume of graphs, not necessarily with many nodes and edges each. The provenance data model semantic is known and provenance graphs have similarities between them, since they are generated by executions of workflow variations. Workflow executions generate thousands of provenance graphs. Typical provenance analytical queries are "obtain provenance traces where program P2 was executed before P8". This is the case in workflows where different programs can be used. As an example, take the SciHMM workflow described by Ocaña et al. [2011a]. In one of its activities (MSA Construction), one of five programs can be used. A possible query could be "obtain the ROC curves produced when MAFFT and Kalign were used for MSA Construction". To obtain this result, the system must scan each provenance graph to check this property. Even if the algorithm traverses very fast on each single graph, it still has to manage the complexity of loading and repeating the search for each one of the provenance graphs. If the scientist decides to expand this search to previous workflow executions the number of graphs will be very large. This article adapts single graph summarization techniques to apply them in large sets of provenance graphs.

This article proposes SGProv, a summarization mechanism for multiple provenance graphs resulting from workflow executions performed during a computational scientific experiment. SGProv generates a single summary graph that represents a set of provenance graphs generated during an experiment, so that paths and nodes found in more than one graph are stored only once in the summary and their differences are highlighted. The summary graph data volume is reduced when compared to the original set of graphs. Queries may be answered from the summary, eliminating the need for original graphs reconstruction, which improves their execution time. Typical provenance queries were evaluated with SGProv, comparing the execution time using the summary graph only to that obtained using the non-summarized graphs in the original database. Results show significant performance improvements, around ten times faster, without any data loss on query results. This article is organized as follows: section 2 describes SGProv data model, section 3 shows an example of SGProv application, section 4 explains SGProv algorithm, section 5 presents experimental results and section 6 concludes.

2. SGPROV

This section describes the data model and how the summary graph is generated from the provenance graphs.

2.1 Data Model

In SGProv, a graph is defined as follows:

Definition 1 (Graph): A graph G is defined as $G = (V, E, A, T)$, where $V = \{v_1, v_2, \ldots, v_n\}$ is a set of nodes, $E = \{e_1, e_2, \ldots, e_m\} \subseteq (V \times V)$ is a set of directed edges, $A = \{a_1, a_2, \ldots, a_p\}$ is a set of attributes associated with nodes and/or edges and $T = \{t_1, t_2, \ldots, t_q\}$ is the set of types of edges. Each node $v_i \in V$ has at least one attribute $a_x \in A$. The value of a_x in the node v_i is represented by $Val(v_i, a_x)$. Each edge $e_j \in E$ has a unique type t_k defined by $Type(e_j)$, where $t_k \in T$. The value of attribute a_y belonging to an edge e_j is represented by $Val(e_j, a_y)$. Since edges are directed, $Origin(e_j)$ e $Destination(e_j)$ represent respectively, the origin and destination nodes of e_j.

A provenance graph is a special graph, defined as follows:

Definition 2 (Provenance Graph): A provenance graph G_p is defined as $G_p = (V_p, E_p, A_p, T_p)$, where the nodes of V_p represent programs or data artifacts and edges of E_p represent node "lineage". An attribute "$type$" $\in A_p$ must exist for all nodes and edges. If a node v_{pi} represents a program, $Val(v_{pi}, type)$ = "$program$". If this node represents a data artifact, $Val(v_{pi}, type)$ = "$data$". The set A_p also contains other attributes found in the provenance collection, as, for instance, programs input parameter names. Nodes that represent programs also have an attribute "$name$" $\in A_p$. The set T_p represents types of edges that can be "informed", "derived", "used", "generated", "attributed", "associate" or "acted".

2.2 Summary Graph

SGProv basic premise is that nodes that represent program executions tend to occur frequently in several provenance graphs. This premise is based on a scientific computational experiment characteristic where programs are repeatedly executed, each time with a different combination of parameters along the variations of the workflow executions. Nodes representing data show much more variation in these graphs because they correspond to the inputs and outputs of programs, which tend to vary

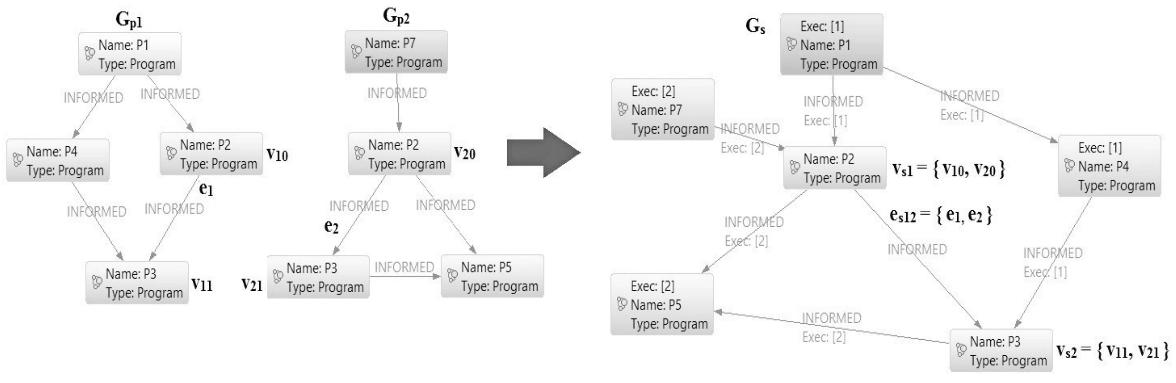

Fig. 1. SGProv example.

from execution to execution. Based on these characteristics, SGProv groups the nodes and edges of the set of provenance graphs to generate the summary graph containing all the nodes and edges of the original graphs without redundancy. In SGProv, a summary graph is defined as follows.

