Efficient Lineage Tracking For Scientific Workflows

Thomas Heinis, Gustavo Alonso
Systems Group
Department of Computer Science, ETH Zurich
8092 Zurich, Switzerland
{heinist, alonso}@inf.ethz.ch

ABSTRACT

Data lineage and data provenance are key to the management of scientific data. Not knowing the exact provenance and processing pipeline used to produce a derived data set often renders the data set useless from a scientific point of view. On the positive side, capturing provenance information is facilitated by the widespread use of workflow tools for processing scientific data. The workflow process describes all the steps involved in producing a given data set and, hence, captures its lineage. On the negative side, efficiently storing and querying workflow based data lineage is not trivial. All existing solutions use recursive queries and even recursive tables to represent the workflows. Such solutions do not scale and are rather inefficient. In this paper we propose an alternative approach to storing lineage information captured as a workflow process. We use a space and query efficient interval representation for dependency graphs and show how to transform arbitrary workflow processes into graphs that can be stored using such representation. We also characterize the problem in terms of its overall complexity and provide a comprehensive performance evaluation of the approach.

1. INTRODUCTION

Data lineage and data provenance have been identified as a major problem in the management of scientific data. The problem has become more acute as scientists increasingly use computational means to produce derived data sets [6, 7, 8, 10, 22, 29, 39].

Without lineage information, a data set is often useless from a scientific point of view. The question is then how to capture the lineage information of a data set, how to store it efficiently, and how to allow queries over it. The first part of the problem, capturing the lineage information, has been made substantially easier by the widespread use of workflow tools to describe scientific computations [3, 24, 25, 34]. The workflow process describes what steps were used to produce a particular data set and, hence, can be used to trace the lineage of a data set. Unfortunately, there are no efficient ways to store and query workflow based lineage information. Existing proposals, e.g., Trio [1] and GridDB [19] use recursive queries to retrieve the lineage of a data set. Such an approach does neither scale for large workflow processes nor for large collections of data sets.

From our experience working with scientists in biology [2] and astrophysics [32, 33], it is clear that obtaining the basic lineage is not enough. Scientists are interested in answering queries such as "What algorithms were used to derive this data set?", "Which data sets have been produced with this algorithm?", "What data sets have been derived from this data set?", and so on. While these queries are related to the basic lineage information, being able to answer all of them efficiently requires to have an efficient way to store and query the provenance of every data set.

In this paper we show that existing approaches to storing lineage information do not scale. We also show that workflows with a tree structure produce lineage dependencies that can be very efficiently stored and queried using interval encoding [17]. We then analyze the problem of encoding general workflow graphs. We first characterize the problem and show that the number of dimensions for the encoding depends on the structure of the graph. This makes it impossible to use a single encoding for arbitrary graphs. However, we need to use a single encoding to be able to store the information in a relational database. Thus, we then proceed to explore the problem of transforming arbitrary workflow graphs into tree-like graphs amenable to interval encoding.

In the paper we provide the transformation algorithms, discuss how to optimize the transformation procedure, and present an extensive collection of experiments that evaluate our approach using random graphs and a set of representative scientific workflows. The experiments show that our approach is more efficient than existing techniques.

The remainder of this paper is organized as follows. First, we study the problem of storing the transitive closure of directed acyclic graphs in relational DBMSs (Section 2). We then show that current solutions do not scale well for large graphs and propose to use interval encoding instead. We then show the limited applicability of this approach for arbitrary DAGs (Section 3) and discuss how to transform arbitrary graphs into interval encodable graphs (Section 4). We evaluate the approach (Sections 5, 6, 7), present related work (Section 8) and conclude with a discussion of the results presented (Section 9).

SIGMOD'08, June 9–12, 2008, Vancouver, BC, Canada.

2. WORKFLOW BASED DATA LINEAGE

2.1 Workflow Model

Workflows are widely used in scientific applications [4, 9, 16, 20, 27, 28, 30]. Workflows orchestrate the execution of tasks by means of a graph that defines the control and data flow between those tasks. A workflow takes a collection of data sets as input and produces a collection of data sets as output. The tasks also take data sets as input and produce data sets as output. The dependencies between tasks and data sets are determined by the control and data flow described as part of the workflow process. Therefore, the lineage of a data set can be determined by tracing back how it was produced using the corresponding workflow.

For the purposes of this paper we treat workflows as directed acyclic graphs (DAGs). Although some workflow tools allow cycles, any execution of a workflow process can be represented as a DAG by unrolling the loops. In this paper we assume that this is always the case. The nodes of the DAG represent tasks and data sets. The edges represent dependencies between them. A directed edge will point from a task to a data set if it the data set is an output of the task, and from a data set to a task if the data set is used as input to the task. This completely captures all dependencies between tasks and data sets, and between data sets and data sets.

The lineage information induced by the DAG is the transitive closure of all dependencies. Hence, from here on we treat the problem of storing and querying the lineage information as the problem of storing and querying the transitive closure of the dependency graph.

For the experimental setup used in the remainder of this section please refer to Section 5. Deviations from this setup are explained in the text.

2.2 Lineage Using Recursive Queries

Systems like Trio and GridDB use recursive queries to retrieve lineage information. The dependencies between data sets are stored using a relation with two attributes of the form Dependency(parent_id, child_id). To find the lineage of a data set, the query recursively asks for the parents of the data set, the parents of the parents, and so on.

We have assessed the performance of this approach by measuring the time it takes to retrieve the lineage of a leaf node of a randomly generated DAG (between 5 and 100 nodes and random edges). The experiments were carried out using a Postgres DBMS and, because of the lack for support for recursive queries, stored procedures querying the relations recursively were used. The results are shown in Figure 1 (a), where the time to obtain the lineage information is plotted against the the number of nodes in the dependency graph.

The time it takes to execute recursive queries is linear with the number of paths in the graph. For small graphs, with less than 30 nodes, the response time grows slowly. Beyond 30 nodes, however, the query time becomes unpredictable, growing exponentially high - in some cases up to 3 seconds per query. This behavior, combined with the linear growth in query time with the number of paths in the graph, makes recursive queries unsuitable for exploring the lineage of large scientific workflows.

2.3 Lineage Using All Paths

Recursive queries minimize the amount of information to store at the cost of longer query running times. The other extreme would be to trade space for speed and store all paths in the DAG so that the query only needs to retrieve the corresponding paths. Finding all paths in a directed acyclic graph amounts to topologically sorting the graph. This can be done in linear time or, more precisely, for a graph $G = (V, E)$, in $O(|V| + |E|)$. A topological sort of the graph will yield all total orders, which is equivalent to all paths P through the graph. This can be done offline and needs to be done only once for every workflow, so the cost is amortized over time. An efficient way of storing all paths is based on the observation that since the path p_i is a total order, each element/data set $e \in p_i$ for $p_i \in P$ can be assigned an integer denoting the position o on p_i. The triples e, i and o are stored in a relation Paths(path_id, node_id, order_no). The query to retrieve the lineage of a data set given the data set ID nid is as follows:

```
SELECT pt2.node_id
FROM Paths pt1, Paths pt2
WHERE pt1.node_id = nid AND
      pt1.path_id = pt2.path_id AND
      pt1.order_no > pt2.order_no
```

The query selects all the paths that contain a given node n, retrieves all the nodes found on those paths and filters over the order, such that only elements occurring before n on those paths are returned.

We evaluate the performance of this approach using a collection of random DAGs. The results are shown in Figure 1 (b) where the time to obtain the lineage information is plotted against the number of nodes in the dependency graph. As the figure shows, the path approach scales better than recursive queries. However, in Figure 1 (c) we show the number of tuples required to store all paths against the number of nodes in the graph. Storing all paths may lead to a large storage overhead. If several thousand workflow executions need to be stored, these large numbers can become problematic by degrading performance.

