Bridging Workflow and Data Provenance using Strong Links

David Koop[1], Emanuele Santos[1], Bela Bauer[2],
Matthias Troyer[2], Juliana Freire[1], and Cláudio T. Silva[1]

[1] SCI Institute, University of Utah, USA,
[2] Theoretische Physik, ETH Zurich, Switzerland

Abstract. As scientists continue to migrate their work to computational methods, it is important to track not only the steps involved in the computation but also the data consumed and produced. While this provenance information can be captured, in existing approaches, it often contains only weak references between data and provenance. When data files or provenance are moved or modified, it can be difficult to find the data associated with the provenance or to find the provenance associated with the data. We propose a persistent storage mechanism that manages input, intermediate, and output data files, strengthening the links between provenance and data. This mechanism provides better support for reproducibility because it ensures the data referenced in provenance information can be readily located. Another important benefit of such management is that it allows caching of intermediate data which can then be shared with other users. We present an implemented infrastructure for managing data in a provenance-aware manner and demonstrate its application in scientific projects.

1 Introduction

As the volume of data generated by scientific experiments and analyses grows, it has become increasingly important to capture the connection between the derived data and the processes as well as parameters used to derive the data. Not surprisingly, the ability to capture the provenance of data products has been a major drive for a wide adoption of scientific workflow systems [1–3]. By tracking workflow execution, it is possible to determine how an output is derived, be it a data file, an image, or an interactive visualization.

However, the common practice of connecting workflows and data products through file names has important limitations. Consider, for example, a workflow that runs a simulation and outputs a file with a visualization of the simulation results. If the workflow outputs an image file to the filesystem, any future run will overwrite that image file. If different parameters are used, or the simulation code is improved and the updated workflow is run, the original image is lost. If that image file were managed with a version control system, the user could retrieve the old version from the repository. However, if the user reverts the output image to the original version, how does she know how it was created?

Since there is no explicit link between the workflow instance (i.e., the workflow specification, parameters and input files) and the different versions of its output, determining their provenance is challenging. If we examine the provenance logs for the workflow runs, we will see that there are two runs that create the specified image file, one with the older simulation routine and the second with the newer one. We may be able to check timestamps in order to guess, but this is far from ideal. This problem is compounded when computations take place in multiple systems, and recording the complete provenance requires tying together multiple workflows through their outputs and inputs. As files are overwritten, renamed, or moved, provenance information may be lost or become invalid. As a result, maintaining an accurate provenance graph which ties processes and the data they manipulate requires a time-consuming and error-prone process.

While version control systems effectively track changes to files, such systems can only determine that changes have occurred, not how they came about. Provenance-enabled workflow systems, on the other hand, are able to capture how changes came about but do not provide a systematic mechanism for maintaining data provenance in a persistent fashion, i.e., given a file it may not be possible to determine which workflow instance generated it. We posit that a *tighter integration between scientific workflows and file management is necessary to enable the systematic maintenance of data provenance.*

Contributions. In this paper, we propose a new framework which, by coupling workflow provenance with the versioning of data produced and consumed by workflows, captures the actual changes to data as well as detailed information about how those changes came about. A persistent store for the data ensures that old inputs and results can be retrieved, and we can tie each version of a result to the provenance that details how the result was generated. We introduce the notion of a *strong link* which reliably captures the connection between a workflow instance and data it derives, and describe an algorithm for generating these links. Instead of relying on the user or ad-hoc approaches to automatically derive file names, strong links are identifiers derived from the file content, the workflow specification, and any parameters. As a result, they accurately and reliably tie a given workflow instance and its input and derived data.

Besides simplifying the process of maintaining data provenance, this approach has several benefits. By automatically capturing versions of data, it seamlessly supports exploratory tasks without requiring users to curate the data (e.g., managing the file names). It also provides a general mechanism for the persistent caching of both intermediate and final results—this is in contrast to previous approaches which supported only in-memory caching [4, 5]. The caching mechanism can be used not only to speed up workflow execution, but also to support check-pointing for long-running computations. In addition, the use of a managed data repository allows the creation of workflows that are location agnostic: unlike workflows that point to files in the filesystem, workflows can be shared and run in multiple environments unchanged. Last but not least, our approach is general and can be combined with existing workflow systems. We describe our implementation in the VisTrails system and present a case-study,

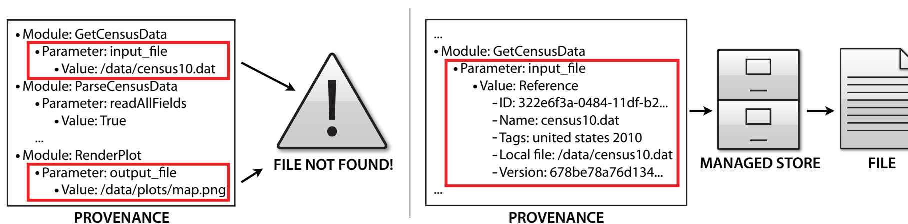


Fig. 1. When provenance information references file-system paths, there is no guarantee those files will not be moved or modified. We propose references that are linked to a persistent repository which maintains that data and with hashing and versioning allows for querying, reuse, and data lineage.