Definition 3 (Summary Graph): A summary graph G_s is defined as $G_s = (V_s, E_s, A_s, T_s)$. Each node $v_s \in V_s$ is called a super-node and each edge $e_s \in E_s$ is called a super-edge.

Given a set of provenance graphs $\{G_{p1} = (V_{p1}, E_{p1}, A_{p1}, T_{p1}), \ldots, G_{px} = (V_{px}, E_{px}, A_{px}, T_{px})\}$ for which one intends to produce a summary graph, super-nodes and super-edges are defined as follows.

Definition 4 (Super-node): A super-node $v_{si} = \{v_{p1}, v_{p2}, \ldots, v_{pn}\} \subseteq (V_{p1} \cup V_{p2} \cup \ldots \cup V_{px})$, where $Val(v_{p1}, type) = Val(v_{p2}, type) = \ldots = Val(v_{pn}, type) =$ "*program*" and $Val(v_{p1}, name) = Val(v_{p2}, name) = \ldots = Val(v_{pn}, name)$.

Definition 5 (Super-edge): A super-edge connects two super-nodes v_{si}, v_{sj} if there exists an edge $e_k = (v_m, v_n) \in (E_{p1} \cup E_{p2} \cup \ldots \cup E_{px})$, where $v_m \in v_{si}$ and $v_n \in v_{sj}$. This way, a super-edge e_{si} is defined as $e_{si} = \{e_{p1}, e_{p2}, \ldots, e_{pm}\} \subseteq (E_{p1} \cup E_{p2} \cup \ldots \cup E_{px})$, where $\{Origin(e_{p1}), Origin(e_{p2}), \ldots, Origin(e_{pm})\} \subseteq Origin(e_{si})$, $\{Destination(e_{p1}), Destination(e_{p2}), \ldots, Destination(e_{pm})\} \subseteq Destination(e_{si})$ and $Type(e_{p1}) = Type(e_{p2}) = \ldots = Type(e_{pm})$.

Each node (edge) that belongs to a single provenance graph G_{pk} is included in the summary as a super-node (super-edge) containing only one node (edge). Some super-nodes and super-edges have an attribute called *"exec"*, which consists of a list of workflow execution identifiers in which they took part. This attribute makes it possible to recover the paths of the original graphs and answer provenance queries based only on the summary graph. All super-nodes and super-edges of the summary graph that contain nodes and edges of the same provenance graph have the corresponding workflow identifier in their *"exec"* attributes. The absence of the attribute *"exec"* in a super-node (super-edge) indicates that it is a program (dependency) present in all provenance graphs. Figure 1 shows an example of the proposed mechanism. From provenance graphs G_{p1} e G_{p2}, the summary graph G_s is generated, where: super-node v_{s1} is created, since $Val(v_{10}, type) = Val(v_{20}, type) =$ "*program*" and $Val(v_{10}, name) = Val(v_{20}, name) =$ "$P2$"; super-node v_{s2} is created, since $Val(v_{11}, type) = Val(v_{21}, type) =$ "*program*" and $Val(v_{11}, name) = Val(v_{21}, name) =$ "$P3$"; and super-edge e_{s12} is created, since edges (e_1, e_2) exist between nodes v_{s1} and v_{s2}, where $\{Origin(e_1), Origin(e_2)\} \subseteq Origin(e_{s12})$, $\{Destination(e_1), Destination(e_2)\} \subseteq Destination(e_{12})$ and $Type(e_1) = Type(e_2) =$ "INFORMED".

3. SGPROV IN ACTION

SGProv envolves iteratively. An input set C_{Gp} of provenance graphs is iteratively summarized, one graph at a time. In each iteration, SGProv takes the summary graph produced in the previous iteration and the next graph in the C_{Gp} and produces a new summary graph. For the first iteration, since there is no summary graph yet, the first provenance graph of C_{Gp} is taken as a summary graph. The procedure continues until C_{Gp} is entirely processed and the final summary graph is produced. The next example illustrates the process. Let an input set of provenance graphs $C_{Gp} = \{G_{p0}, G_{p1}, G_{p2}\}$, where each node in the graph has two attributes called "*"name"* and *"type"* and its edges are of type "INFORMED". Figure 2 illustrates the first iteration of C_{Gp} summarization process, where the summary graph G_{s1} is generated from G_{p0} and G_{p1}. In this iteration the graph G_{p0} is assigned as the graph G_{s0} because no summary graph was been produced so far. Figure 3 shows the second iteration, where the summary G_{s1} (figure 2) and the next graph of the input set, G_{p2}, are summarized generating graph G_{s2}.