2.4 Lineage Over Interval Tree Encoding

The results of these initial experiments indicate that the two techniques examined represent two extremes. Recursive queries require little space but can be very slow. Storing all paths leads to faster queries but the storage requirements grow too large. Clearly, an alternative is needed. Encoding the transitive closure of a tree to store it in a database and retrieve it efficiently has already been used in several applications [12, 23, 35] with the basic idea stemming from [17]. The approach uses one-dimensional intervals over the natural numbers to represent nodes in the tree. If a node n_1 is a predecessor of another node n_2, the interval representing n_1 must enclose the interval representing n_2. More formally, a node n_i is represented as an interval (l_i, r_i). Then:

- n_1 is a predecessor of $n_2 \Leftrightarrow l_1 < l_2$ and $r_1 > r_2$;
- n_2 is a predecessor of $n_1 \Leftrightarrow l_2 < l_1$ and $r_2 > r_1$;
- n_1 and n_2 are unrelated $\Leftrightarrow (l_1 > l_2 \wedge r_1 > r_2)$ or $(l_1 < l_2 \wedge r_1 < r_2)$.

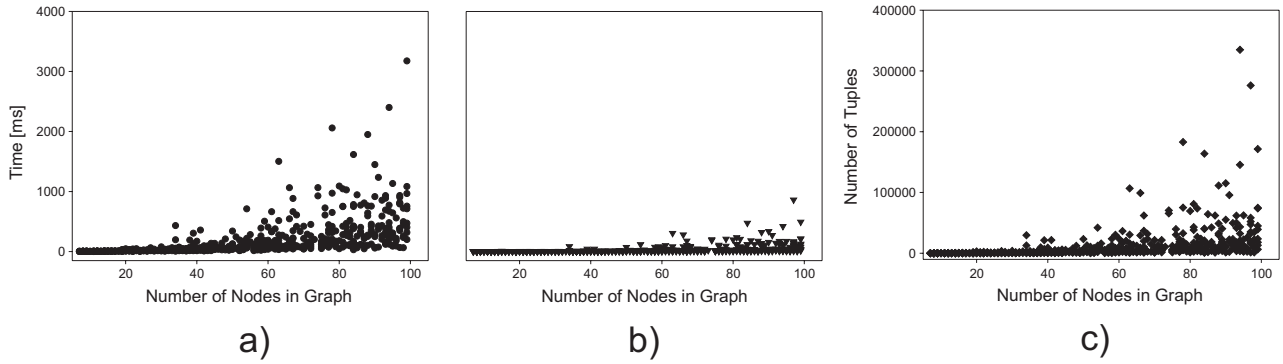

Figure 1: Response time for (a) recursive queries and (b) queries over all paths for random graphs. Storage space required for the all paths approach (c).

All successors of a node can be determined by finding all the intervals that include the interval of the node. Similarly, all predecessors (e.g., the lineage) can be determined by finding all the intervals that enclose the interval of this node. We store this information in a relation of the form TC(node_id, left, right). The query for determining the lineage of a node with node_id *nid* then is:

```
SELECT tc2.node_id
FROM TC AS tc1, TC AS tc2
WHERE tc1.node_id = nid AND
      tc2.left < tc1.left AND
      tc1.right < tc2.right
```

Unfortunately, tree encoding cannot be used on arbitrary DAGs. Hence we cannot compare it directly with the other two techniques. We have nevertheless performed an experiment on randomly generated trees with between 5 and 100 nodes. The results of finding the lineage of a leaf in the tree are shown in Figure 2. The result, compared with the previous results, is that the running time is very stable independently of the size of the tree and the overhead is actually very low compared to the times for recursive queries or queries over all paths. Such is the behavior that we aim to achieve when encoding arbitrary DAGs.

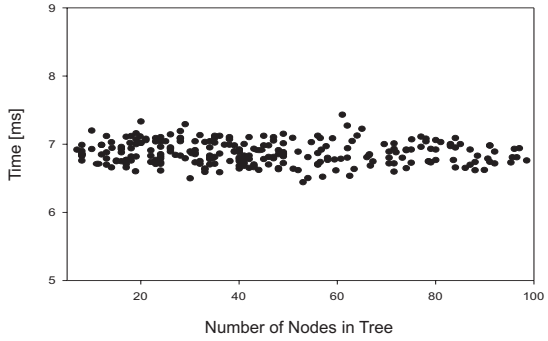

Figure 2: Response time for queries for a tree stored with the directed tree encoding.

3. ENCODING DAGS WITH INTERVALS

3.1 Overview

Arbitrary DAGs cannot always be encoded using one-dimensional intervals. This can be easily illustrated with the example DAG depicted in Figure 3. The intervals representing nodes A, B, and C must have overlapping regions because they have common successors. No matter how the intervals are arranged, one of the intervals for D, E, or F will end up having a predecessor that does not exist in the real graph.

This can be formally proven, however, here we just outline the proof. For the intervals A, B, and C to overlap, but not enclose each other, there is one intersection between two of these intervals that will always be completely enclosed by the third interval. Hence, intervals in that intersection will be successors of the first two intervals but also of the third. There is no possibility to have successors of the two intervals that are not also successors of the third.

The graph of Figure 3 can be encoded by using two-dimensional intervals (rectangles in the plane) instead of one-dimensional intervals. However, if we use rectangles in the plane, there are instances where the same situation arises in two dimensions. Given any number of dimensions, one can always come up with a graph that needs more dimensions to be encoded. In what follows, we explore the problem more formally.

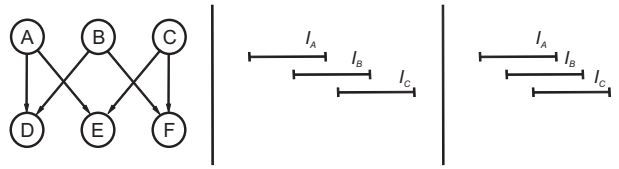

Figure 3: The graph (left) can not be encoded with intervals: with the only two interval assignments in which I_A&I_B and I_B&I_C can have common successors, I_A&I_C can not have common successors (middle) and all common successors of I_A&I_C must also be successor of I_B (right).

3.2 Graph Encoding

The problem of encoding dependency graphs has been extensively studied in the literature.

The set of directed acyclic graphs that can be encoded with intervals is called the class of *interval containment graphs* [18]. More generally, the family of graphs that can be encoded with d-dimensional objects in Euclidean space is called *containment graphs*.

The number of dimensions needed to encode a graph can be determined from the structure of the graph. A directed acyclic graph $G = (V, E)$ represents a partially ordered set (*poset*) P of the nodes V of G: $P = (V, <)$. In the poset P, $i, j \in V$ are comparable ($i < j$) if $[i, j] \in E$ or $[j, i] \in E$. In other words, two nodes i and j are comparable if G contains a directed path from i to j or from j to i. Otherwise, if neither $[i, j] \notin E$ nor $[j, i] \notin E$, i.e., no path linking them exists in G, then i and j are incomparable. The dimension of a poset is defined as the minimum size k of the realizer of P, where the realizer is a collection of linear orders $L_1 = (V, <_1), ..., L_k = (V, <_k)$ such that $x < y$ if $x <_i y \ \forall i$, or $P = \bigcap_{i=1}^{k} L_i$ [13]. The number of dimensions necessary to encode the graph is a function of the dimension of the poset of the graph [15] as follows:

- if the dimension of the poset is at most 2, then the dimension d of the objects required to represent G is 1, meaning that G can be represented by intervals on the line (i.e., one-dimensional intervals) [14].
- if the dimension of the poset represented by G is at most $2d$, then the objects required to represent G are boxes in d-dimensional Euclidean space [15].

From here it follows that there is no encoding with a fixed number of dimensions that can perfectly encode arbitrary DAGs. For trees, the poset has dimension 2 [40]. This is why a tree can be encoded with intervals in one-dimensional space.

3.3 Related Complexity Results

Testing if the dimension of a poset is no bigger than 2 can be solved in polynomial time [41]. The problem of determining the dimension of a poset bigger than 2, however, is NP-complete [15].