where the persistent data provenance infrastructure was deployed in a real application: managing data products in the context of the ALPS project [6].[3]

Outline. We begin by introducing our persistence scheme in Section 2, and then show how it can be applied to support data provenance in Section 3. In Section 4 we describe how our approach can be used to extend workflow caching strategies and for publishing scientific results. In Section 5, we describe how managed repositories can be shared among multiple users for both data access and caching. We describe an implementation of our scheme in Section 6, and describe its use in the ALPS project in Section 7. We highlight related work in Section 8 before concluding with future directions in Section 9.

2 Persisting Data Provenance Links

By integrating file management and version control with workflows, we aim to maintain stronger provenance by referencing data in a versioned, managed repository instead of via file paths (see Figure 1). This repository stores input, output, and intermediate data products, and can be used to facilitate caching and data sharing.[4] Similar to version control systems, this repository stores multiple versions of files, but to connect to workflow provenance information, it also contains metadata that represents identity and annotations.

Our approach to this problem is user-driven. As a user designs a workflow, she can specify which results (or input data) should be persisted in the repository. As we describe in Section 6, a possible implementation is to provide special workflow modules that can be connected to the output ports of modules whose results should be persisted (see the `ManagedIntermediateDir` module in Figure 6). When users run workflows using data from the repository, we can ensure that future provenance queries can not only identify the data involved in the computations but also retrieve the actual data. In addition, given provenance of a workflow execution, we can reproduce it using the exact versions of the data

[3] http://alps.comp-phys.org

[4] In the remainder of the text, we use the terms “repository” and “managed store” interchangeably.

used in the original execution. In these provenance applications, there is no need to archive data according to specific path-name conventions or remember to keep each separate version of the input data. Also, the automatic and transparent identification and versioning require little user involvement in maintaining these stronger links.

In what follows, we start by describing a scheme to derive reliable and representative ids for linking data products and their provenance. We also present the file-management infrastructure and the attributes we maintain in the managed repository, the differences in our storage depending on the role of the data, and how data should be updated and stored. Note that while we discuss *file* management, the techniques described can be easily extended to directories as well.

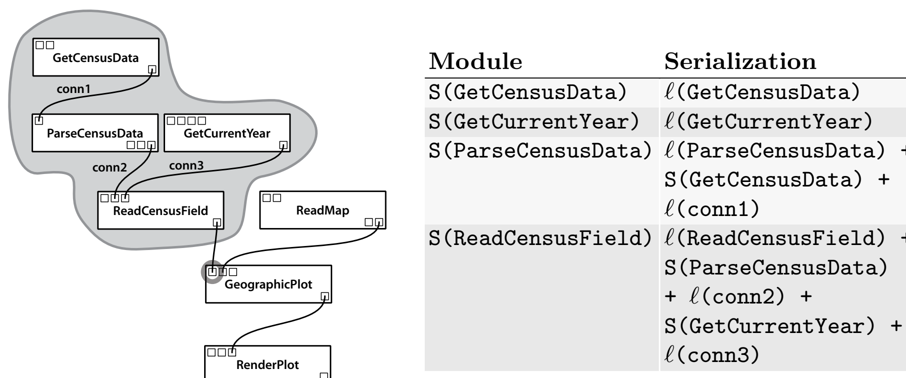
2.1 Deriving Strong Links

Our approach to deriving strong links was inspired by the in-memory caching mechanism proposed by Bavoil at al. [4] and the content hashing used in version control systems including `git` [7]. We use the signatures of workflows to identify intermediate and output data derived by the workflows, and content hashing to identify input data.

The central idea of caching in workflow systems is that any self-contained piece of a computation can be reused if the computation is deterministic and its structure, input data, and parameters do not change. For dataflows, we can formalize this concept by defining the *upstream subworkflow* of a module m in a workflow W as the subgraph induced by all modules $u \in W$ for which there exists a path from u to m in W (including m itself). Note that the existence of such a path implies that the results of u may have an effect on the computation of m. Then, if any module or connection in the upstream workflow of m changes, we must recompute m. Conversely, if the upstream workflow does not change, we need not recompute m, and can reuse results from a previous execution. Thus, for any other workflow W' that contains an upstream subworkflow U that also exists in W, we can reuse the intermediate results of U from W in W'.

