Efficiently discovering critical workflows in scientific explorations

Qihong Shao, Peng Sun *, Yi Chen

Computer Science and Engineering Department, Arizona State University, United States

A R T I C L E   I N F O

Article history:
Received 11 November 2007
Received in revised form 1 May 2008
Accepted 25 June 2008
Available online 29 June 2008

Keywords:
Scientific workflow
Log
Join algorithm
Database system

A B S T R A C T

Existing workflow management systems assume that scientists have a well-specified workflow design
before the execution. In reality, a lot of scientific discoveries are made as a result of a dynamic process,
where scientists keep proposing new hypotheses and verifying them through multiple tries of various
experiments before achieving successful experimental results. Consequently, not all the experiments
in a workflow execution have necessarily contributed to the final result. In this paper, we investigate
the problem of effectively reproducing the results of previous scientific workflow executions by
discovering the critical experiments leading to the success and the logical constraints on their execution
order. Relational schema and SQL queries have been designed for effectively recording the workflow
execution log, efficiently identifying the critical experiments from the log, and recommending experiment
reproduction strategies to users. Furthermore, we propose optimization techniques for evaluating such
SQL queries according to the unique characteristics of the log data. Experimental evaluations demonstrate
the performance speedup of our approach.

1. Introduction

Workflows are widely used in various scientific fields, such as
chemical physics, astronomy, environmental science and bioinformatics. A scientific workflow represents a set of experiments per-
formed in an order that is consistent with certain constraints in
order to achieve a scientific goal. We differentiate the design and
the execution of a workflow [16]. The design captures the concep-
tual model of the workflow, including the format of its input and
output, the functionality of each experiment, as well as the logi-
cal constraints on experiment execution order. The execution of a
workflow represents a sequence of the experiment runs, consisting
of the actual input and output of each experiment, and the order of
the execution that satisfies the specification in the design.

In a lot of scenarios, scientific discoveries are made as a
result of a dynamic process, where scientists have a desired
goal in mind without a clear workflow design. To this end, they
make a hypothesis and then design experiments for verification.
Depending on whether the experiments succeed or not, they may
refine the hypothesis or propose a new one, and then perform
further experiments for validation. Usually many iterations of
hypotheses and experiments may be needed and the whole process
could take weeks or months. Once they succeed, it is important
that the scientists and their colleagues can reproduce the scientific
discoveries.

There are two challenges in reproducing the final result of
a workflow execution in scientific exploration when the work-
flow design is not available. First, how should we document
the log of all the experiments that have been performed such
that the storage and access to the log can be effective and effi-
cient? Database technology shows promising results for storing
and querying workflow designs and executions, as exploited in
[3,13,19]. However, these approaches require scientists to speci-
fy the designs of the workflow first, based on which relational
schemas are designed and corresponding SQL queries are gen-
ernated, therefore do not fit for scientific exploration applications.

Second, how should we identify the experiments that are criti-
cal to reproduce the workflow results? In scientific exploration,
not all the experiments in a workflow execution have necessarily
contributed to the final result, since scientists may not have a good
design before the execution and may keep proposing new hypothe-
ses and verifying them during the execution till the success. Exper-
iments that have parameter errors, execution errors, and/or design
errors are not necessary to be repeated for reproducing the same
result of the corresponding workflow execution. Identifying these
experiments can reproduce the workflow results more efficiently.

Research efforts have been made to process workflow execution
log in the absence of workflow design [1,6]. However, it focuses on
mining frequently occurring patterns in workflow execution log
in order to infer the design. There is a lack of integrated support
for storing and querying workflow executions. Furthermore, it is
assumed that every experiment run in the workflow execution log
is necessary in leading to the final success. Although this is often

* Corresponding address: Computer Science and Engineering Department,
Arizona State University, 85281 Tempe, AZ, United States. Tel.: +1 480 278 5406.
E-mail addresses: qihong.shao@asu.edu (Q. Shao), peng.sun@asu.edu (P. Sun),
yi@asu.edu (Y. Chen).

© 2008 Elsevier B.V. All rights reserved.

0167-739X/$ – see front matter © 2008 Elsevier B.V. All rights reserved.
doi:10.1016/j.future.2008.06.005
the case for business workflows, it may not always hold in scientific exploration where a scientist may keep proposing new hypotheses and verifying them during the workflow execution.

In this paper, we propose a new technique that enables efficient reproduction of a previously executed workflow. This is achieved by identifying critical workflows from the log of the previous workflow execution. Existing workflow management systems [5,7,2] record the log of workflow execution, but have not investigated how to leverage it for efficient workflow reproduction. Our system could be integrated into these existing workflow systems, which, upon a user’s request of reproducing a previously executed workflow, automatically presents the essential experiments with desirable settings and input parameters. Then the user can follow the recommendation to execute those experiments to correctly and efficiently reproduce previous workflow results without the need of re-executing unnecessary experiments.