Testing whether the dimension of the poset is 2 can be done by testing whether the incomparability graph $I_G = (V, E_{IG})$ of $G = (V, E)$ is transitively orientable [41]. The incomparability graph I_G can be derived from G by adding to E_{IG} an undirected edge if $x, y \in V$ are not comparable, i.e., if x, y do not have any predecessor/successor relationship. Testing whether a graph is transitively orientable can be done in polynomial time [26] or even in linear time [21].

Determining if a given graph is an interval containment graph, and can therefore be encoded with intervals on the line, can also be determined by finding forbidden subgraphs. If a graph contains any induced subgraph of a known set (defined in [31, 36]) then it is not an interval containment graph. The set of forbidden subgraphs is limited to a small number of graphs. One strategy for transforming an arbitrary graph into a graph with a poset of dimension 2 is to detect any forbidden subgraphs it might contain and transform the forbidden subgraphs into subgraphs of poset dimension 2 while maintaining the transitive closure. Unfortunately, detecting forbidden subgraphs is related to the problem of subgraph isomorphism which is NP-complete [37].

4. TRANSFORMATION ALGORITHM

Our goal is to come up with a transformation algorithm that takes an arbitrary DAG as input and outputs an equivalent DAG (with the same transitive closure) that can be encoded with one-dimensional intervals. A brute force approach is to take the DAG and transform it into a tree by duplicating nodes. The result is an optimized version of the "all paths" approach (the optimization arises from not having to replicate common subpaths). Yet the behavior is similar to storing all the paths. We can also not just replace the forbidden subgraphs with tree structures because finding them is NP-complete.

What we can do, however, is to determine whether a subgraph has a poset of dimension 2 or higher. Thus, we can take the original DAG, find its incomparability graph, determine the independent subgraphs of the incomparability graph, and check for each one of those subgraphs whether they are transitively orientable (which can be done in polynomial time, see above). If they are, they can be encoded with intervals. If they are not, their corresponding subgraph in the DAG needs to be transformed. Note that this is not the same as finding induced forbidden subgraphs (which is NP-complete as indicated above).

Each of the identified problematic subgraphs of the DAG is transformed into a tree through node duplication. Optimizations are applied to minimize the space overhead generated by node duplication. The trees are then glued back into the original graph. Once the transformation is complete, we proceed with the interval encoding.

The process is summarized in Algorithm 1. In what follows we describe in detail how the algorithm works by explaining each one of its functions: computing the incomparability graph, finding independent subgraphs, testing for transitive orientability, transformation to a tree and optimizations. We also describe how the encoding works.

Algorithm 1 High-level view of the algorithm used to reduce the dimension of an arbitrary DAG.

Algorithm: Transformation Algorithm

Input: graph: input DAG

```
1 Graph icgraph = computeIncomparabilityGraph(graph)
2 foreach subgraph in independentSubgraphs(icgraph) do
3     if dimension(subgraph) > 2 then
4         optimize(subgraph)
5         transformToTree(subgraph)
6     end
7 end
```

4.1 computeIncomparabilityGraph(*graph*)

The incomparability graph I_G has the same set of nodes as G, $V_{IG} = V$, but has a different set of edges E_{IG}: if two nodes $v_1, v_2 \in V$ have no transitive relation, then the undirected edge $(v_1, v_2) \in E_{IG}$. This means in particular that if v_1 and v_2 do not share a path, i.e., they are not part of the same total order over E (and hence not comparable in the partial order defined by G), then they are connected through an undirected edge in E_{IG}. Figure 4 (left) depicts a graph G and its corresponding incomparability graph I_G (middle).

To compute I_G efficiently, the algorithm uses a $|V|/2 \times |V|/2$ matrix C of booleans. $C[i, j]$ is true if there is a path

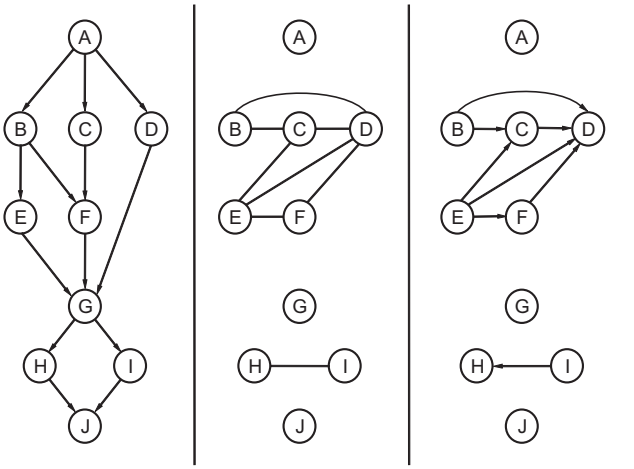

Figure 4: The incomparability graph (middle) for a graph (left) and a possible transitive orientation of the incomparability graph (right).

between v_i and v_j $\forall i < j$. The matrix can be computed by traversing the graph in $O(|V| + |E|)$. Computing I_G from the matrix C only requires one pass over C and adding an undirected edge (v_i, v_j) to E_{IG} if $C[i, j]$ is false. This is done in $O(|V|^2)$.

4.2 independentSubgraphs(*icgraph*)

The graph I_G may contain several subgraphs not connected to each other. Such an example is shown in Figure 4 (middle). These situations arise very often in real workflows. For instance when there are tasks that are in all possible paths (e.g., a start task like task A in Figure 4). The same situation arises at synchronization points for parallel threads (e.g., task G in Figure 4).

How these independent subgraphs arise can be illustrated using the example in Figure 4.

Assume there is a node s contained on all execution paths such that each other node $v \in V$ on each execution path, and total order is either $s < v$ or $v < s$. Consider two sets A and B. B contains all nodes $v \in V$ which satisfy $s < v$ in all total orders over G. A contains all nodes $v \in V$ which satisfy $v < s$ in all total orders over G. Note that $A \cap B = \emptyset$, as all nodes $v \in V$ are either before or after s on all execution paths. All nodes $v \in A$ are on a path through s. Thus in I_G there will not be any edge between any of the nodes in A and s as they are comparable in G. The same applies to B and s. Because every node in A reaches s and s reaches all the nodes in B, there will be no edges between nodes in A and nodes in B. Hence A, B and s result in independent subgraphs of the incomparability graph. This is the situation induced by node G in Figure 4.

Finding such subgraphs in I_G is important as they can be treated independently in determining their dimension and also in the reduction of their dimension as will be shown later. The independent subgraphs of I_G are identified by traversing I_G starting at each source (node with in-degree 0) and visiting all nodes which are connected to the source. This is done by traversing the graph in $O(|V| + |E|)$.

Algorithm 2 Transitively orient an undirected graph.

Algorithm: Orient Transitively

Input: undirected subgraph of $G_{SIG} = (V_{SIG}, E_{SIG})$
Output: $\vec{E_{IG}}$: set of directed edges

```
1  E⃗_IG ← ∅
2  choose arbitrary edge ∈ E_SIG, remove edge from E_SIG
3  direct edge arbitrarily: (u, v), add edge to E⃗_SIG
4  repeat
5      test if any edge incident to edge in E_SIG with (v, w),
       assign to I_v
6      test if any edge incident to edge in E_SIG with (t, u),
       assign to I_u
7      if I_v = I_u = ∅ ∧ E_SIG ≠ ∅ then
8          choose arbitrary edge ∈ E_SIG
9          remove edge from E_SIG
10     end
11     foreach incident ∈ I_v do
12         if (u, w) ∉ E_SIG then
13             direct incident (w, v), add to E⃗_SIG
14             remove incident from E_SIG
15         end
16     end
17     foreach incident ∈ I_u do
18         if (t, v) ∉ E_SIG then
19             direct incident (u, t), add to E⃗_SIG
20             remove incident from E_SIG
21         end
22     end
23 until E_SIG = ∅ ;
```

4.3 dimension(*cluster*)

Each independent subgraph of the incomparability graph I_G is tested separately. If the dimension of the partially ordered set represented by G is no higher than 2, then the incomparability graph I_G is transitively orientable. I_G is transitively orientable if for each edge in E_{IG}, a direction can be assigned such that all directions of the edges satisfy transitivity, i.e., if the directed edges (u, v) and $(v, w) \in \vec{E}_{IG}$ then there must also be a directed edge (u, w) in $\vec{E}_{IG}$.