Caching thus requires the identification of equivalent subworkflows. This would be expensive if we needed to perform graph matching, but we instead use a recursive serialization of the upstream workflow that allows us to quickly check the cache. We define the default *label* of a module m, $\ell(m)$, as the serialization of its type and parameter values ordered by parameter name. Note that individual module types can override this default label to better capture module state; for example, a module linked to a specific file would define its label based on the contents of the file—if that file changes, the label changes. Similarly, the *label* of a connection c, $\ell(c)$, is the serialization of the types of the ports it connects. Then, a canonical serialization of the upstream subworkflow of a module m is defined recursively as

$$S(m) = \ell(m) + \sum_{c \,\in\, \mathrm{UC}(m)} S(\mathit{source}(c)) + \ell(c)$$



Module	Serialization
S(GetCensusData)	ℓ(GetCensusData)
S(GetCurrentYear)	ℓ(GetCurrentYear)
S(ParseCensusData)	ℓ(ParseCensusData) + S(GetCensusData) + ℓ(conn1)
S(ReadCensusField)	ℓ(ReadCensusField) + S(ParseCensusData) + ℓ(conn2) + S(GetCurrentYear) + ℓ(conn3)

Fig. 2. The *upstream signature* S(M) for a module is calculated recursively as the signature of the module concatenated with the upstream signatures of the upstream subworkflow for each port and the signature of the connection.

where UC(m) is the set of upstream connections into m sorted by $\ell(c)$, *source* returns the source (upstream) module of the given connection, and $+$ is concatenation. The *upstream signature* is the SHA1 hash of this serialization.

Figure 2 shows an example workflow and the serialization of the upstream subworkflow of the `ReadCensusField` module. Note that the upstream subworkflow will not always be a tree, but the recursive serialization always branches like it is. This allows two topologically different upstream subworkflows to have the same signature, but when this happens, the computations must be identical. For example, consider a subworkflow with a single module m that connects upstream to two other modules. Whether those two modules connect upstream to a single module n or to two identical modules that both do the same computation as n will not affect the downstream computation of m. In addition, by using memoization, we can keep this computation efficient despite the added branching.

For input files, we define the signature as the hash of its contents. Then, if the file's contents changes, its signature changes even though its path or other identifying information may not. Note that we store the content hash separately as well so a file that is the output of one workflow and the input of another can be identified in both ways. Thus, the signature provides a strong link that contains a precise and accurate representation of the workflow fragment that derived a given result. As we describe in Section 3, we use this signature as the means to link a data product to the computation that derived it.

2.2 File Management

We are concerned with three roles for files in workflows: inputs, outputs, and intermediate data. Note that a single file may fill different roles depending on the workflow it is used in; an output from one workflow may be used as the input to another. Thus, the distinction between roles does not affect the use of data

in any situation, but rather determines what metadata can be captured, stored, and utilized. An output file can store information about the process that created its contents but an input file selected from the filesystem cannot. Similarly, an intermediate file need not be annotated at all if is used for caching, but files that are to be used again should be named and tagged to allow users to query for them.

Each file in the repository is uniquely identified by a combination of an id and a version string, and annotated with user-defined and workflow-generated information including its signature and content hash. By allowing a collection of files to share the same id, a reference to that id can be configured to always retrieve the latest version. This is helpful to a user who wishes to run a workflow with the latest version of a data set but does not wish to manually configure which is the latest version. On the other hand, reproducing a workflow execution exactly requires a particular version of the data, and thus identifying data by both the id and version guarantees that the exact data will be retrieved.

Input Files. An input file must reference an existing file, whether it is already in the managed store or only in the local system. Upon selection, we either use the existing identifier (from the store), or create a new unique identifier for the data. Note that we can detect whether the contents of a file already exists in the repository by computing the hash and checking for that hash in the repository. By default, changing the contents of the file creates a new version while changing the selected file creates a new id and version. Users can configure this behavior if necessary.

Output Files. The main difference between output and input files is that input files are not affected by changes in the rest of the workflow. For outputs, any changes to the part of the workflow that is upstream of the output file may affect its contents. In addition, it is less clear when an output is a new entity and when it is a new version of an existing entity. When only parameters change, the output is likely a tweaked version of the original, but when the input files are switched, the output is more likely new. By default, we create new versions for each execution but allow users to change this behavior in order to version the outputs. Like inputs, output files can be both stored in the persistent store for future reference and use and saved to a local file for immediate use or inspection.

Intermediate Files. An intermediate file is exactly the same as an output file except that it is not a sink of the workflow; execution continues after the file is serialized and the contents are used in further calculations. Such files can be used as checkpoints for debugging, but they can also be used to cache computational results in order to speed further analyses. Note that an intermediate file need not be manually annotated or named; it is defined by its signature–the serialization of the upstream subworkflow.