In the following sections, we first identify the information that needs to be recorded in a workflow execution log, based on which we determine the data flow among experiments in Section 2. To effectively record a workflow execution log in a relational database management system (RDBMS) when the workflow design is absent, we propose a generic relational storage schema in Section 3. Then techniques have been designed to automatically discover the minimal set of experiments that must be performed in order to reproduce a scientific result by posing appropriate SQL queries in Sections 4 and 5. Although such SQL queries can be evaluated using an off-the-shelf database system, we investigate the unique characteristics of the workflow log data and optimization techniques for evaluating such SQL queries efficiently in Section 6. Empirical evaluation of the proposed algorithm is presented in Section 7. After discussing related work in Section 8, we finally conclude the paper and discuss the future plans in Section 9.

2. Preliminary and workflow execution log

A scientific workflow consists of a set of logically connected experiments that achieve a scientific goal. Each experiment is a self-contained activity or step with a specific functionality. An experiment has a set of input parameters and output parameters, which could be values of strings integer, or other data types. Two experiments in the workflow are logically connected if part of the output of one experiment is an input of the other experiment. The execution log of a workflow consists of a sequence of events, each of which records an experiment execution.

Definition 2.1. An event \( E \) of an experiment execution is described by a tuple \( (N, T, I, O) \), where \( N \) is the name of the experiment, \( T \) is the time when the experiment is performed, \( I \) and \( O \) are inputs and outputs of the experiment, respectively. We use \( E.c \) to denote the \( c \) component of an event \( E \), where \( c \) can be \( N, T, I \) or \( O \).

Since an experiment can take multiple inputs and produce multiple outputs, both \( I \) and \( O \) denote sets. Each distinct input and output has a unique identifier (such as a file name or material ID).

Example 2.1. Let us examine a workflow example in the area of molecular electronics (or Moletronics) [23] that investigates the electron transport properties of molecules. An execution log is shown in Table 1, consisting of a list of events in the order of their execution time. Each event is a run of one of the following experiments:

- **Compound synthesis** (C). The target molecules are either commercial or synthesized by conventional routine and characterized by mass spectrum to check the purity of the material.
- **Solution preparation** (S). The solubility of the target molecules are explored in different solvent, such as water, ethanol, HClO4 and so on. The compound is dissolved in one of the selected solvent in a certain concentration, like \( \sim 1 \) mol/liter.
- **Probe preparation** (P). A probe is made by cutting metal wire (such as Au, Ag and etc.) manually and the sharpness of its tip is checked using microscope. The probe is then cleaned via a regular cleaning procedure.
- **Test bed setting** (T). A test bed is set up, including appropriate power supply, electrometer configuration, etc.
- **Current monitoring** (M). The moving speed of the probe is optimized and a possible current is monitored.
- **Data collection** (D). Once desirable current traces are observed, approximately 1000 transients are collected.
- **Data analysis** (A). Based on the collected data, the Labview program is used to analyze the data and correlate the relationship between molecular conductance and molecular structures.

Scientists often need to test various possible combinations of parameters in different experiments before finally reaching a conclusive successful state. For example, in the solution preparation experiment, different solutions may be tested for a given singular molecule, since it can have different degrees of solubility in different solutions. Also, the purity of the material, the

<table>
<thead>
<tr>
<th>Name</th>
<th>Time</th>
<th>Input</th>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>C</td>
<td>( T_1 )</td>
<td>1.1 mmol dicarboxylic acid, 2.2 mmol HCTU</td>
<td>99% Fe-1</td>
</tr>
<tr>
<td>S</td>
<td>( T_2 )</td>
<td>Methanol, 99% Fe-1</td>
<td>1 mM methanol solution</td>
</tr>
<tr>
<td>P</td>
<td>( T_3 )</td>
<td>Gold wire</td>
<td>0.25 ( \mu )m probe</td>
</tr>
<tr>
<td>T</td>
<td>( T_4 )</td>
<td>1 mM methanolic solution, 0.25 ( \mu )m probe</td>
<td>Test environment 1</td>
</tr>
<tr>
<td>M</td>
<td>( T_5 )</td>
<td>Test environment 1, 0.2 V bias</td>
<td>0.0 mA</td>
</tr>
<tr>
<td>S</td>
<td>( T_6 )</td>
<td>HClO4, 99% Fe-1</td>
<td>1 mM HClO4 solution</td>
</tr>
<tr>
<td>T</td>
<td>( T_7 )</td>
<td>1 mM HClO4 solution, 0.25 ( \mu )m probe</td>
<td>Test environment 2</td>
</tr>
<tr>
<td>M</td>
<td>( T_8 )</td>
<td>Test environment 2, 0.2 V bias</td>
<td>0.0 mA</td>
</tr>
<tr>
<td>P</td>
<td>( T_9 )</td>
<td>Gold wire</td>
<td>0.15 ( \mu )m probe</td>
</tr>
<tr>
<td>T</td>
<td>( T_{10} )</td>
<td>1 mM HClO4 solution, 0.15 ( \mu )m probe</td>
<td>Test environment 3</td>
</tr>
<tr>
<td>M</td>
<td>( T_{11} )</td>
<td>Test environment 3, 0.2 V bias</td>
<td>0.2 mA</td>
</tr>
<tr>
<td>D</td>
<td>( T_{12} )</td>
<td>0.2 mA</td>
<td>1000 transients</td>
</tr>
<tr>
<td>A</td>
<td>( T_{13} )</td>
<td>1000 transients</td>
<td>Analysis report</td>
</tr>
</tbody>
</table>