Clearly, if I_G contains several independent subgraphs, the dimension of each of these subgraphs can be tested separately: as no undirected edge connects any two nodes of two independent subgraphs, each of the subgraphs must be transitively orientable for the entire graph to be transitively orientable. If one subgraph is not transitively orientable, only the dimension of this subgraph needs to be reduced. Then the subgraph becomes transitively orientable and, if this is the case for all subgraphs, the whole graph becomes transitively orientable as well.

We use a polynomial time algorithm to test if a subgraph is transitively orientable [26]. Given an undirected subgraph $G_{SIG} = (V_{SIG}, E_{SIG})$ of I_G, the algorithm initially picks a random undirected edge $e \in E_{SIG}$, removes e from E_{SIG}, directs it arbitrarily and adds the resulting $\vec{e}$ to a set of directed edges called $\vec{E_{SIG}}$. It then tries to direct as many edges as possible by taking an edge $\vec{e}$ out of $\vec{E_{SIG}}$ and directing all undirected edges $e \in E_{SIG}$ incident to the source or destination of $\vec{e}$. The incident edges are directed such that giving each incident edge a direction will not violate the transitivity property. This means for a directed edge

$\vec{e} = (u, v)$, that if an edge f incident to v is directed such that $\vec{f} = (v, w)$ then there must also exist an edge g which can be directed $\vec{g} = (u, w)$ in order to maintain the transitivity property. Otherwise, f must be directed such that $\vec{f} = (w, v)$.

More formally, the edges incident to a directed edge $\vec{e} = (u, v)$ are directed according to the following rule: an undirected edge $d = (t, u)$ incident to the source u of $\vec{e}$, is directed such that $\vec{d} = (u, t)$ and added to $\vec{E_{SIG}}$, if $(t, v) \notin E_{SIG}$. This leaves the two directed edges $\vec{d} = (u, t)$ and $\vec{e} = (u, v)$. If the edges were directed $\vec{d} = (t, u)$ and $\vec{e} = (u, v)$ or $\vec{d} = (u, t)$ and $\vec{e} = (v, u)$ then E_{SIG} would be required to contain an edge $f = (t, v)$ (which would be required to be directed accordingly in order to maintain transitivity). The edges incident to the destination of $\vec{e}$ are directed using the same rule. The newly directed edges are removed from E_{SIG} and added to $\vec{E_{SIG}}$.

The algorithm directs edges according to this rule as long as there are undirected edges in E_{SIG} and as long as there are directed edges in $\vec{E_{SIG}}$ with undirected edges incident to either their source or destination. Once no such edges are left in $\vec{E_{SIG}}$, a random edge is chosen from E_{SIG} and is directed arbitrarily. The algorithm then again continues to direct the edges using the same rule.

The procedure is sketched in Algorithm 2. The algorithm finishes once all edges are directed. What follows is a test of the transitivity of the graph defined by $(V, \vec{E_{SIG}})$. For this, all edges $\vec{e} = (u, v) \in \vec{E_{SIG}}$ are tested iteratively. Particularly, given $\vec{e}$, if there exists a $\vec{f} = (v, w)$, then $\vec{E_{SIG}}$ must also contain a directed edge (u, w). If this is not the case for all $\vec{e} \in \vec{E_{SIG}}$ then $(V, \vec{E_{SIG}})$ is not transitively orientable.

4.4 optimize(*cluster*)

It is possible that within an independent subgraph of the incomparability graph that is not transitively orientable, there are parts that can be encoded. Consider, as an example, the shaded subgraph of the graph in Figure 5 (left). Such a subgraph is a common pattern in workflows that execute sets of parallel tasks [38]. The graph as a whole cannot be encoded, but the shaded subgraph can as it is of dimension 2. There are many workflow patterns that share this property of being of dimension 2 and which therefore do not need to be transformed into a tree (e.g., a sequential chain of tasks). However they might be embedded into larger graphs that cannot be encoded. An obvious optimization is to detect these patterns before proceeding with the transformation of the whole graph into a tree.

Instead of trying to detect patterns, the algorithm identifies subgraphs and tests for their dimension. If the dimension is no higher than 2, then the subgraph does not need to be transformed. Then, in practice, the subgraphs which do not need to be transformed are removed from the graph and replaced with a placeholder node. The graph is then converted and, after the transformation, the removed subgraphs are put back in the graph.

The subgraphs to be identified by the algorithm have one common property: they all contain two designated nodes of which a first node i is the only node in the subgraph which is the destination of edges outside the subgraph and the second node o is the only node in the subgraph which is the source of edges leading outside the subgraph. The subgraph between i and o, S_G is then tested for its dimension. If it is of dimension higher than 2, then the subgraphs in S_G are again tested.

Finding the subgraphs with the property described above is done by testing all possible pairs of nodes and checking to see if they are on the same set of paths in $O(|V|^2)$. This check can be done using the matrix C introduced in Section 4.1.

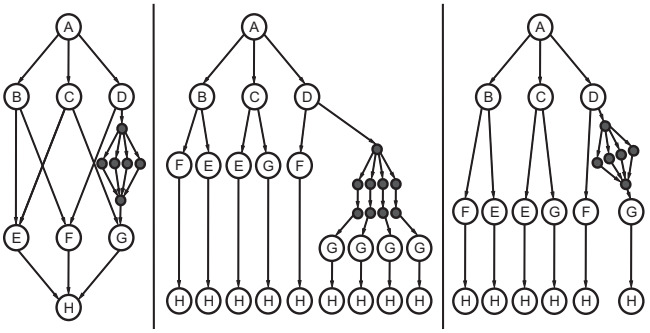

Figure 5: Illustration of the transformation optimization. The initial graph (left), the corresponding transformed tree (middle) and the optimized transformed tree (right).

4.5 transformToTree(*cluster*)

If an incomparability subgraph is not transitively orientable, then the corresponding dependency subgraph needs to be transformed into a tree while maintaining the transitive closure of the dependency subgraph. Algorithm 3 traverses the subgraph and in case a node with an in-degree in greater than 1 is visited, then this node will be copied $in - 1$ times. For each of the copied nodes, also the outgoing edges of the node are also copied. The in incoming edges of v will be reconnected to the $in - 1$ copies and to the node. Reconnecting the incoming edges to the copies ensures that each of the resulting nodes has an in-degree of at most 1, thus ensuring that the resulting graph is indeed a tree. Also, connecting each incoming edge of the node to a copy ensures that the transitive closure is preserved. Figure 6 illustrates how the algorithm works. The algorithm is equivalent to a graph traversal, hence its complexity is $O(|V| + |E|)$.

The tree is inserted into the original graph by first removing all the nodes and edges of the problematic subgraph and then gluing the tree in its place.

4.6 Interval Assignment

Subsequent to the graph transformation, each node in the graph needs to be assigned an interval on the line. The graph, which is going to be encoded, is the result of the transformation algorithm discussed before and hence we can assume that it is encodable with intervals, i.e. it represents a poset of dimension 2.

The interval assignment of such a graph can be determined by using two linear extensions realizing the poset defined by G. In order to compute two linear extensions, the transitive orientation of the incomparability graph I_G of graph G is required. Since G has been modified, the transitive orientation has to be computed again by using the procedures described in Sections 4.1, 4.2 and 4.2.

Algorithm 3 Algorithm used to transform an arbitrary DAG into a tree while maintaining the transitive closure.