Customization. It may be necessary for users to configure the behavior of the persistence of files in the store in order to link similar files or maintain separate identities for data products. By selecting an existing reference and linking it to a local file, a user can tie the reference to a new local file. In addition, users can decide whether files are only persisted in the managed store or if they are also

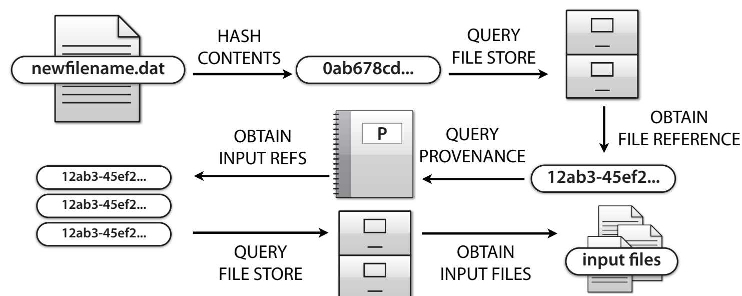


Fig. 3. Given a file which has been moved and renamed, we can use the managed file store and provenance to first locate the managed copy, and we can locate the original input files as well.

saved to local files. If they use a local file, they can configure whether the contents of the file should take precedence or whether a new version should always be obtained from the repository. Similarly, if the local file contents change, a user can choose whether those changes should always be persisted to the managed store.

3 Linking Provenance

Below we discuss how we exploit the strong provenance links to answer important queries. We also suggest how stronger links from data to provenance can be accomplished. With the advent of extensible file formats (e.g., HDF5[5]), it is possible to include direct links to provenance or even the provenance itself with the data. Finally, we present an application of the improved results from provenance queries in publishing scientific results.

3.1 Algorithms for Querying Linked Provenance

Perhaps the most basic provenance query is one that retrieves the lineage of a data product, specifically what input data and computations contributed to the result [8]. With only the provenance of the execution, a user may find the path to an input but even if a file still exists at that location, there is no guarantee it has not been modified. To protect against such problems, users store the exact data used with the provenance, manually archive all of the data, or add archival as part of the workflow process [9]. With our file management scheme, we can store the id, version, and content hash of any input as part of the provenance. Then, for lineage queries, we can return references that can be accessed from the provenance store using the id and version and verified using the content hash. Most workflow systems that support provenance capture also provide support for determining lineage queries [8].

Note that the content hash also gives us a way to locate the provenance of files that are un-managed and may have been moved to a different location or had their names changed. We begin by hashing the contents of the file, then

[5] http://www.hdfgroup.org/HDF5

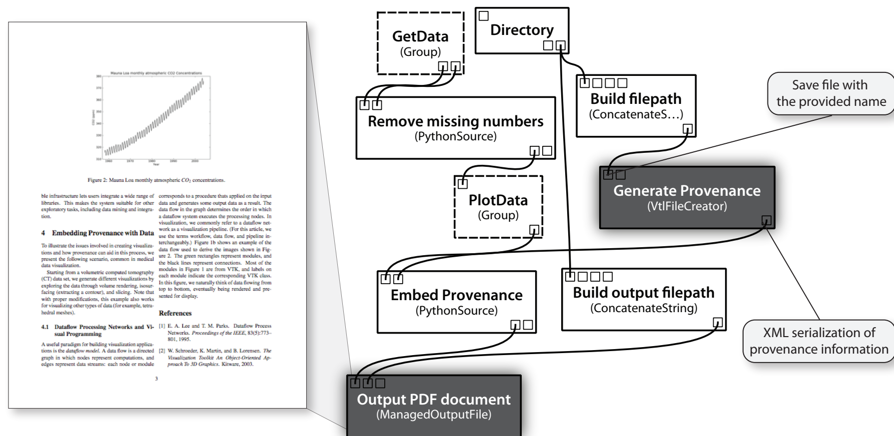


Fig. 4. Embedding provenance with data: provenance can be either saved to a separate file or serialized to XML and embedded in an existing file.

query the managed store for this content hash. The resulting entries have ids and versions for which we can then search our provenance for. Because the provenance contains these stronger references, we can also identify and return the input data via the managed store. An outline of this algorithm is shown in Figure 3.

Because we abstract workflows from a specific filesystem, the provenance of the workflow executions can be tied directly to the exact inputs and outputs. This ensures better reproducibility because the exact content can be retrieved; with links to the file names, we have no guarantee that the files at those locations were unchanged. To reproduce a workflow execution, we retrieve the workflow specification and execute it using the data pointed to by the managed file references. Recall, however, that some workflow specifications may include only data identifiers and not the versions of the data used. This allows a user to re-run a workflow with the latest data which is not what we desire for reproduction. Thus, we need to examine the provenance for the execution, retrieve the exact version specified by the provenance and modify the specification.