\(^1\)Typically a log records various information of an experiment run, such as input and output parameters, the start time and finish time of an experiment run, the problems or failure, experimental setting, user/owner etc., as supported by [5,7,2]. For the conciseness of presentation, we show the input, output and time stamps of an event of experiment run, as only those are relevant to efficiently identify critical workflows. We also use \( T \) as time stamp, and the start time and finish time are individually recorded.
3. Recording workflow execution log in RDBMS

Having defined the workflow execution log, an immediate question is where and how to record the log. A log can be recorded as a text document, or it can be stored in a relational database (RDBMS). As argued by [19, 13], relational database technology should be fully exploited for workflow management for several reasons. First, the query capabilities provided by the RDBMS would allow users to query large amounts of experimental data as well as operational data for workflow data access, monitoring and diagnosis. It would also allow users to query the status of their personal jobs and data without compromising the privacy of other users. Second, the database techniques on query optimization, especially dynamic query optimization over distributed data sources to minimize CPU time, network and disk I/O can be useful for planning data-intensive scientific workflows. Third, transactional semantics in databases would provide guarantee of atomicity and durability of a sequence of actions in workflow execution. Fourth, concurrency control in databases in maintaining replica consistency and resolving conflicts can be useful for workflow with many simultaneously executing jobs. Furthermore, recovery techniques in databases could be used to recognize and handle multiple failure modes of distributed workflows. Due to all the advantages described above, it is good to use a RDBMS as the storage for workflow logs.

3.1. Existing approach

Before we present our proposed techniques, let us look at the existing approach [19] that leverages RDBMS for recording a workflow execution log based on the workflow design. A relation is created for each experiment that is involved in the workflow design, which records the name, time stamp, each input and output parameters of the experiment. A tuple in the relation records an execution of the experiment. Then the data flow among experiment executions can be specified using SQL joins over the input and output attributes across different relations according to the workflow design specification.

Example 3.1. In the molecular electronics workflow, suppose that each of the experiments S and T takes two input parameters and produces a single output parameter. As shown in Table(a) in Fig. 1, where each tuple records an execution. The relational schema for recording their experiment runs are as follows:

\[
\begin{align*}
T & \text{ Relation: } \{(\text{Name}, \text{Time}, I_1, I_2, O)\} \\
S & \text{ Relation: } \{(\text{Name}, \text{Time}, I_1, I_2, O)\}
\end{align*}
\]

The join results illustrate that the output produced by the event with name S and time stamp T_2 is fed as one of the input to the event with name T and time stamp T_4.

Furthermore, the workflow design states that the second input parameter of an experiment T is obtained from the output of an experiment S, then we can specify an inclusion dependency: for any value of attribute I_2 in relation T, there exists a value of attribute O in relation S. Therefore we can easily find out the data flow by joining the corresponding attributes in relation S and T.

However, if the workflow design is not available, although we can still create one relation for each possible experiment according to its input and output parameters, the relationship between the inputs and outputs of two experiments is unclear before the experiment execution. Furthermore, the connection between two experiments can vary for different workflows. Therefore, it is hard to automatically generate SQL queries to determine the data flow among experiment executions. In Example 3.1, without the knowledge of the workflow design, it is not clear which attributes in which relations should be joined together in SQL queries in order to determine the data flow and construct a workflow execution from an individual experiment.

3.2. Our proposal

In order to effectively record a workflow execution log and reconstruct data flow in the absence of workflow design, we propose a generic relational schema to record a workflow execution log as follows:

\[
\begin{align*}
\text{Exp}(\text{expID}, \text{wfID}, \text{Name}, \text{Time}), \\
(\text{ExpIn}(\text{expID}, \text{Input}), \\
\text{ExpOut}(\text{expID}, \text{Output})).
\end{align*}
\]

Where \text{expID} and \text{expID} are unique identifier of a workflow in the repository, and an experiment run in a workflow, respectively.

As we mentioned in Section 1, some existing workflow management systems provide their own log mechanisms, but we can still extract the information we need from their logs conveniently as described in [5].

The key attributes of each relation are underlined. For the Exp relation, an alternative key is (\text{Name}, \text{Time}, \text{wfID}). As will be shown in Section 5, SQL queries on this schema can be easily generated to build the workflow data flow from individual experiment execution.

Example 3.2. Given the above relational schema, the workflow execution log represented in Table 1 is stored in the relations in Fig. 2.

4. Identifying critical workflows from log data

The execution of a workflow can be conceptually represented as a graph \(G_t\), named as initial graph.
An initial graph \( G_i = (V, E, S) \) of a workflow execution log \( L \) is a directed acyclic graph constructed as follows. Each node \( v \) in \( G_i \) represents a distinct event in \( L, f(v) \), where \( f \) is the function that maps a node in \( V \) to the corresponding event in \( L \). All these nodes and an initial node \( v_0 \) consist of the node set \( V \). For any two nodes \( v_i, v_j \in V \), there is an edge from \( v_i \) to \( v_j \) in \( G_i \) if and only if \( \exists d \in f(v_i).O, d \in f(v_j).I \), where \( d \) is a data item transferring from event \( f(v_i) \) to \( f(v_j) \). There is also an edge from the initial node \( v_0 \) to each node \( v_i \), if \( \exists d \in f(v_i).I \), such that there does not exist \( v_j \in V, d \in f(v_j).O \). The edge set \( E \) consists of all such edges. \( S \subseteq V \) denotes the successful states of the workflow execution.