Algorithm: Tree Transformation

Input: graph: directed subgraph of $G_S = (V_{SG}, E_{SG})$

```
1  foreach node ∈ V_SG do
2      if in-degree of node > 1 then
3          foreach edge of all incoming edges - 1 of node do
4              copy node
5              set copy as destination of edge
6              foreach edge of all outgoing edges of node do
7                  copy edge
8                  set copy as source of edge
9              end
10         end
11     end
12 end
```

Two linear extensions can then be computed by traversing the graph from sources to sinks, visiting each node only once its predecessors have been visited. If several nodes are ready to be visited, they are by definition incomparable and are visited in the order of the transitive orientation to obtain the first linear extension. For the second extension, the incomparable nodes are visited in the reverse order of the transitive orientation. The intuition behind this is that in one linear order the two incomparable nodes x, y must occur $x < y$ while in the other order they must occur $y < x$. The case in which several nodes are ready to be visited occurs if they have no predecessor/successor relationship. Hence they are incomparable and therefore their order is defined in the transitive orientation of the incomparability graph.

Assume two linear orders, L, $\overline{L}$ realizing the partial order P on the set of vertices V of graph G (the dimension of the poset therefore is 2). L defines an order over V whereas $\overline{L}$ defines an order over a copy of V called $\overline{V}$ in which the elements are renamed, e.g., x becomes $\overline{x}$. The two linear orders are then appended such that $L = L_2 + L_1^{-1}$, where L_1^{-1} is simply the inverese of L_1 such that if $x, y \in V$ with $x < y \in L_1$, then $y < x \in L_1^{-1}$. L then has the following properties:

- if two vertices $x, y \in V$ are comparable and if $x < y$ (in L_2) then $\overline{y} < \overline{x}$ (in L_1^{-1}) and vice versa. Hence in this case in L: $x < y < \overline{y} < \overline{x}$.
- if two vertices $x, y \in V$ are not comparable, then $x < y$ in L_2 and $\overline{x} < \overline{y}$ in L_1^{-1} (because it is $\overline{y} < \overline{x}$ in L_1) and in this case in L: $x < y < \overline{x} < \overline{y}$.

L will then allow us to easily label all the nodes in the graph. If two nodes x, y have a predecessor/successor relationship, they are comparable and in case they have no such relationship they are not comparable. Assume each element in L is labeled increasing from left to right, each node x will be assigned the interval $I_x = [x, \overline{x}]$. Then if two elements x, y are comparable, they have a predecessor/successor relationship and then, as discussed previously, L: $x < y < \overline{y} < \overline{x}$. From the assignment of intervals follows that I_x will enclose I_y. If x, y are not comparable, they have no predecessor/successor relationship and, L: $x < y < \overline{x} < \overline{y}$. I_x and I_y will therefore overlap. The semantic of I_x enclosing I_y depends on the relation $<$ used in the poset. If $x < y$ implies x is a predecessor of y, then I_x will enclose I_y and vice versa.

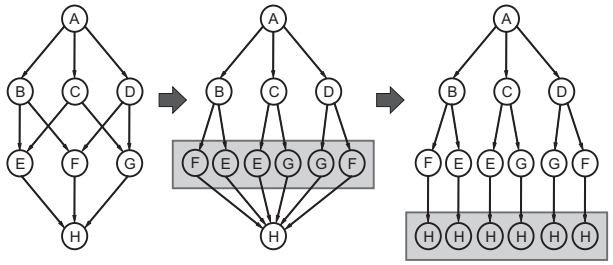

Figure 6: Illustration of the transformation of a graph into a tree.

The complexity of computing the transitively oriented incomparability graph has been derived in Sections 4.1, 4.2 and 4.2. After transitively orienting G, all that is left to be done in order to assign an interval for each node, is to traverse the graph. The complexity of this step is $O(|V|+|E|)$.

5. EVALUATION SETUP

To demonstrate the feasibility and competitive advantage of our approach, we have carried out experiments for two classes of graphs, the first being random DAGs and the second being a set of representative and interesting use cases of scientific workflows.

For the evaluation of our approach we have performed experiments to measure the overhead of the graph transformation and interval assignment. Additionally, we have also measured the response time for a lineage query (the lineage of a randomly selected leaf node) as well as the number of tuples to be stored for a graph. We compare the numbers obtained with the two other approaches, querying over all paths and recursive queries.

The setup used was the same throughout all experiments: two nodes in a cluster of Linux machines (dual Opteron 2.4 GHz machines with 2 GB memory) connected with a 1Gb/s local area network were used. One node hosted a Postgres 8.2.1 database where the graph information was stored while the other node hosted the client. The client performed the calculations, the transformation of the graph as well as the interval assignment, and issued all queries.

Random DAGs are not very realistic representations of scientific workflows. Many optimizations we have proposed do not apply, even though they are very common in real workflows. Yet random graphs can be seen as a worst case scenario for comparison purposes.

6. EVALUATION ON RANDOM DAGS

In a first set of experiments we generated random DAGs and, if necessary, applied the transformation to them. The resulting graph was assigned intervals for each node and was stored in the database. Each random DAG was assigned a random number of nodes between 5 and 100 and each node was assigned between 1 and 4 outgoing edges to other randomly chosen nodes. The edges were assigned such that the addition of an outgoing edge did not lead to a cycle in the graph.

6.1 Preprocessing

In a first experiment we measured the overhead of transforming the graph. For the all paths approach, this involves computing all paths in the graph, whereas, in the case of the interval encoding, this includes the graph transformation and the interval assignment. The recursive queries approach does not require any preprocessing: the relationships or edges are directly stored in the database. Thus, we exclude it from this comparison.

The time required to preprocess the graph in both approaches does not directly depend on the number of nodes in the graph but rather on the structure of the graph. It can, however, generally be assumed that the more nodes a graph has, the more complex its structure is going to be (this is particularly true for random DAGs). Figure 7 shows that the time required for the preprocessing is generally higher the more nodes a graph has. In general, the storing all paths approach performs better than the interval encoding approach. This is not surprising, given that the interval encoding approach also needs to compute all paths in the graph. Note, however, that the preprocessing is done offline and only once, hence the overhead is acceptable.

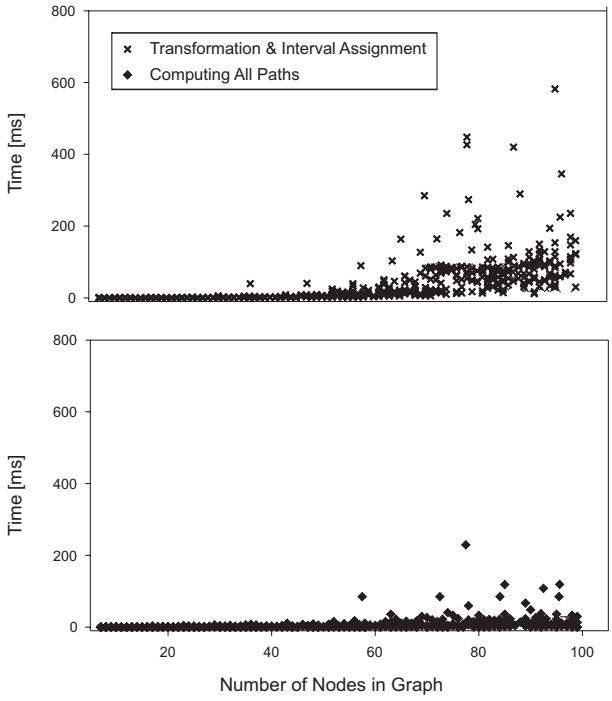

Figure 7: Time required to preprocess a random DAG.

6.2 Querying

Figure 8 shows a comparison of the three approaches regarding response time. For each DAG, a query asking for all predecessors of a leaf (e.g., the lineage of the node) was issued.