In this example, SGProv proceeds as follows. SGProv starts traversing the two graphs searching: nodes of G_{p1} in G_{s0}, that are of type "program" and have the same value for the attribute *"name"*; and edges of G_{p1} in G_{s0} with the same type (in example type "INFORMED") and source and destination nodes of type "program", where source nodes have the same values for the attribute *"name"*, as well as the destination nodes. In first iteration (figure 2), nodes that are common to both input graphs (programs with the same name) represent programs P1, P2, P4 and P5. The edges of such nodes that exist in G_{p1}, but do not exist in summary G_{s0} are inserted in summary G_{s1}, along with the attribute *"exec"* with the respective execution value to which they belong. Its value is equal to [1] (G_{p1}). For instance, the edge between nodes that represent programs P4 and P5 exists in the graph G_{p1}, but not in graph G_{s0}. Therefore, this edge is inserted in G_{s1} and the value of its *"exec"* attribute is equal to [1]. The nodes of the G_{p1} not found in G_{s0} are inserted into G_{s1} along with their edges. For example, the node that represents the program P8 does not exist in G_{s0}. Thus, that node and its associated edges are inserted into G_{s1}, and their *"exec"* attributes are set to [1]. Finally, the nodes of G_{s0} that do not exist in G_{p1} and their respective edges are inserted into the G_{s1}, along with the attribute *"exec"* equal to [0], since they belong only to G_{s0}. For example, nodes that represent programs P12 and P10, and the edge between them, exist in G_{s0}, but not in G_{p1}. So they are inserted into G_{s1} with the attribute *"exec"* equal to [0]. At the end of this iteration, the summary graph G_{s1} is generated containing all the nodes and edges of the two input graphs, but with the redundant nodes and edges summarized. In second iteration (figure 3), SGProv compares G_{p2} and G_{s1}. In G_{s2}, nodes representing programs P1, P2, P4 and P5 are present in all executions, because they lack the attribute *"exec"*. Nodes representing programs P3, P8 and P10 are common to G_{s1} and G_{p2}, but the nodes that represent the programs P3 and P10 belong to executions 0 and 2, while the node that represents the program P8 belongs to executions 1 and 2. Edges in G_{s2} also have their attribute values updated with the identifiers of the executions to which they belong to. For example, the edge between nodes P2 and P3 is common to the graph G_{s1} (exec=0) and graph G_{p2}. In graph G_{s2}, this edge has its attribute *"exec"* updated with the execution value of 2 (G_{p2}). As all of the graphs of the input set were analyzed, the summarization mechanism is finished.

By analyzing the example, it is possible to verify the potential efficiency of SGProv. In figure 2, G_{s1} was produced from G_{p0} and G_{p1}. Together, these graphs have 17 nodes and 22 edges. G_{s1} has 13 nodes and 20 edges. Thus, a reduction of approximately 23% of nodes and 9% of edges was obtained, at a cost of 27 attributes (*"exec"*). In figure 3, G_{s2} was produced from G_{s1} and G_{p2}. If the summarization was not applied, the amount of actual data would correspond to the sum of the volumes of the original graphs G_{p0}, G_{p1} and G_{p2}. These three graphs have 26 nodes and 32 edges, while Gs2 has 15 nodes and 25 edges. Thus, reductions of approximately 42% and 21% of the nodes and edges, respectively, were obtained at a cost of 35 attributes (*"exec"*). Thus, after each SGProv iteration, the reduction of the data volume becomes more significant. After $m - 1$ iterations on m provenance graphs, a final summary graph is obtained, containing all the nodes and edges of the