Note that we differentiate between workflow designs and executions [16]. A graph that represents a workflow design in general has loops, where a node denotes an experiment (type). While \( G_i \), which represents an execution of a workflow, is acyclic, where a node denotes an experiment run. Each run has distinct input data, output data, and a time stamp. An edge \((v_i, v_j)\) in \( G_i \) represents the data flow between two experiment runs. Specifically, one output of experiment run represented by \( f(v_i) \) is an input of the experiment run represented by \( f(v_j) \).

Data flow dependency, captured by the edges in an initial graph, is related but not equal to the execution order dependency among experiments. Let’s consider a fragment of our running example. In Fig. 6 we list all the possible experiment execution orders that satisfy the data dependencies. Among them, \( |I_0, C_0, S_0, P_0, T_0| \) is identical to the actual order of the workflow execution. However, any of them can correctly reproduce the final results of this workflow execution, even if the order is different. In general, all the nodes in a sequential relationship in the initial graph must be executed in sequential time order; while the nodes in a parallel relationship have no constraints on their execution order.

Example 4.1. Continuing our running example, the initial graph of the workflow execution log in Example 2.1 is presented in Fig. 3. For presentation conciseness, we use \( N_i \) to represent the event with name \( N \) that occurred at time \( T_i \). For instance, \( S_6 \) denotes the execution of the experiment of name \( S \) at time \( T_6 \) in the log. \( A_{13} \) is the final successful state. We also introduce node \( I \) to represent the initial node.

Not all the experiments executions are necessary in reproducing the same result of the corresponding workflow. There are three types of useless experiments:

(i) trial execution: experiments that are semantically relevant to the final results, but their current executions do not contribute to the results due to inappropriate input parameters.

(ii) unfinished execution: experiments that are semantically relevant to the final results, but of which the execution terminates abnormally and therefore not contributing to the final results.

(iii) irrelevant execution: experiments that are identified as semantically not relevant to the final results after their executions.

It is important to find a way in which we can remove all these unnecessary experiments from the initial graph. We propose to generate a critical graph which does not contain useless experiment executions. We observe that in the initial graph representation of the execution log, the nodes on the paths from the initial node to a node in the successful states \( S \) are considered to be critical since their outputs contribute to the final result of the workflow execution. Those nodes and the edges they consist of are the critical workflow graph.

Definition 4.2. A critical graph \( G_c = (V', E', S) \) is a subgraph of an initial graph \( G_i = (V, E, S) \), \( V' \subseteq V \), such that every node \( v' \in V' \) has a directed path from \( v' \) to a node \( s \in S, E' \subseteq E \), such that for every edge \( e' \in E' \), the nodes that \( e' \) connects to are in \( V' \).

\[\begin{array}{|c|c|c|c|}
\hline
\text{expID} & \text{Name} & \text{Time} & \text{wID} \\
\hline
1 & C & T_1 & 1 \\
2 & S & T_2 & 1 \\
3 & P & T_3 & 1 \\
4 & T & T_4 & 1 \\
5 & M & T_5 & 1 \\
6 & S & T_6 & 1 \\
7 & T & T_7 & 1 \\
8 & M & T_8 & 1 \\
9 & P & T_9 & 1 \\
10 & T & T_{10} & 1 \\
11 & M & T_{11} & 1 \\
12 & D & T_{12} & 1 \\
13 & A & T_{13} & 1 \\
14 & C & T_{14} & 2 \\
\hline
\end{array}\]

Table (1) Exp

\[\begin{array}{|c|c|c|c|}
\hline
\text{expID} & \text{Input} & \\
\hline
1 & 1.1mmol deacarbonylic acid & \\
2 & 2.2mmol HTC & \\
3 & methanol & \\
4 & gold wire & \\
5 & 0.25 \mu m probe & \\
6 & Test Environment1 & \\
7 & Test Environment2 & \\
8 & 99% Fe1 & \\
9 & 0.04A & \\
10 & 1.00A & \\
11 & Test Environment3 & \\
12 & 0.2mA & \\
13 & 100 measurements & \\
\hline
\end{array}\]

Table (2) ExpIn

\[\begin{array}{|c|c|c|}
\hline
\text{expID} & \text{Output} & \\
\hline
1 & 99% Fe1 & \\
2 & 1.00A & \\
3 & 0.25 \mu m probe & \\
4 & Test Environment1 & \\
5 & Test Environment2 & \\
6 & Test Environment3 & \\
7 & 99% Fe1 & \\
8 & 0.04A & \\
9 & 1.00A & \\
10 & 0.2mA & \\
11 & 100 measurements & \\
\hline
\end{array}\]

Table (3) ExpOut

Fig. 2. A generic solution for recording workflow execution log in RDBMS.