Figure 8 (a) clearly indicates that the recursive approach does not scale very well with graphs having an increasing number of nodes. While in some cases the response time is in line with the other two approaches, there are many cases where the running time is much higher. Figure 8 (b) & (c) illustrate the difference between our approach and storing all paths. In the majority of cases, the interval encoding approach clearly outperforms the storing all paths approach.

6.3 Storage Size

It is also important to measure the storage space required by each approach. In this experiment we have therefore compared how many tuples are required to store the transitive closure information of the DAGs.

The number of tuples required to store the transitive closure in case of the recursive approach is equal to the number of edges in the graph. Clearly, no other approach will require less tuples. In Figure 9 we compare the amount of storage required by all paths and that required by interval encoding.

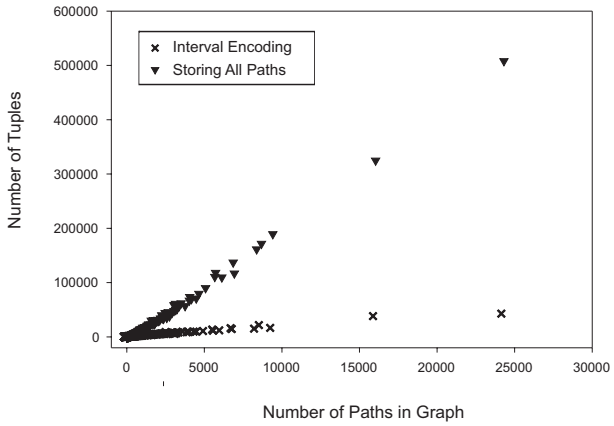

Figure 9: Storage space requirements.

As was to be expected, storing all paths in the graph requires the largest number of tuples. The number of tuples required roughly scales linear with the number of paths in the graph. Our approach with the optimized transformation into an encodable graph requires significantly less tuples to store the transitive closure.

7. EVALUATION ON SCIENTIFIC WORKFLOWS

In the next series of experiments we compare the different approaches using a set of real scientific workflows: fMRI, Montage and EMAN (Figure 10). These three workflows have interesting structures that make them particularly suitable for our experiments. The fMRI workflow serves as an example of a small static acyclic workflow which makes use of parallelism to speed up the computation. The EMAN workflow iterates over a loop to improve the overall quality of the result. By doing so, it leads to dependency graphs of very large depth. The Montage workflow on the other hand does not have the depth of the EMAN workflow, but it is massively parallel.

7.1 Functional Magnetic Resonance Imaging (fMRI)

The functional magnetic resonance imaging workflow (fMRI) [16] is used to process raw data of brain scans. It takes in

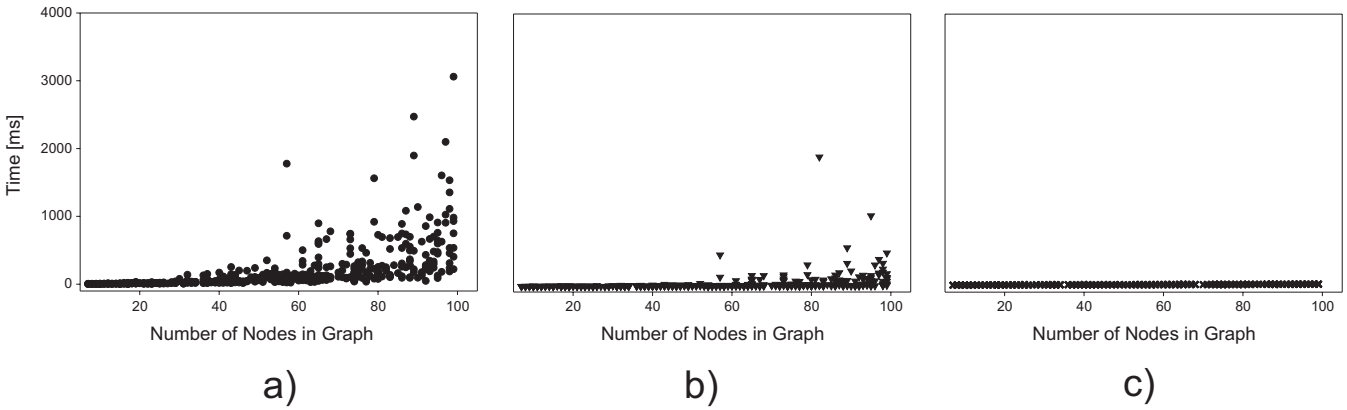

Figure 8: Response time over random DAGs the three approaches, (a) recursive, (b) all paths and (c) interval assignment.

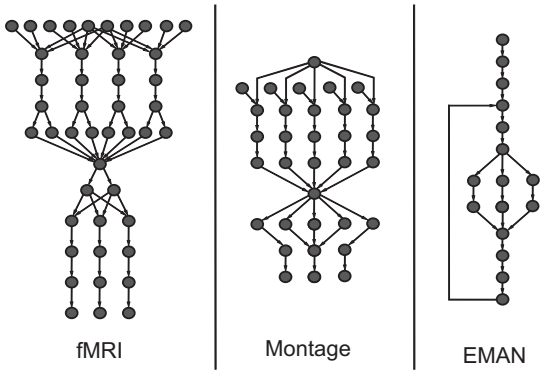

Figure 10: Dependency graphs for three scientific workflows: fMRI, Montage and EMAN (left to right).

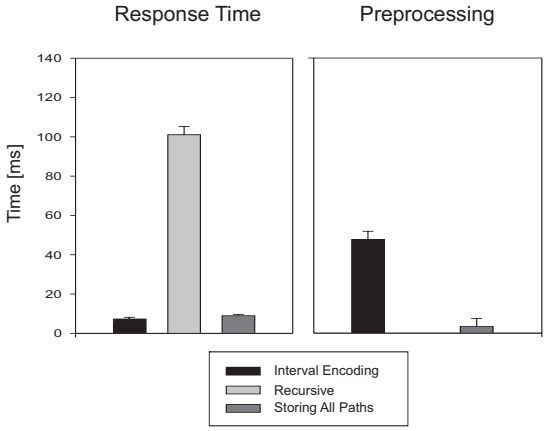

Figure 11: Comparison of the response time for queries and time for the calculation for fMRI.

a first step the raw data, aligns it to a reference brain image by reslicing it, averages over several scans executed with different wavelengths, and finally slices along the x, y and z dimensions. The structure is reasonably simple, with two phases of parallel program executions, the first phase ending at the averaging over all scans (where all execution paths meet) and the second starting thereafter.

Although being the smallest workflow we use in the experiments, it is not a simple workflow in terms of structure as it has to be transformed (the first phase of the parallel program executions). Thus, from the original size of 45 nodes, it is transformed into a graph of 107 nodes. The increase, however, is still modest in comparison, as the approach storing all paths would require 2496 tuples.

Figure 11 compares the response time (left) for a lineage query over the fMRI workflow encoded using the three approaches. The interval encoding approach clearly outperforms the others, especially the recursive approach. As it is shown in Figure 11 (right), the storing all paths approach performs reasonably well regarding response time, but requires twice the number of tuples. The preprocessing takes 46.27ms for interval encoding, 9.7ms for the storing all paths approach and no time for the recursive approach.

7.2 Astronomical Image Mosaic (Montage)

The Montage workflow [4] was developed as part of the National Virtual Observatory (NVO), aiming at processing raw instrument data or images from telescopes and assembling them into a mosaic out of many (possibly hundreds) pieces of data. The workflow runs through several stages, starting by transferring the input files in parallel, reprojecting and fitting them into a common plane. The execution paths of the workflow then converge in the background modeling node, subsequently go through the background correction and finally through the assembly of the image stages in parallel. The degree of parallelism of the phases before and after the background modeling depends on the number of input files to be processed. Smaller instances of the workflow typically assemble more than 40 input files, resulting in 40 parallel execution threads leading to a dependency graph of more than 1000 nodes.