Another provenance query that our strong links solves is the lineage of data when the input of one workflow is the output of another. In the Second Provenance Challenge [10], teams were asked to answer provenance queries from outputs that were the result of running data through three consecutive workflows. One issue was the identification of data as it was transferred from the output of one workflow to the input of another. With the managed store, we allow users to designate inputs as the output of another workflow by assigning them the same id. Thus, when the first workflow changes, the second workflow will incorporate the changed results. Even if users do not use the same identifiers, we can perform provenance queries using the content hashes to link data across different workflows.

3.2 Embedding Provenance with Data

We have demonstrated methods to find the provenance of data by searching a provenance store for the hash of a given file. However, such methods depend on access to the provenance store. An alternative approach is to embed provenance with the data itself. With many file formats including HDF5 supporting annotations, it is possible to embed provenance information or links to provenance with the data. In directories of data, we can add an additional file that contains the same information. Then, verifying data or regenerating a data product can be accomplished by examining the provenance stored with the data.

We have developed a schema that allows a user to either link to or directly encode provenance information in a file. Information represented in this schema can be serialized to XML and embedded in an existing file or saved to a separate file. Figure 4 shows an example of a workflow using this schema. While a provenance link can refer to a local file, we provide support for accessing a central repository of provenance information. With a central repository, if the file is transferred to a different user or machine, the link remains valid. With a local reference, it will be more difficult to link back to provenance information.

4 Using Strong Links

4.1 Caching

Caching the intermediate results of workflow computations is useful to avoid redundant computations. If a user executes a workflow, we can reuse any intermediate results from that first execution in future executions [4]. Using our file management for intermediate files, we are able to add support for caching *files* to existing in-memory caching which means that cached data can be persisted across sessions. With this extension, we can also consider how to share cached data between different users as well. We begin by reviewing the in-memory workflow caching algorithm and then introduce an extension for caching across sessions using the managed file store.

In-memory Caching. Using the upstream signatures, we build a cache by labeling each intermediate result with its upstream signature. Dataflow computation proceeds in a bottom-up fashion; a sink (a module with no outgoing connections) requests data from all of its inputs which may in turn request data from their inputs and so on. Our caching algorithm works by hijacking this request for data and checking if the upstream subworkflow has already been calculated, returning the result from the cache when it exists instead of doing the computations.

Before executing any workflow, we compute the upstream signatures for each module in the workflow. Note that the recursive computation of all signatures is easily memoized. During workflow execution, before a module is set to execute, we check if that module's upstream signature exists in the cache. If it does, we return the result from the cache. If not, we proceed with the computation, and after the module finishes executing, we store the results of the computation in the cache labeled by the upstream signature.

There are some modules that may not perform deterministic calculations. We allow the module developer to designate such modules as non-cacheable. After a workflow with one or more such modules executes, we immediately remove all modules downstream of such a module from the cache.

Persistent Caching. We can extend the in-memory caching techniques to persist results to disk, allowing users to cache results across sessions or share intermediate results. Note that we need a serialization of the results of any module type in order to mirror the entire in-memory cache. In addition, saving every intermediate result to disk can needlessly slow down computation. For these reasons, we have developed persistent caching as a user-driven technique for intermediate files. We allow the user to connect a new module to the workflow that designates that the upstream subworkflow of the module should be cached. For non-cached computation, this module receives a file and passes it downstream. However, when the module finds that the signature associated with needed file exists in the cache, it retrieves the linked file without doing the upstream calculations.

This allows any module with serializable results to be persisted in a disk-based cache, but we can improve this process using the file management scheme described in Section 2.2. Using this scheme, additions to the cache are managed as intermediate files and cache lookup is a simple query to the store. In addition, users need not identify or in any way configure the intermediate files using for caching; the store assigns identity and stores signature information automatically. When the upstream workflow of the caching module changes, the cache lookup fails, and the store adds a new version of the intermediate file. Thus, a user does not lose any intermediate results when exploring different workflow configurations.

4.2 Publishing

When publishing scientific results, it is important to describe the lineage of a result. Providing data sets and computer code allows scientists to verify and reproduce published results and to conduct alternate analyses. In the past years, interest in this subject has increased in different communities which led to different approaches for publishing scientific results (see [11] for an overview). Our schema for embedding provenance with data can be combined with these approaches. In particular, it simplifies the process of packaging workflows and results for publication. In addition, we have also implemented a solution that allows users to create documents whose digital artifacts (e.g., figures) include a deep caption: detailed provenance information which contains the specification of the computational process (or workflow) and associated parameters used to produce the artifact [12].

5 Sharing Data

We have shown that maintaining workflow data in a managed store allows us to quickly locate existing data, store accurate provenance, and cache intermediate

results across sessions. Additional benefits can be gained from having multiple users share the repository. For example, if one user has run a time-intensive step of a calculation, making that result available to other users allows them to proceed with later steps without each re-computing the same result. Similarly, if one user has already added a specific file to the store, other users with access to that store can access the data without locating and copying the same data. Below, we describe both centralized and decentralized approaches for sharing managed data across systems, and note that the advantages and disadvantages mirror those encountered with version control systems.