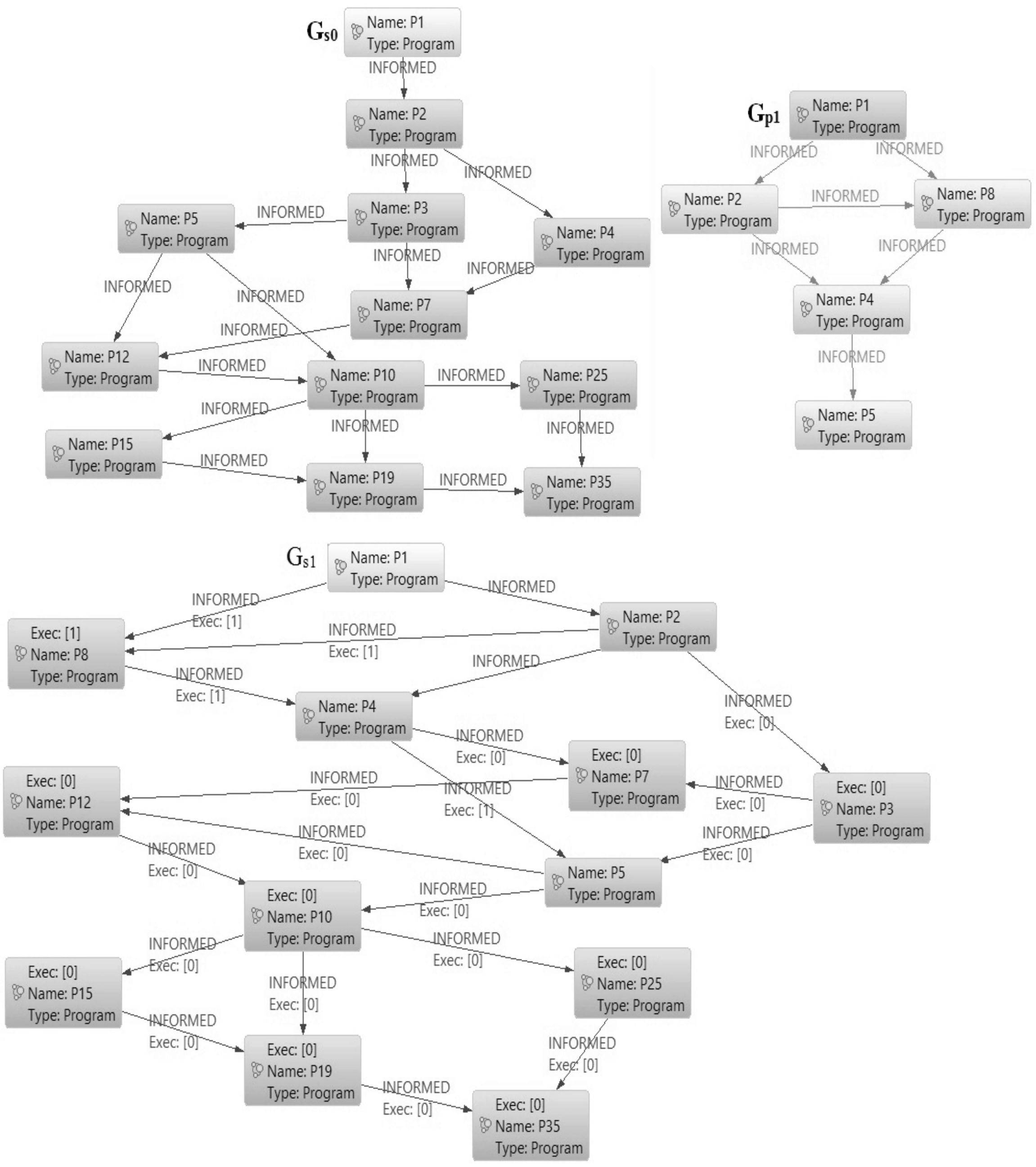

Fig. 2. SGProv first iteration.

original graphs without redundancy. In the example, by querying only the summary graph, G_{s2} it is possible to query G_{p0}, G_{p1} and G_{p2} data without the need for individually traversing each graph. For this, the values of the *"exec"* parameter of nodes and edges of G_{s2} are examined. Thus, the nodes and edges that do not have this attribute belong to all executions; those that have value 0 in their *"exec"* belong to G_{p0}; those that have the value 1 in their *"exec"* belong to G_{p1}; and those that have the value 2 in their *"exec"* belong to G_{p2}.

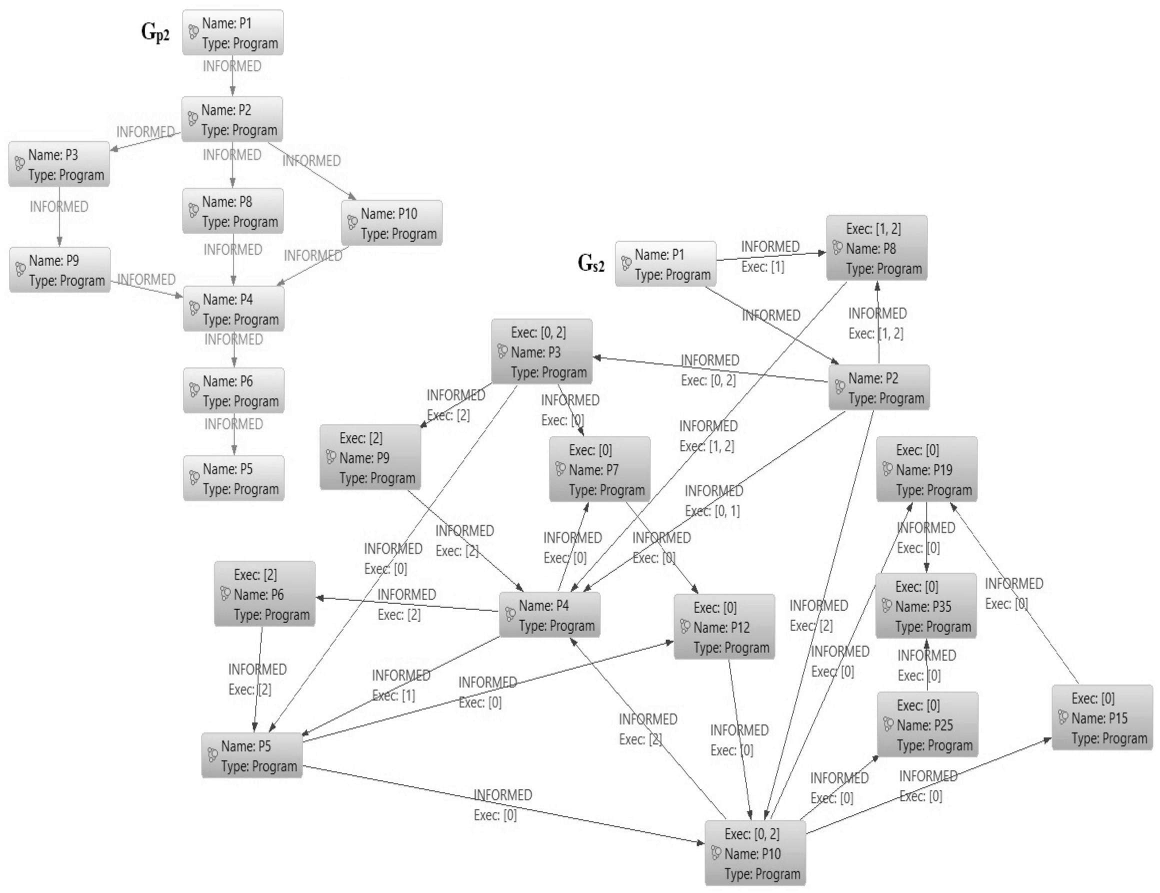

Fig. 3. SGProv second iteration.