Figure 12 (top) shows the space required by the different approaches to store the transitive closure of the montage workflow depending on the number of input files. Since this workflow does not need transformation, both the recursive approach and the interval encoding do not require

much space. The all paths approach requires considerably more space and the space needed grows significantly with the number of input files, making it unsuitable for this type of workflow.

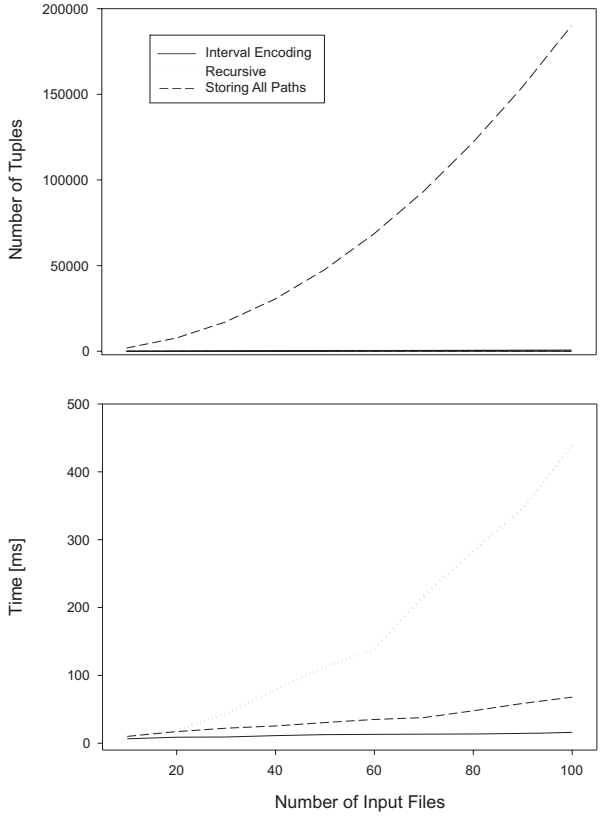

Figure 12: Space requirements for the Montage workflow (top) and response time (bottom).

Figure 12 (bottom) shows the response time as a function of the number of input files processed. While the recursive approach grows almost exponentially, the other two approaches grow linearly, interval encoding being clearly better.

7.3 Electron Micrograph Analysis (EMAN)

The EMAN workflow [20, 30] processes thousands of micrographs from electron microscopes, iteratively trying to determine a macromolecular structure. The goal of the computation is to fit individual micrographs of particles, like viruses or proteins (ion channels) to a hypothetical 3D structure. More precisely, images of nanoscale molecules embedded in ice are collected and are analyzed with an electron microscope. A 3D model is then built using the EMAN workflow. The workflow runs through several potentially parallel stages and, depending on the resolution of the required 3D model, iterates through several refinement loops.

The structure of the workflow does not require any transformation. Thus, in case of one iteration, the space required to store the transitive closure is fairly small. Figure 13 (top) compares the space required to store the transitive closure of the three approaches depending on the number of iterations of the refinement loop. As before, recursive queries and interval encoding require constant space, while the space required by the storing all paths approach grows exponentially.

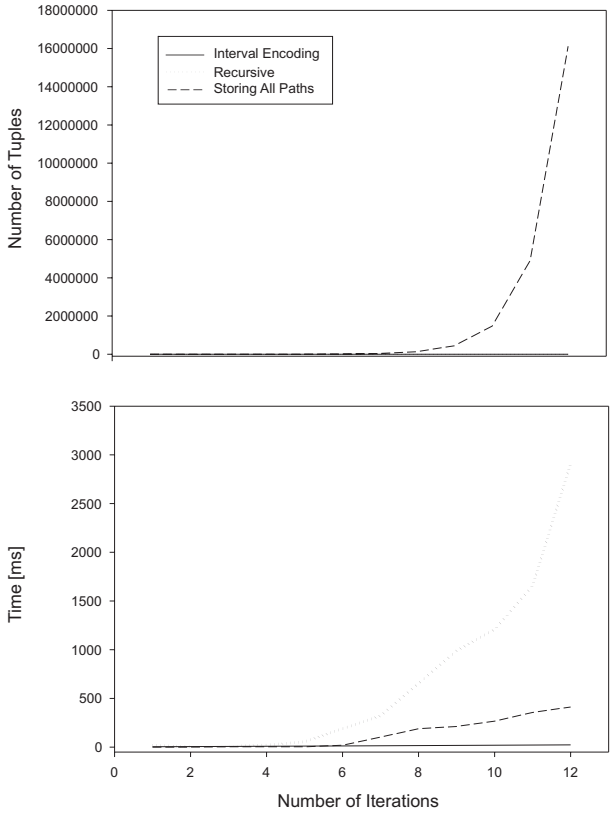

Figure 13: Space requirements for the EMAN workflow (top) and response time (bottom).

Figure 13 (bottom) shows the response time for queries using the different approaches with respect to the number of iterations over the refinement loops.

Figure 13 clearly demonstrate the benefits of our approach. The recursive approach does not require much space but its performance is not acceptable as the complexity of the workflow increases. Storing all paths has acceptable performance, but requires exponential growth in space as the complexity of the workflow grows. Interval encoding requires very little space for storage and has the best performance, independently of the number of iterations and, hence, independently of the size and complexity of the graph.

8. RELATED WORK

Although data lineage is an important problem, there is surprisingly very little work done on efficiently storing and querying lineage information.

The Trio [1] system is an extension of relational databases to support uncertainty and lineage. It bases lineage storage on methods described in [5] by storing the lineage information in additional tables. Retrieving the lineage information is then done as described in [11] using recursive queries.

The GridDB [19] system aims at providing a data-centric overlay for scientific Grid data analysis. Data is registered and processed in GridDB, similar to it being processed by

workflows. Lineage information is stored during the execution in the same tables as are the files registered in the system. For the purpose of retrieval of the lineage information, GridDB also uses the recursive mechanisms described in [11].

The interval encoding we use in this paper is described in [17] and has already been used for storing the tree hierarchy of XML documents [12, 23, 35].

9. CONCLUSIONS

The ability of tracking lineage is of paramount importance for scientific applications. Without being able to track back the origin of a given data set, no conclusive statement can be made about it, rendering it virtually useless. Several different systems have been implemented to track the lineage of individual data sets, however, none of them focuses on the efficient storage and retrieval of lineage information. They instead use recursive queries, which do not scale for very large workflows as we have shown in Section 2.

In this paper, we have presented an alternative approach for storing the transitive closure of dependency graphs of data sets and tasks. The new approach transforms, if necessary, an arbitrary graph into an interval encodable graph while maintaining the transitive closure, encodes it and stores it in the database. As the evaluation in Sections 5, 6, 7 demonstrates, this encoding allows for very fast lineage queries over the transitive closure for a wide range of workflow types without imposing an undue penalty in terms of storage.

10. ACKNOWLEDGMENTS

The work presented in this paper was in part supported by the European IST-FP6-15964 project AEOLUS (Algorithmic Principles for Building Efficient Overlay Computers) and by a grant from the Hasler Foundation (ManCom Project No. 2077).

11. REFERENCES

[1] P. Agrawal, O. Benjelloun, A. D. Sarma, C. Hayworth, S. Nabar, T. Sugihara, and J. Widom. Trio: A System for Data, Uncertainty, and Lineage. In *VLDB '06: Proceedings of the 32nd International Conference on Very Large Data Bases*, pages 1151–1154, 2006.

[2] G. Alonso, W. Bausch, C. Pautasso, and A. Kahn. Dependable Computing in Virtual Laboratories. In *ICDE 2001: Proceedings of the 17th International Conference on Data Engineering 2001*, pages 235–242, 2001.

[3] I. Altintas, C. Berkley, E. Jaeger, M. Jones, B. Ludascher, and S. Mock. Kepler: An Extensible System for Design and Execution of Scientific Workflows. In *SSDBM '04: Proceedings of the International Conference on Scientific and Statistical Database Management*, pages 423–424, June 2004.