Centralized Storage. With a central store, users may either read and write directly to a common repository or transfer data between a local repository and a central repository. If users have access to a common disk, it may be possible to simply store all managed files and metadata in a single store on that disk. Then, all users will access the same repository and automatically have access to each other's input, output, and intermediate files. However, this solution may become impractical for large numbers of users. A second problem is that whenever users do not have access to that disk, they are unable to access their managed data.

When a central store is added to individual local repositories, a user will always have access to the local repository but can also retrieve from and add to a central repository. This allows a set of geographically distant users to share common data. In addition, it allows users to maintain and access local data even when disconnected from the central store. However, we maintain an extra copy of the data in this case, and there may be overhead in transferring files, especially if the distance from the central store is far. In addition, it requires building and maintaining infrastructure.

Decentralized Storage. In a decentralized approach, users would advertise their data and allow other users to transfer data directly from their repository. A search for a particular piece of data by, for example, name or signature, would query individual systems instead of one store. If the desired file is found, it is transferred directly from the source location to the requesting user. Thus, unlike with the central store, data is only transferred when it is needed. Combined with P2P approaches, the transfer may be distributed over several machines. However, if a particular machine is offline, the data generated on that machine may not be available.

A hybrid approach that supports a central table of files but decentralized storage would allow users to locate files even if they were not currently accessible. Users would not push data to or pull files from the repository but rather register the available files as they are added and whenever that data is requested, directly transfer it to the requesting machine.

6 Implementation

We added file management to the VisTrails system [13] by introducing a new package that included module types for input, output, and intermediate files and directories. The package also includes code to set up the managed store as well

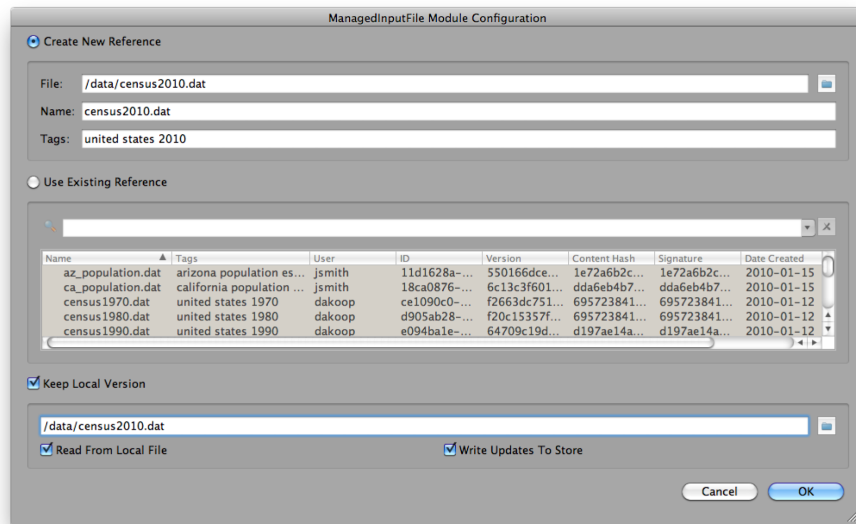


Fig. 5. The ManagedInputFile configuration allows the user to choose to create a new reference from a file on his local filesystem or use an existing reference from the managed store.

as navigate and update it through configuration dialogs. Our goal was to add this support in a way that changes little in workflow structure while providing ways for users to directly locate and identify data during workflow construction. Thus, users that normally only configure the path of an input file can do exactly the same for a managed input file module. In addition, adding an output file has fewer requirements; a user only needs to connect the data to be persisted to a managed output file module. The system generates unique ids and signatures automatically. At the same time, we provide methods for annotating data and configuring its storage and use.

The interface of our prototype implementation is shown in Figure 5. We define three new module types for files: `ManagedInputFile`, `ManagedOutputFile`, and `ManagedIntermediateFile` and their equivalents for directories. As described in Section 2.2, all share a common set of attributes and options. The key difference between inputs and outputs (or intermediates) is that outputs have a workflow-dependent signature. Thus, an input file needs to be manually identified by the user while an output file can be totally identified by its upstream signature.

A user can select a file by either referencing an existing identifier or by creating a new reference. When referencing a file that already exists in the managed store, the user can search the repository for metadata including name, tags, user id, date added, or a specific id or version. When creating a new reference, the user may provide a name and tagging information, and for input files, the local file that contains the data.

By default, an identifier for an input file changes when a new local path is selected but does not change if the contents of the file changes. In the second case, we maintain versions of the data, but update the “current version” whenever the contents changes. Thus, any user that wishes to use this data in another workflow will always get the latest data by referencing that identifier. Note that users may choose to link data to an existing reference even if that reference was initially linked to different data.