4. SGPROV ALGORITHM

The SGProv algorithm (figure 4) takes as input a set of provenance graphs (C_{Gp}). The algorithm complexity does not affect queries, since they are executed on summary graph generated at the end of the SGProv run. SGProv is intended to be executed as a standalone process that prepares data to be used by future queries. The SGProv runs iteratively and, in each iteration, traverses, compares and summarizes the summary graph (G_s) produced in the previous iteration and the next graph from the input set, producing as output a new summary graph, an updated version of G_s. This will be used in the next iteration, until the entire set of input graphs is processed. In the first iteration, as no G_s is created, the first graph of the input set is assigned as the initial summary (line 1). Then, the algorithm runs $m-1$ iterations, where m is the total number graphs in the C_{Gp}. At the end of all iterations, the final G_s is produced. In a given iteration n, a provenance graph $G = C_{Gp}[n]$ is taken. The algorithm first searches for G nodes that have the same type and name of nodes in G_s (lines 5-23). If any node of G is found, it marks the corresponding G_s node (*"found"* attribute receives the value 1) and updates its *"exec"* attribute with the number of the current execution (lines 9-14). Then, the procedure UpdateEdges (line 16) inserts into G_s the edges of the corresponding node in G not present in the summary graph. If no corresponding G_s node is found for the G node, it is inserted into the $gNode$ list (lines 20-22). The algorithm continues until all nodes in G are compared with the nodes of G_s. In the second phase, the algorithm searches for marked nodes (found=1) in G_s and unmarks (found:=0) them. Unmarked nodes (found=0) are inserted into the $sNode$ list (lines

Table I. Characteristics of the test base and summary graph produced after applying the SGProv.

	Test Base	**Summary Graph**	**Approximate Reduction**
Number of nodes	8,490	30	99%
Number of edges	10,149	305	97%

24-30). Then, the algorithm inserts the nodes and edges of the $gNode$ list into G_s (lines 31-47) and updates the *"exec"* attribute of the nodes and edges of the $sNode$ list with the number of all previous runs, leaving the current run out of the list (lines 48-62). The n variable is incremented (line 63) and the $gNode$ and $sNode$ lists are reset (lines 64-65). The algorithm continues until there are no more provenance graphs in the input set, producing the final summary graph G_s as output.

5. EXPERIMENTAL EVALUATION

SGProv was implemented using the Neo4j[1] GDB and queries were implemented using its Cypher[2] query language. Neo4j showed good performance for storing and querying graphs from the database. However, a broader analysis about the use of GDB to manage provenance graphs requires tests with other graph-oriented DBMS. The experiments were performed on a computer with 2.4 GHz Intel Core i7 processor, 8GB RAM and 750 GB 5400 rpm HD. To perform an initial assessment of SGProv performance, a test database with 1000 provenance graphs was generated from synthetic scientific workflows executions. Some graphs have similar (but not identical) structures, and others have very different structures. They follow the Prov-DM[3] data model proposed by the W3C[4] (World Wide Web Consortium) as the standard for provenance data. On the test database, graph nodes represent programs used by scientists in their experiments, in which a program can belong to one or more graphs. Thirty distinct programs were defined for synthetic workflows that have, on average, 15 programs, and, at each run, some programs are replaced by similar ones or new programs are included. Table I shows the characteristics of the test data originally produced, those of the summary graph produced by SGProv (each one stored in a different database), and approximate data volume reduction. The databases are managed by Neo4j. Since 30 programs are repeatedly referenced by each provenance graphs, the summary graph is composed of 30 nodes. This is because the summarization mechanism groups graph nodes with the same program names. As the same dependencies (edges) between programs can be found in many graphs, the summarization mechanism groups the edges that have the same source and destination program, producing a small number of edges in the summary.

In scientific experiments, scientists need to query the provenance database to obtain information such as: the relationship between production and consumption of data sets by processes, data derivation history and intermediate results obtained. It is thus possible to interpret, evaluate and validate experiments and detect possible errors [Gadelha et al. 2011]. Typical provenance queries defined in Woodman et al. [2011] were executed on the test base and on the summary graph in order to assess the variation in processing time in both cases. Queries were implemented in Neo4j Cypher query language, and were repeated 100 times. Table II shows the queries and the average processing times obtained for each one.

Query Q1 traverses all nodes of the graph and returns the destination nodes of its edges. Running Q1 on the summary graph obtained an average processing time much lower than that obtained with graphs of the test base. This is because the data volume of the summary is much smaller than the volume of the graphs of the test base. With Q1 running on the summary graph, it would also be possible to return the number of workflow execution to which each edge of the queried nodes belongs, by adding "r.exec" in the RETURN clause.