Fig. 3. Initial graph \( G_i \); Critical graph \( G_c \) consisting of only highlighted nodes and edges.
--- Join the Exp, ExpIn, ExpOut to create a logic view of experiments
--- Create view ExpLogic as
  Select E.EXPID, E.WFID, E.Name, E.Time, I.Input, O.Output
  From Exp as E, ExpIn as I, ExpOut as O
--- Where E.EXPID = I.EXPID and E.EXPID = O.EXPID
--- -- Self-join to retrieve the experiments that produce data
--- -- as an input to the experiment with ExpID =
--- Select Exp1.EXPID
  From ExpLogic as Expl, ExpLogic as Exp2
  Where Exp1.INPUT = Exp1.OUTPUT and Exp2.EXPID = x
  and Exp2.WFID = Exp1.WFID
---

**Fig. 4.** Construct edges in the initial graph.

```sql
WITH ExpCritical (EXPID, name, time, WFID, input, output) AS
  (--- Anchor member definition
  Select e.EXPID, e.name, e.time, e.WFID, e.input, e.output
  From ExpLogic AS e
  Where e.out = o
  Union ALL
  (--- Recursive member definition
  Select e.EXPID, e.name, e.time, e.WFID, e.input, e.output
  From ExpLogic AS e
  INNER JOIN ExpCritical AS c
  ON c.output = e.input and c.WFID = e.WFID
  )
  (--- Statement that construct the critical workflow graph
  Select distinct(*)
  FROM ExpCritical
  GO

**Fig. 5.** Constructing the critical workflow graph.

**Example 4.2.** The critical graph \( C_C \) for the initial graph \( G_I \) in Fig. 3 consists of the nodes that are highlighted in gray color as well as the edges connecting them.

The critical graph can only be identified based on the dataflow information between experiments, not the order of execution. An unnecessary experiment run can be performed before an experiment run in the critical graph, such as \( P_3 \) is before \( S_5 \); or it can be executed after an experiment run in the critical graph, such as \( T_7 \) is after \( S_6 \). Furthermore, as we discussed earlier, the order of the previous workflow execution may not be strictly enforced to ensure the correctness of workflow reproduction. For example, the order of \( S_6 \) and \( P_9 \) can be switched. As long as the logical dataflow is captured, a critical graph can be constructed, as to be discussed in Section 5, irrespective to the execution time of experiments.

5. **Retrieving critical workflow graphs using SQL queries**

Given the relational schema presented in Section 3 for recording workflow execution logs, the problem of identifying critical workflow graphs can be solved using SQL queries.

We first build a view `ExpLogic(expID, WFID, Name, Time, Input, Output)`, which represents experiment executions with both input and output information using a natural join on relations `Exp`, `ExpIn` and `ExpOut`. Since an experiment can take several inputs and produce several outputs, it may correspond to multiple tuples in `ExpLogic`. According to Definitions 4.1 and 4.2, there is an edge in the initial or critical workflow graph from the node representing event \( E_i \) to the node representing event \( E_j \), if and only if the output set of \( E_i \) has a non-empty intersection with the input set of \( E_j \), that is: \( E_i \cdot i \cap E_j \cdot O \neq \emptyset \). The data flow can be identified by checking whether there exists a tuple \( t_i \) in `ExpLogic` corresponding to \( E_i \) and a tuple \( t_j \) corresponding to \( E_j \), such that \( t_i\cdot Input = t_j\cdot Output \). The sample SQL statement is in Fig. 4.

The view of the critical workflow graph can be constructed using recursive SQL queries as shown in Fig. 5 supported in SQL Server 2005. Initially, the tuples in `ExpLogic` whose `Output` attribute values match the desired final workflow output are retrieved and composed of a view `ExpCritical`. Then the query recursively retrieves tuples whose `Output` values match the `Input` of a tuple in the current `ExpCritical` view through joins. That is, nodes that are connected to a node in the current critical graph become part of the expanded critical graph. The result of `ExpCritical` is finalized when no more tuples can be added. The initial workflow graph can be constructed similarly. Due to space limitation, we refer readers to [20] for more details.

6. **Effectively implementing join algorithm**

We have discussed how to retrieve critical workflows using SQL queries in the previous section. Such queries involve many join operations on the log table, which is very time-consuming. An effective optimization technique for processing join operations is highly desirable. In this section we propose a new join algorithm to evaluate those SQL queries efficiently by leveraging the special features in workflow execution logs.

6.1. **Time join algorithm**

We observe a special feature in the workflow log data, tuples representing experiments are recorded into the database in actual execution order, which provides us an opportunity to speed up query processing. Specifically, an experiment can only start its execution after all the experiments that it depends on have finished. In other words, if two tuples representing experiment executions \( t_i \) and \( t_j \) satisfy the following condition:

\[
t_i\cdot output = t_j\cdot input, \quad \text{then we have} \quad t_i\cdot Time < t_j\cdot Time.
\]

Intuitively, if an experiment execution \( E_i \) represented by \( t_i \) has an output parameter fed as an input parameter to another experiment execution \( E_j \) represented by \( t_j \), then \( E_j \) is performed before \( E_i \). Based on this observation, we propose to cluster the relations `Exp`, `ExpIn` and `ExpOut` according to the time stamps of the experiments. For a centralized workflow system, this can be easily implemented by appending newly generated tuples to the end of the existing relations. In parallel execution and/or distributed environment, the time stamp associated with each experiment run is generally based on a “ground truth” clock [12]. If a tuple that records the information of an experiment arrives a database out of order due to network latency, the orders of some tuples may be switched to enforce the table clustered by time stamps.

Therefore, when we perform joins on the `input` and `output` attribute values in `ExpLogic`, `ExpInitial` and `ExpCritical` relations, we only need to check the tuples whose time stamps satisfy the above constraints, rather than all the tuples in the relation.