Storing Data. We use the `git` version control system [7] to manage files because it stores content independent of filesystem structure, and an `SQLite` database[6] to store its metadata. Thus, when the managed store is initialized for the first time, we create a `git` repository along with a database to store file information. While a reference is created and annotated during workflow design, the data is not persisted until execution. Upon execution, we save the file in the repository with its id (a UUID) as its name. We use `git` to add and commit the version of the file, and retrieve the content hash (`git` uses SHA1) and the version id (a SHA1 hash of the commit). Then, we update the database with the id, version, content hash, signature (if applicable), name, tags, user, and modification date.

Finding Data. In order to locate existing data, we provide methods to match content hashes and signatures as well as query the store for specific metadata like name or tag information. When a user selects a file, we can check the repository to see if that content has already been added by querying the database for the selected file's hash. If it does exist, we can prompt the user to reuse the existing reference. Additionally, when we execute a workflow, we can check to see if an intermediate file's signature matches one that already exists; if so, we can reuse that file instead of computing the upstream workflow. Finally, the configuration for managed file selection includes a free-text query field for the managed file database. A user can query for a specific name or tag to locate matching files that can be used as references. This is accomplished by querying the `SQLite` database and retrieving the matching id and, optionally, version.

7 ALPS Case Study

We have used the file management solution implementation for VisTrails with the ALPS project[7] (Algorithms and Libraries for Physics Simulations) [6]. ALPS is an open source software package that makes modern, high-performance algorithms for the simulation of quantum systems available to experimental and theoretical condensed-matter physicists. Typically, a simulation with ALPS consists of three steps:

- Preparing the input files describing the model to be simulated.
- Simulating the model using one of the ALPS programs. Such a simulation can take between minutes on a laptop for very small test cases and weeks on large compute clusters or supercomputers for demanding applications.

[6] http://www.sqlite.org
[7] http://alps.comp-phys.org/

- Collaboratively evaluating the “raw” simulation output by exploring & analyzing the data, comparing it to experimental data, and creating figures.

In one specific use case, we have simulated a quantum Heisenberg spin ladder, a model for quasi-one-dimensional copper oxide materials where magnetic excitations are suppressed at low temperature by an energy gap Δ [14]. The purpose of the simulation is to determine this gap Δ by calculating the magnetic susceptibility χ as a function of the temperature T and fitting it to the expression $\chi(T) \sim \frac{1}{\sqrt{T}} \exp^{-\Delta/T}$ [15]. We first use the “looper” program [16] of ALPS to calculate $\chi(T)$ and then use the exploration features of VisTrails to explore the data and find the optimal range $[T_{min}, T_{max}]$ for the non-linear fit. The results of this exploration are shown in Figure 6.

Persistent caching and provenance adds a number of important advantages for the ALPS users:

- Caching persistent files on a shared filesystem means that after one physicist runs the simulation, her colleague can modify the evaluation part of the workflow and explore the data without having to redo the time-intensive simulation.
- Identifying the cached files with the workflow signature avoids potentially critical mistakes of using old simulation results when input parameters to the simulation change. In our experience, simulations have often been recomputed only to ensure that the data has been produced with the latest version of codes and input files.
- Embedding provenance information in the data and figures gives immediate access to the provenance including any aspect of the simulation a physicist might wish to know. Since most projects involve collaborations with other scientists—often at different institutions—facilitating the exchange of data is very valuable. A common source of confusion is incomplete documentation of data sent to collaborators. Embedded provenance information has been invaluable in making remote collaborations more efficient.
- Decoupling the executions of different parts of the workflows using persistent data enables physicists to explore data without the need to always rerun the entire workflow—while still having the workflow provenance accessible when needed.

In Figure 6, we show one ALPS workflow along with plots resulting from an exploration of the fitting range parameter. The modules colored in blue, including the time-consuming simulation module “AppLoop”, were not run when this workflow was executed to create the plots because the output of the simulation had previously been persistently cached. Only the evaluation part of the workflow was re-executed when the fit range $[T_{min}, T_{max}]$ was modified. Note that the `SimulationID` module is striped blue and yellow; this is because it has two outgoing connections, one used in a file stored in the persistent cache and the other as part of a computation using the in-memory cache. Changing its value or structure would thus invalidate both cached results and all others downstream.