[1]`http:\\www.neo4j.org`
[2]`http:\\www.neo4j.org\learn\cypher`
[3]`http:\\w3c.org\TR\2012\WD-prov-dm-20120202`
[4]`http:\\w3c.org\TR\prov-aq`

```
Algorithm SGProv ($C_{Gp}$, $G_s$)
Input: $C_{Gp}$: { $G_{p0}, G_{p1}, \ldots, G_{pm-1}$}, where $G_{pi}=(V_{pi}, E_{pi}, A_{pi}, T_{pi})$, i = 0 ... m-1
        // set of provenance graphs
Output: $G_s = (V_s, E_s, A_s, T_s)$ //summary graph
gNode := [] //list of provenance graph nodes not found in summary
sNode := [] // list of summary nodes not found in provenance graph
Begin
    $G_s$ := $C_{Gp}$ [0], n:=1, foundNode := 0
    While n < $C_{Gp}$.length do
        G := $C_{Gp}$ [n]  // G = ($V_g$, $E_g$, $A_g$, $T_g$)
        foundNode := 0
        For $v_g \in$ G.$V_g$ do // $v_g \rightarrow$ provenance graph node
                              // G.$V_g \rightarrow$ set of provenance graph nodes
            For $v_s \in$ $G_s$ .$V_s$ do // $v_s \rightarrow$ summary graph node
                                  // $G_s$.$V_s \rightarrow$ set of summary graph nodes
                If (Val($v_g$, type) = Val($v_s$, type) = "program")
                  and (Val($v_g$, name) = Val($v_s$, name)) then
                  Val($v_s$, found) := 1
                    If ($\exists$ $v_s$.exec) then
                      Val($v_s$, exec[++]) := n
                    End if
                    foundNode :=1
                    UpdateEdges(G, $G_s$, $v_g$, $v_s$)
                    break
                End if
            End for
            If foundNode = 0 then
              gNode[++] := $v_g$
            End if
        End for
        For $v_s \in$ $G_s$ .$V_s$ do
            If Val($v_s$, found) = 1 then
              Val($v_s$, found): = 0
            Else
              sNode[++] := $v_s$
            End if
        End for
        For $v_g \in$ gNode do
            create($v_s$)
            Val($v_s$, name) := Val($v_g$, name)
            Val($v_s$, type) := Val($v_g$, type)
            insert($v_s$, $G_S$)
            Val($v_s$, exec) := [n]
            For ($v_g$, $v_{gDest}$) $\in$ $E_g$ do
                $v_{sDest}$ := $v_{gDest}$
                insert(($v_s$, $v_{sDest}$), $G_S$)
                Val(($v_s$, $v_{sDest}$), exec) := [n]
            End for
            For ($v_{gOrig}$, $v_g$) $\in$ $E_g$ do
                $v_{sOrig}$ := $v_{gOrig}$
                insert(($v_{sOrig}$, $v_s$), $G_S$)
                Val(($v_{sOrig}$, $v_s$), exec) := [n]
            End for
        End for
        For $v_s \in$ sNode do
            If ($\nexists$ $v_s$.exec) then
              Val($v_s$, exec) := [0, ..., n-1]
            End if
            For ($v_s$, $v_{sDest}$) $\in$ $E_s$ do
                If ($\nexists$ ($v_s$, $v_{sDest}$).exec) then
                  Val(($v_s$, $v_{sDest}$), exec) := [0, ..., n-1]
                End if
            End for
            For ($v_{sOrig}$, $v_s$) $\in$ $E_s$ do
                If ($\nexists$ ($v_s$, $v_{sOrig}$).exec) then
                  Val(($v_{sOrig}$, $v_s$), exec) := [0, ..., n-1]
                End if
            End for
        End for
        n = n+1
        clear(gNode)
        clear(sNode)
    End while
End
```

```
Procedure UpdateEdges(G, $G_s$, $v_g$, $v_s$)
Input:  G: provenance graph
        $G_s$: summary graph
        $v_g$: provenance graph node
        $v_s$: summary graph node

Begin
  foundEdgeD:=0, foundEdgeO:=0
  For ($v_g$, $v_{gDest}$) $\in$ G.$E_g$ do
  // G.$E_g$ set of provenance graph edges
    For ($v_s$, $v_{sDest}$) $\in$ $G_s$.$E_s$ do
    // $G_s$.$E_s$ set of summary graph edges
      If (Val($v_{gDest}$, type) = Val($v_{sDest}$, type)) and
        (Val($v_{gDest}$, name) = Val($v_{sDest}$, name)) then
        If ($\exists$ ($v_s$, $v_{sDest}$).exec) then
          Val(($v_s$, $v_{sDest}$), exec[++]) := n
        End if
        foundEdgeD :=1
        break
      End if
    End for
    If (foundEdgeD = 0) and
      ($\exists$ {v $\in$ $G_s$.$V_s$ | Val(v,name) = Val($v_{gDest}$, name) and
      Val(v.type) = Val($v_{gDest}$, type)} ) then
      $v_{sDest}$ := find v in $G_s$.$V_s$ where
            Val(v,name) = Val($v_{gDest}$, name) and
            Val(v,type) = Val($v_{gDest}$, type)
      insert(($v_s$, $v_{sDest}$), $G_s$)
      Val(($v_s$, $v_{sDest}$), exec) := [n]
    End if
  End For
  For ($v_{gOrig}$, $v_g$) $\in$ G.$E_g$ do
    For ($v_{sOrig}$, $v_s$) $\in$ $G_s$.$E_s$ do
      If (Val($v_{gOrig}$, type) = Val($v_{sOrig}$, type)) and
        (Val($v_{gOrig}$, name) = Val($v_{sOrig}$, name)) then
        If ($\exists$ ($v_s$, $v_{sOrig}$).exec) then
          Val(($v_s$, $v_{sOrig}$), exec[++]:= n
        End if
        foundEdgeO :=1
        break
      End if
    End for
    If (foundEdgeO = 0) and
      ($\exists$ {v $\in$ $G_s$.$V_s$ | Val(v,name) = Val($v_{gOrig}$, name) and
      Val(v.type) = Val($v_{gOrig}$, type)) then
      $v_{sOrig}$ := find v in $G_s$.$V_s$ where
            Val(v,name) = Val($v_{gOrig}$, name) and
            Val(v,type) = Val($v_{gOrig}$, type)
      insert(($v_{sOrig}$, $v_s$), $G_S$)
      Val(($v_{sOrig}$, $v_s$), exec):= [n]
    End if
  End for
End
```

Fig. 4. SGProv algorithm and procedure UpdateEdges.

Table II. Average processing time for provenance queries.