The algorithm that performs joins is presented in Fig. 7. We name the proposed join algorithm as `time join` as it exploits the time stamp constraints of experimental runs to achieve efficiency. It takes two lists of tuples `list1` and `list2` as input, which have the same `WFID` attribute value and are both in ascending order of their time attribute. The output is the join of the tuples in `list1` whose `input` equal to the `output` of the tuples in `list2`. The algorithm scans the two lists of tuples and tests whether the input
6.2. The efficiency of time join

As observed in Section 6.1, if the input of a tuple $t_i$ at time $T_i$ is the output of another tuple $t_r$, then we must have $T_j < T_r$, where $T_j$ is the time of $t_j$. Since the tuples in the log data are stored in the increasing order of execution time, the time join algorithm can leverage this to limit the scope of table scan when searching tuples for match on Input and Output attributes.

The cost of the join algorithm depends on the number of tuples to be scanned until we find the match, referred as tuple distance. Since the log tables are clustered by time stamps, tuple distance is the number of tuples whose time stamps are from $T_j$ to $T_i$. The smaller the tuple distance is, the fewer scans and comparisons need to be made, the more efficient the time join algorithm will be. Considering our running example in Fig. 2, the tuple with expID 10 represents 1 mM HClO$_4$ solution in Input table, and the tuple with expID 6 represents 1 mM HClO$_4$ solution in Output table. The tuple distance between them is 4, which means 4 tuples need to be scanned and tested to find the match of join attributes.

Now the question is how to measure tuple distance from the experiment log data, so that we can estimate the cost of the time join operation. In other words, considering the experiments in the initial graph that represent a workflow execution, how do we measure the physical tuple distance?

Two factors in the workflow execution affect tuple distance. First, the more different experiment runs there are between two experiment runs that have data flow (referred to as experiment distance) in the initial workflow graph, the larger the corresponding tuple distance is, as the tuples are sorted according to execution time. Second, the more output parameters an experiment has (referred to as experiment parameters), the more tuples that this experiment corresponds to (specifically, one tuple per output parameter), and therefore the larger the corresponding tuple distance is. For instance, the experiment distance between $S_0$ and $T_0$ in Fig. 3 is 4, as there are 4 experiments executed after $S_0$ (inclusive) and before $T_0$ (exclusive). Since each of the 4 experiments has 1 output parameter, the tuple distance between the tuple with expID 6 in Output table and the tuple with expID 10 in Input table is also 4.

To illustrate this, suppose an experiment $N_i$ provides an output to experiment $N_j$ as shown in the initial workflow graph on the left hand side of Fig. 8. The tuple $V(j, x)$ in ExpOut that corresponds to $N_j$ and the tuple $V(k, y)$ in Expln that corresponds to $N_i$ are shown in the tables in Fig. 8. Here, we use $V(m, n)$ to stand for the tuple recording $n$th input/output in experiment $N_m$. To detect that the tuple $V(j, x)$ in ExpOut can be joined with tuple $V(k, y)$ in Expln, the time join algorithm compares the tuple $V(k, y)$ with the sequence of tuples in ExpOut from experiment $V(k \in T_0 \cup O_0 \cup D_0)$ back to experiment $V(j, x)$. Each comparison is shown as an arrow start from $V(k, y)$ to the tuples in ExpOut.

As we can see from the figures, all the ExpOut tuples in the experiments between $N_i$ and $N_j$ will be compared with $V(k, y)$ in table Expln, and each experiment has one tuple for each output parameter in table ExpOut. Therefore the number of comparisons required to join a tuple $V(k, y)$ in Expln with a tuple $V(j, x)$ in ExpOut is their tuple distance, which is proportional to the number of experiments between $N_i$ and $N_j$, and the number of output parameters that each such experiment has.

There are multiple tuples in table Expln which are corresponding to all the edges in the initial graph, so the overall cost of joining table Expln and ExpOut is therefore proportional to the size of Expln, the experiment distance, and the number of experiment parameters using the time join in Algorithm 1 (Fig. 7).

Formally, these three parameters that affect the efficiency of our algorithm are: experiment distance $D$, experiment parameters $O$, and the tuple number in table Expln $|N_{input}|$. The experiment distance $D$ from $N_j$ to $N_i$ is the number of different experiments in table ExpOut between $N_j$ and $N_i$.

$$D = \text{arg}[\text{no. of different experiments between } N_j \text{ and } N_i].$$

For each node, assume the max value of $O$ is $O_{max}$.

$$O_{max} = \text{arg} \max O(i), \forall i \in [j, k].$$

In worst case, the number of comparison is:

$$T_{worst} \propto O \ast O_{max} \ast |N_{input}|.$$  

Given formula (3), we can further analyze the impact of parameters, by fixing some of them and observing the changes of others. Theoretically, if we set experiment parameter $O$ and the edge number $|N_{input}|$ be constants, while increasing experiment distance $D$, the time of join algorithm increases linearly, the complexity of the algorithm is $O(D)$. On the other hand, if we fix the edge number $|N_{input}|$, and vary experiment parameters $O$, the complexity of time join is $O(D^2)$. The reason is that when we increase experiment parameters, the experiment distance $D$ would change together with $O$ and has an upper boundary $D$. In this way, the complexity becomes $O(D \ast D)\ast D$, which is $O(D^2)$. In the experiment section later, we design two individual data sets to prove our analysis here.

7. System implementation and experimental evaluation

In this section, we present the empirical evaluation on the proposed time join algorithm in Section 6.

### Algorithm 1 Time Join Algorithm (list1, list2, ”list1.in = list2.out”)

1: set cursor1 at the end of list1;
2: set cursor2 at the end of list2;
3: while cursor1 ≠ start(list1) do
4:   while cursor2.time > cursor1.time and cursor2 ≠ start(list2) do
5:      cursor2--; 
6:   end while
7:   temp = cursor2;
8:   while temp ≠ start(list2) do
9:      if cursor1.input = temp.output then
10:     output the tuples pointed by cursor1;
11:    break;
12:   end if
13:   temp--; 
14:   end while
15: cursor1--;
16: end while