[4] A. Bergou, B. Berriman, E. Deelman, J. Good, J. C. Jacob, D. S. Katz, C. Kesselman, A. Laity, T. Prince, G. Singh, M.-H. Su, and R. Williams. Montage: A Grid Enabled Image Mosaic Service for the National Virtual Observatory. In *Astronomical Data Analysis Software and Systems (ADASS) XIII*, 2003.

[5] D. Bhagwat, L. Chiticariu, W.-C. Tan, and G. Vijayvargiya. An Annotation Management System for Relational Databases. In *VLDB '04: Proceedings of the 30st International Conference on Very Large Data Bases*, pages 900–911, 2004.

[6] R. Bose and J. Frew. Lineage Retrieval for Scientific Data Processing: A Survey. *ACM Computing Survey*, 37(1):1–28, 2005.

[7] P. Buneman, S. Khanna, and W. C. Tan. Data Provenance: Some Basic Issues. In *FST TCS 2000: Proceedings of the 20th Conference on Foundations of Software Technology and Theoretical Computer Science*, pages 87–93, London, UK, 2000.

[8] P. Buneman, S. Khanna, and W.-C. Tan. Why and Where: A Characterization of Data Provenance. In *ICDT 2001: 8th International Database Theory Conference*, pages 316–326, 2001.

[9] W. Chrabakh and R. Wolski. GridSAT: A Chaff-based Distributed SAT Solver for the Grid. In *SC '03: Proceedings of the 2003 ACM/IEEE Conference on Supercomputing*, page 37, Washington, DC, USA, 2003.

[10] Y. Cui and J. Widom. Practical Lineage Tracing in Data Warehouses. In *ICDE '00: Proceedings of the 16th International Conference on Data Engineering*, pages 367–378, Washington, DC, USA, 2000.

[11] Y. Cui, J. Widom, and J. L. Wiener. Tracing the Lineage of View Data in a Warehousing Environment. *ACM Transactions on Database Systems*, 25(2):179–227, 2000.

[12] D. DeHaan, D. Toman, M. P. Consens, and M. T. Özsu. A Comprehensive XQuery to SQL Translation using Dynamic Interval Encoding. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 623–634, 2003.

[13] B. Dushnik and E. W. Miller. Partially Ordered Sets. *American Journal of Mathematics*, 63:600–610, 1941.

[14] S. Even, A. Pnueli, and A. Lempel. Permutation Graphs and Transitive Graphs. *Journal of the ACM*, 19(3):400–410, 1972.

[15] M. C. Golumbic and E. R. Scheinerman. Containment Graphs, Posets, and related Classes of Graphs. In *Proceedings of the Third International Conference on Combinatorial Mathematics*, pages 192–204, New York, NY, USA, 1989. New York Academy of Sciences.

[16] J. D. V. Horn. Online Availability of fMRI Results Images. *Journal of Cognitive Neuroscience*, 15(6):769–770, 2003.

[17] D. E. Knuth. *Art of Computer Programming, Volume 1: Fundamental Algorithms*. Addison-Wesley Professional, November 1969.

[18] C. Lekkerkerker and J. Boland. Representation of a finite graph by a set of intervals on the real line. *Fundamenta Mathematicae*, 51:45–64, 1962.

[19] D. Liu and M. Franklin. GridDB: A Data-Centric Overlay for Scientific Grids. In *VLDB '04: Proceedings of the 30st International Conference on Very Large Data Bases*, pages 600–611, 2004.

[20] S. J. Ludtke, P. R. Baldwin, and W. Chiu. EMAN: Semiautomated Software for High-Resolution Single-Particle Reconstructions. *Journal of Structural Biology*, 128(1):82–97, December 1999.

[21] R. M. McConnell and J. P. Spinrad. Modular Decomposition and Transitive Orientation. *Discrete Mathematics*, 201(1-3):189–241, 1999.

[22] S. Miles, P. Groth, M. Branco, and L. Moreau. The Requirements of Using Provenance in e-Science Experiments. *Journal of Grid Computing*, 5:1–25, 2005.

[23] G.-J. Na and S.-W. Lee. A Relational Nested Interval Encoding Scheme for XML Storage and Retrieval. In *Information Retrieval Technology*, pages 715–720, 2005.

[24] T. Oinn, M. Greenwood, M. Addis, M. N. Alpdemir, J. Ferris, K. Glover, C. Goble, A. Goderis, D. Hull, D. Marvin, P. Li, P. Lord, M. R. Pocock, M. Senger, R. Stevens, A. Wipat, and C. Wroe. Taverna: Lessons in creating a workflow environment for the life sciences: Research Articles. *Concurrency and Computation: Practice and Experience*, 18(10):1067–1100, 2006.

[25] C. Pautasso. JOpera: Process Support for more than Web services. http://www.jopera.org.

[26] A. Pnueli, A. Lempel, and S. Even. Transitive Orientation of Graphs and Identification of Permutation Graphs. *Canadian Journal of Mathematics*, 23:160–175, 1971.

[27] R. Rifaieh, R. Unwin, J. Carver, and M. A. Miller. SWAMI: Integrating Biological Databases and Analysis Tools Within User Friendly Environment. In S. C. Boulakia and V. Tannen, editors, *Data Integration in the Life Sciences*, volume 4544 of *Lecture Notes in Computer Science*, pages 48–58. Springer, 2007.

[28] K. Seymour, A. YarKhan, S. Agrawal, and J. Dongarra. NetSolve: Grid Enabling Scientific Computing Environments. In L. Grandinetti, editor, *Grid Computing and New Frontiers of High Performance Processing*. 2005.

[29] Y. Simmhan, B. Plale, and D. Gannon. A Survey of Data Provenance in e-Science. *SIGMOD Record*, 34(3):31–36, March 2005.

[30] S. W. Sorde, S. K. Aggarwal, J. Song, M. Koh, and S. See. Modeling and Verifying Non-DAG Workflows for Computational Grids. *IEEE Congress on Services*, pages 237–243, 9-13 July 2007.

[31] F. Stefan. 3-Interval irreducible partially Ordered Sets. *Order*, 11(12):97–125, 1994.

[32] E. Stolte and G. Alonso. Efficient Exploration of Large Scientific Databases. In *VLDB '02: Proceedings of the 28th International Conference on Very Large Data Bases*, pages 622–633, 2002.

[33] E. Stolte, C. von Praun, G. Alonso, and T. Gross. Scientific Data Repositories - Designing for a Moving Target. In *SIGMOD '03: Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, pages 349–360, New York, NY, USA, 2003. ACM.

[34] I. Taylor, M. Shields, I. Wang, and A. Harrison. The Triana Workflow Environment: Architecture and Applications. In I. Taylor, E. Deelman, D. Gannon, and M. Shields, editors, *Workflows for e-Science*, pages 320–339. 2007.

[35] V. Tropashko. Nested Intervals Tree Encoding in SQL. *SIGMOD Rec.*, 34(2):47–52, 2005.

[36] W. T. Trotter and J. I. Moore. Characterization Problems for Graphs, Partially Ordered Sets, Lattices and Families of Sets. *Discrete Mathematics*, 16:361–381, 1976.

[37] J. R. Ullmann. An Algorithm for Subgraph Isomorphism. *Journal of the ACM*, 23(1):31–42, 1976.

[38] W. M. P. van der Aalst, A. H. M. ter Hofstede, B. Kiepuszewski, and A. P. Barros. Workflow Patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.

[39] H. Veregin and D. Lanter. Data Quality Enhancement Techniques in Layer-Based Geographic Information Systems. *Computers, Environment and Urban Systems*, 19:23–36(14), 1995.

[40] E. Wolk. A Note on the Comparability Graph of a Tree. *Proceedings of the American Mathematical Society*, 16(1):17–20, February 1965.

[41] M. Yannakakis. The Complexity of the Partial Order Dimension Problem. *SIAM Journal on Algebraic and Discrete Methods*, 3(3):351–358, 1982.