Queries		**Test base**	**Summary**
Q1. Obtain all graph programs and its descendants.		110ms	15ms

Cypher	START n=node(*)
	MATCH n-[r]->m
	RETURN n.name, m.name
	ORDER BY n.name, m.name;		

Q2. Obtain direct descendants and descendants of descendants (indirect descendants) of the node representing the program whose attribute *"name"*= P5.		93ms	2ms
Cypher	START n=node:node_auto_index(name = 'P5') MATCH n-[r1:INFORMED]->n2-[r2:INFORMED]->n3 RETURN n.name, n2.name, n3.name;		
Q3. Obtain shortest path between nodes that represent programs whose attribute *"name"* = P1 and whose attribute *"name"* = P4 and search for dependencies between programs whose attribute *"name"* = P4 and whose attribute *"name"* = P5.		-	1ms

Cypher	START n=node:node_auto_index(name = 'P1'),
	n2=node:node_auto_index(name = 'P4'),
	n3= node:node_auto_index(name = 'P5')
	MATCH p = shortestPath(n-[:INFORMED*..50]->n2),
	p2 = n2-[r : INFORMED]->n3
	RETURN length(p), n.name, n2.name, n3.name, r		

Q4. Obtain all programs that generated input data to programs whose attribute *"name"* = P10 (ascendant query).		35ms	1ms
Cypher	START n=node:node_auto_index(name = 'P10') MATCH n2-[r]->n RETURN n2.name, n.name, r;		

Query Q2 is the most generic provenance query and is considered a baseline query that all provenance query systems should support [Woodman et al. 2011]. Q2 returns all the direct and indirect descendants of the program whose attribute *"name"* = P5. This query obtained a much lower processing time in the summary graph than in the test base. By running Q2 on the summary graph, it would also be possible to return to which workflow executions P5 edges and its descendants belong, as in the query Q1. In order to do this, it would suffice to include "r1.exec" and "r2.exec" in the RETURN clause.

Query Q3 gets the shortest path between programs P1 and P4. It then seeks for paths between them. The query returns the path length between programs P1 and P4, the programs and edges present in the path and the paths between P4 and P5. In the summary graph, this query ran in a very short processing time. On the other hand, when running on the test base, it timed out. This can be explained by the large volume of graphs on the test base and the large number of nodes representing programs P1, P4 and P5. One way to minimize this problem would be to individually query each graph in the test base. However, the processing time of each query was on average 5,000 ms, in other words, more than 1 hour to query all provenance graphs. On the other hand, the processing time running on summary graph was 1 ms.

Query Q4 is used to explain the presence of output data produced by a program. Q4 returns all programs that have directly or indirectly contributed to generate the output data. This query is especially useful when an error is found in the output data and scientists wish to seek the probable cause [Woodman et al. 2011]. Q4 running on the summary graph resulted in much less processing time than on the test base. Query Q4 could also be run on the summary graph and return to which workflow executions each edge of the result belongs. For this, it would suffice to include "r.exec" in the RETURN clause.

The reduction in processing time obtained with queries acting on the summary graph shows the potential of SGProv. Q1 processing time was reduced by approximately 87%, Q2 and Q4 approximately 97%, and Q3 cannot be answered running on the test base. Thus, increasing the number of graphs in a test base, query processing time must also increase. As the summary graph represents all provenance graphs in test base, eliminating redundant data, queries can be answered based only on it. In the experiment, the generation time of the summary graph was 58 seconds. Even though it may take longer to obtain the summary than executing some of the queries using the whole test base, in some cases, like Q3, it would easily payoff. In addition, summary is generated only once and queried many times, since provenance data does not suffer from updates.

According to Freire et al. [2008], in scientific experiments, provenance helps to interpret and understand the results: by examining the sequence of steps that led to a result, the scientist may have the perception of the chain of reasoning used in its production, verify that the experiment was performed according to accepted procedures, identify entries in the experiment and, in some cases, to reproduce the results. Therefore, powerful analytical provenance queries are critical along the exploratory nature of developing science.

6. CONCLUSIONS

The SGProv summarization mechanism considerably reduced the volume of test base graphs in the example analyzed. In the example, the more programs and their dependencies with each other are repeated in the provenance graphs, the better is the result obtained with summarization. The extra cost corresponds to the introduction of the "*"exec"*" attribute, which will be assessed in future work. Metrics to evaluate the quality of the summary and proof of correctness are under development. One way to evaluate quality is to analyze the participation rate of origin and destination nodes of super-nodes in each super-edge. Participation rates greater than 50% indicate a strong super-edge, otherwise it is considered a weak one. The ideal summary should have the largest possible number of strong super-edges [Tian and Patel 2010]. One way to verify correctness would be to reconstruct the original graph from the summary and analyze the results of provenance queries running on the test base and on the summary graph to identify if they return the same values without losses or redundancies.

The results obtained in the experimental tests showed a significant reduction in query processing time when they are executed on the summary graph. SGProv is of limited value when the base graphs are very different from each other, regarding programs and their dependencies. In this scenario, the summarization mechanism would produce a summary graph as massive as the test base, compromising query processing time. A way to overcome this could be to make the summarization based on other characteristics of nodes and edges of the provenance graphs. One approach might be to group nodes that have the same set of neighboring nodes [Liu and Yu 2011]. However, the main characteristic of provenance graphs from the same experiment is the great similarity between them.

The summarization mechanism considered repeated programs for grouping nodes and edges. Provenance graphs in the PROV-DM format, besides programs (activities), have other types of nodes, such as agents and entities (data), and four types of edges that need to be evaluated. Furthermore, the programs may have several attributes with different values. Currently, queries submitted to SGProv that do not use programs in their specifications would not benefit from summaries and would be submitted to the whole database. However, data entities also present opportunities for summarization. A workflow may be executed several times with the same input data, changing other data like a filter value, a similarity metric or a time step. Entity summarization is an on-going effort within SGProv.

REFERENCES

Abramson, D., Bethwaite, B., Enticott, C., Garic, S., and Peachey, T. Parameter exploration in science and engineering using many-task computing. *IEEE Transactions on Parallel and Distributed Systems* 22 (6): 960–973, 2011.

Aggarwal, C. C. and Wang, H. *Managing and Mining Graph Data.* Springer, 2010.