Fig. 7. Time algorithm.
7.1. Experimental setup

The experiments were performed on a 3.6 Ghz Pentium 4 machine. The proposed join algorithm is implemented using C++ in Visual Studio 2005. All experiments were repeated 5 times independently with cold cache; and we report the average processing time disregarding the maximum and minimum values. We compare the performance of the proposed time join algorithm with the traditional join algorithms: indexed nested loop and default “best” join, provided by MS SQL server 2005. We set the parameters in the query to enforce a particular join algorithm to be used in query execution.\(^3\)

In order to analyze the efficiency of these join operations in various cases, we generate synthetic data of varying characteristics to the performance. As we have analyzed in Section 6.1, the experiment distance, as well as the experiment parameters affect the cost of the time join algorithm. We generate two data sets of synthetic logs with various parameters to evaluate the performance of time join algorithm, and it will be discussed in Sections 7.2 and 7.3 respectively.

7.2. The effect of experiment distance

To test the impact of experiment distance, we generate a set of log data, such that the size of relational tables that are used to store each log is the same (10,000 tuples), and the number of output parameters of each experiment is fixed to be 1. Each log has a different experiment distance \(D\), where \(D\) ranging from 1 to 700 in the data set.

The performance of time join, default best join and nested loop join on this data set is presented in Fig. 9. The graph on the left shows the wall-clock time required by each approach. As can be seen, the time required by time join is linear to experiment distance. The result is highly consistent with our theoretical analysis in Section 6.1.

On the other hand, the processing time used by the other two join implementations (nested loop and default setting) remains the same when the experiment distance increases. This is mainly because neither nested loop nor default best join leverages the time stamp property of the data when implementing the join operations. Therefore the full joining tables are considered for each log data, regardless the varying experiment distance (and therefore varying tuple distance).

We also observe that when the experiment distance is less than 700, time join is much faster than the other two join algorithms. When the experiment distance is larger than 700, the performance benefit of time join reduces and gradually becomes closer to nested loop join, since it needs to compare nearly all the tuples in the data table. However, two experiments \(E_i\) and \(E_j\) with an experiment distance 700 means that they have data flow (e.g. an output of \(E_i\) is an input of \(E_j\), or vice versa) and there are 700 experiments are executed between them. Most of workflow executions will not have such a large scale, and our proposed approach can achieve substantial performance speedup.

Furthermore, since the two traditional join algorithms need to perform system optimization and support more functionalities, which may reduce their performances on processing time, we normalize their absolute time and calculate the ratio of time join processing time with their processing time, to observe the trend with respect to the increasing experiment distance. The graph on the right of Fig. 9, shows the ratio of time join with the other two join algorithms. As can be seen, with the increase of experiment distance, the ratio increases steadily which proves our assumption, that time join benefits from the heuristic information, and the time required by time join is linear to experiment distance.

\(^3\) SQL server 2005 has a default “best” join algorithm, and we can infer that it is an “indexed hash join” after our experiments. The reason is that, when we set the test data to be linear, which has one input and one output matching within it, the execution time is close to 0 in the system.
7.3. The effect of experiment parameters

On the other hand, to test the impact of experiment parameters, we generate another set of log data, in which the size of relational tables that are used to store each log is the same (10,000 tuples), and the number of experiment parameters of each experiment is ranging from 1 to 40 in the data set. The performance of time join, default best join and nested loop join on this set of data are presented in Fig. 10. Similar to the previous experiment, the graph on the left shows the time required by each join algorithm. As can be seen, the time required by time join is square to experiment parameters, the same as our analysis. And the processing time used by the other two join implementations (nested loop and default setting) remains the same when the experiment degree increases. This is because neither of them benefits from the heuristic time information when implementing the join operations.

We also compare their ratio with respect to the number of experiment parameters. The graph on the right of Fig. 10, shows the ratio of time consumed by time join compared with the other two join algorithms. As can be seen, the ratio increases steadily which proves our assumption again.

8. Related work

Research has been conducted for managing workflows in both business and scientific domains. Compared with business workflows which are often control-flow oriented and have the design well-specified before execution, scientific workflows are often data-flow oriented and may not have a finalized design available before execution.

**Workflow modeling.** Several workflow management systems have been proposed to model scientific workflows. The **Griffyn project** [7,4] exploits the Grid technologies for collecting and analyzing distributed, terabyte-scale datasets in scientific and engineering projects. It allows users to use **Chimera** [7] to specify workflow components and their relationships in constructing an abstract directed acyclic graph to represent the workflow. **Kepler** [2] allows users to specify components and their relationships in a workflow, and automates the scheduling and execution of workflow in a Grid environment. The workflow management systems described above require scientists to specify the designs of the workflows before they are executed. In contrast, this paper addresses the issue of managing workflows given their execution log with no clear existing workflow design, which is often the case in scientific exploration.

**Managing scientific workflows in RDBMS.** Research efforts have been made in exploiting database technology in workflow management [3,4,13,19,11]. **Object Protocol Model (OPM)** [3] and **Zoo** [11] exploit an Object-Oriented data model to specify the relationships among experiments in the design, as well as to store and retrieve workflow executions. **GridDB** [13] stores workflows in relational databases, where users specify the programs in a workflow and their relationships using a functional data modeling language. **Memo tables** are used to record the correspondences between inputs and outputs of completed programs, and **process tables** are used to store the state information of currently executing programs. In Condor [19], each scientific program in a workflow is associated with an active table or an active view, where the invocation of the programs is expressed in SQL queries. These approaches work well for applications where workflow designs are specified before execution, and then relational schemas can be designed and SQL queries can be generated accordingly. As discussed in Section 3, these approaches are not suitable for applications where the workflow designs are not available before their executions.

**Mining workflows from log.** The problem of mining workflow patterns from an execution log has also been studied [1,6,8,10,9]. Based on the start and finish time of each experiment in a workflow log, possible time constraints among experiment executions, such as sequence, exclusive choice and parallel split, can be mined by applying techniques borrowed from Petri nets and machine learning.

This approach and our work both target applications where the workflow execution log is available but not the workflow design. Nevertheless, there are several key differences. First, existing work assumes that there is an implicit workflow design before the execution even though it is not explicitly recorded. Therefore every process in the workflow execution log is necessary in leading to the final success. We argue that this assumption may not always hold, especially in scientific explorations where the scientist may keep proposing new hypotheses and verifying them during the workflow execution. Second, patterns and constraints of execution order are mined based on the start and finish time of each process recorded in the log. As discussed in Section 2, additional information needs to be recorded in the log to detect unnecessary experiments in a workflow execution. Furthermore, the above work mainly focuses on business workflows, which are control-flow oriented. We emphasize the data flows in scientific workflows.

**Critical Path Analysis in workflow system.** There are existing works [21,22,15] on finding critical paths in a workflow, where the semantics of critical paths are defined differently compared with this paper. In [21,22], a critical path is defined as the longest execution path in a workflow which has time constraints (such as deadlines). Methods have been proposed to improve the workflow performance by intelligently assigning resources to workflow activities on the critical path. In [15], critical path is considered as the longest path in the task graph. Techniques have been proposed...
to schedule workflow tasks in the critical path. In this paper, a critical graph consists of the workflow experiments/tasks that are essential to execute to achieve the final successful state. We proposed techniques to efficiently identify the critical graph in order to efficiently reproduce a workflow result.

**Research topics in Online Workflow System.** Existing work [17] in this issue about online workflow systems focuses on the study of workflow-based Grid computing, with the goal of enabling users to compose complex Grid applications on distributed and unreliable hardware resources without taking care of lower level details. It helps users to design workflows which can leverage the Grid computing techniques. While our job is to take the workflow execution log as input and enable scientists to efficiently reproduce scientific discoveries.

9. Conclusions and future work

In this paper, we have examined the problem of effectively reproducing the results of exploratory scientific workflows. We have proposed a generic relational schema for leveraging RDBMS to record the workflow execution log which does not require workflow design information and therefore is suitable for scientific exploration applications. Not all the experiments performed in a workflow execution have necessarily contributed to the final desired results due to possible wrong parameters, abnormal termination, and/or design errors. We have designed SQL queries to identify the critical experiments that contribute to the successful states and the logical constraints on their execution order from the log database, such that workflow execution can be reproduced efficiently. We are designing and implementing optimization techniques for efficiently evaluating such SQL queries.

In the future, we plan to exploit a repository of workflow execution log data, and present relevant existing workflows to assist scientific users in constructing new workflows by querying the database effectively.

**References**


Qihong Shao received the M.S. degree from Renmin University of China, and current a Ph.D. student in the Department of Computer Science at Arizona State University. Her research interests include database systems, especially XML and scientific workflow.

Peng Sun is currently a Ph.D. student in the Department of Computer Science and Engineering at Arizona State University. He received the B.S. degree from Nanjing University of China in 2006. His research interests include database systems, especially scientific workflow and data provenance.

Yi Chen is an assistant professor in the Department of Computer Science and Engineering at Arizona State University. She received her Ph.D. degree in Computer Science from the University of Pennsylvania in 2005. Her research interests include query evaluation and optimization techniques for database systems and data streams, data integration in the presence of uncertainty and incompleteness, data model and query language design, and scientific data management. For more information, please refer to her home page http://www.public.asu.edu/~ychen127/.