Anand, M. K., Bowers, S., and Ludascher, B. Provenance browser: displaying and querying scientific workflow provenance graphs. In *Proceedings of the IEEE International Conference on Data Engineering (ICDE).* Long Beach, USA, pp. 1201–1204, 2010.

Anand, M. K., Bowers, S., and Ludascher, B. Database support for exploring scientific workflow provenance graphs. In *Proceedings of the International Conference on Scientific and Statistical Database Management (SSDBM).* Crete, Greece, pp. 343–360, 2012.

Angles, R. and Gutierrez, C. Survey of graph database models. *ACM Computing Surveys* 40 (1): 1–39, 2008.

Davidson, S. B. and Freire, J. Provenance and scientific workflows: Challenges and opportunities. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* New York, USA, pp. 1345–1350, 2008.

Deelman, E., Gannon, D. B., Shields, M., and Taylor, I. Workflows and e-science: an overview of workflow system features and capabilities. *Future Generation Computer Systems* 25 (5): 528–540, 2009.

Freire, J., Koop, D., Santos, E., and Silva, C. T. Provenance for computational tasks: a survey. *Computing in Science and Engineering* 10 (3): 11–21, 2008.

Gadelha, L., Wilde, M. M. M., and Foster, I. Provenance query patterns for many-task scientific computing. In *Proceedings of the Usenix Workshop on the Theory and Practice of Provenance (TAPP).* Greece, 2011.

Gil, Y., Deelman, E., Ellisman, M., Fahringer, T., Fox, G., Gannon, D., Goble, C., Livny, M., Moreau, L., and Myers, J. Examining the challenges of scientific workflows. *Journal Computer* 40 (12): 24–32, 2007.

Guerra, G., Rochinha, F. A., Elias, R., de Oliveira, D., Ogasawara, E., Dias, J., Mattoso, M., and Coutinho, A. L. G. A. Uncertainty quantification in computational predictive models for fluid dynamics using workflow management engine. *International Journal for Uncertainty Quantification* 2 (1): 53–71, 2012.

Haichuan, S. and Kitsuregawa, M. Efficient breadth-first search on large graphs with skewed degree distributions. In *Proceedings of the International Conference on Extending Database Technology (EDBT).* Italy, pp. 311–322, 2013.

Huahai, H. and Singh, A. K. Graphs-at-a-time: Query language and access methods for graph databases. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* New York, USA, pp. 405–418, 2008.

Liu, Z. and Yu, J. X. On summarizing graph homogeneously. In *Proceedings of the Database Systems for Advanced Applications (DASFAA).* Hong Kong, China, pp. 299–310, 2011.

Mattoso, M., Werner, C., Travassos, G. H., Braganholo, V., Ogasawara, E., Oliveira, D., Cruz, S., Martinho, W., and Murta, L. Towards supporting large scale in silico experiments life cycle. *International Journal of Business Process Integration and Management* 5 (1): 79–92, 2010.

Moreau, L. and Missier, P. The prov data model and abstract syntax notation. http://www.w3.org/tr/prov-dm, 2011.

Navlakha, S., Rastogi, R., and Shrivastava, N. Graph summarization with bounded error. In *Proceedings of the ACM International Conference on Management of Data (SIGMOD).* Vancouver, Canada, pp. 419–432, 2008.

Ocaña, K. A. C., Oliveira, D., Dias, J., Ogasawara, E., and Mattoso, M. Optimizing phylogenetic analysis using scihmm cloud-based scientific workflow. In *Proceedings of the IEEE International Conference on e-Science.* Washington, USA, pp. 62–69, 2011a.

Ocaña, K. A. C., Oliveira, D., Ogasawara, E., Dávila, A. M. R., Lima, A. A. B., and Mattoso, M. SciPhy: A Cloud-Based Workflow for Phylogenetic Analysis of Drug Targets in Protozoan Genomes. In O. Norberto de Souza, G. Telles, and M. Palakal (Eds.), *Advances in Bioinformatics and Computational Biology.* Lecture Notes in Computer Science, vol. 6832. Springer, pp. 66–70, 2011b.

Ogasawara, E., Dias, J., Oliveira, D., Porto, F., Valduriez, P., and Mattoso, M. An algebraic approach for data-centric scientific workflows. In *Proceedings of the VLDB Endowment.* Chicago, USA, pp. 1328–1339, 2011.

Robinson, I., Webber, J., and Eifrem, E. *Graph Databases.* O'Reilly Media, 2013.

Sadalage, P. J. and Fowler, M. *NoSQL Distilled: a brief guide to the emerging world of polyglot persistence.* Addison-Wesley Professional, 2012.

Santos, I., Dias, J., Oliveira, D., Ogasawara, E., Ocaña, K. A. C., and Mattoso, M. Runtime dynamic structural changes of scientific workflows in clouds. In *Proceedings of the IEEE/ACM International Conference on Utility and Cloud Computing.* Washington, USA, pp. 417–422, 2013.

Taylor, I. J., Deelman, E., Gannon, D. B., and Shields, M. *Workflows for e-Science.* Springer, 2007.

Tian, Y. and Patel, J. M. Interactive Graph Summarization. In P. S. Yu, J. Han, and C. Faloutsos (Eds.), *Link Mining: models, algorithms and applications.* Springer, New York, USA, pp. 389–409, 2010.

Woodman, S., Hiden, H., Watson, P., and Missier, P. Achieving reproducibility by combining provenance with service and workflow versioning. In *Proceedings of the Workshop on Workflows in Support of Large-scale Science.* New York, USA, pp. 127–136, 2011.