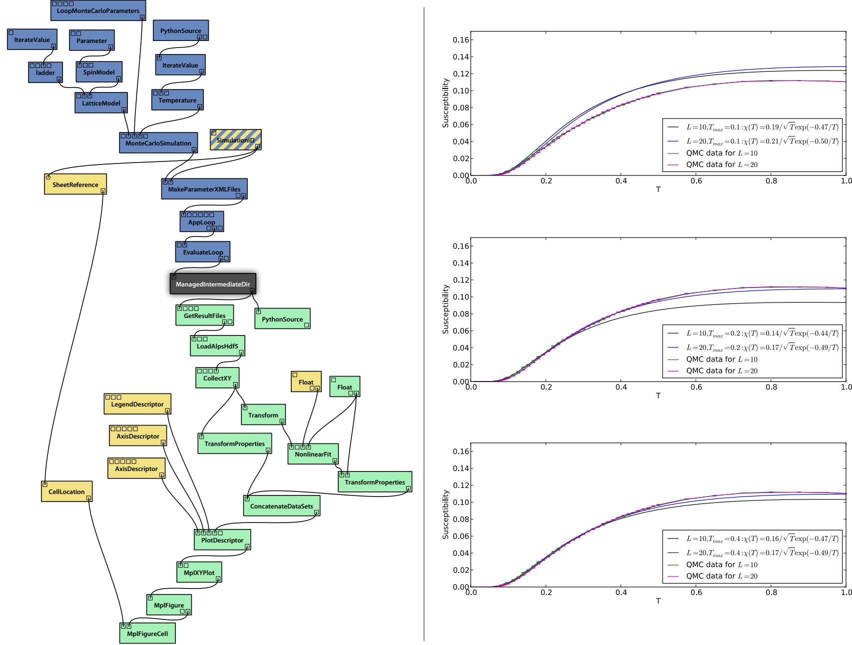


Fig. 6. An ALPS workflow colored by execution information and the results of a parameter exploration (i.e., multiple runs of the same workflow with different parameter values) of the fitting range. The colors of the modules indicate their status: blue modules were not executed because the data was found in a persistent directory on disk, yellow modules were cached in memory and green modules were executed.

8 Related Work

Data provenance consists of the trail of processing steps and inputs that led to the creations of a given data object. Tracking changes to files and entire directory structures is well-studied, and version control systems have been developed exactly for this purpose [17, 18]. However, such systems can only determine that changes have occurred, not how they came about. More recently, version control systems that focus on tracking content and directory structure separately have been developed (see e.g., [7]). Such systems identify files with hashing, and if duplicate files exist, the content is stored only once in the repository.

A number of workflow systems have been developed to help automate and manage complex calculations. The structure and abstraction provided by such systems have made them appealing to wide assortment of scientific domains. Many of these systems [19, 20, 13] have included provenance capture to document the process and data used to derive data products [1, 3]. Standard provenance captured by these systems, however, is not sufficient to identify exactly which workflow generated a specific file. In fact, in recent exercises to investigate requirements for querying and integrating provenance information, the lack of

effective means to identify intermediate and final results of workflows has been identified as an important challenge in provenance management [10, 21, 22]

Techniques have been developed to track provenance in databases [23]. These track fine-grained provenance, i.e., changes to individual data items. In contrast, our approach is targeted to (whole) files. In future work, we plan to investigate how we can adapt our system to utilize database provenance given encapsulated changes.

There is a significant amount of work with workflows that access and maintain curated data. In these cases, the provided ids or URIs are usually guaranteed to exist, and thus provenance information with them. Plale et al. have examined the issues involved in maintaining and cataloging large meteorological data, and noted the importance of allowing users to search and access this data [24]. Simmhan et al. have proposed data valets as a workflow-based method for facilitating the management of stores on the Cloud [25]. Note that if data for computations comes from or is persisted to a curated source, a separate managed store is not required to ensure access to those files. However, maintaining local copies of these files does allow users to run workflows even when they cannot connect to the store.

For curated scientific data, the identification of that data is important. There are standards for such identification including LSID [26] and DOI [27]. Our primary goal is orthogonal to these: we aim to maintain strong links between data and its provenance. We are not concerned with registering ids for our local persistent stores and use UUIDs to identify data. Identifying data by content hashes is has been accomplished using the MD5 and SHA1 hashes. Hashing has also been used in the context of secure provenance to maintain the confidentiality and integrity of provenance [28]. We use hashing to both identify and search for content as well as compute signatures for upstream subworkflows.

The problem with maintaining the data with workflows has been examined before. Some systems have provided specific modules for file management as part of workflow execution [9]. For example, after generating a data product, the result is not only displayed but also archived in a specific location or disk. This approach works well for static workflows, but for exploratory tasks, archival is not often included. The `cacher` package for R[8] provides a way to export verifiable statistical analysis and data in a tamper-proof scheme that utilizes hashing [29].

While we developed our store to aid users who use local files as data sources, our discussion of sharing the data in these stores overlaps many issues that have been considered. There already exist a number of solutions for managing scientific data on the grid and in cloud environments. GridFTP [30] and storage resource managers [31] have been developed to efficiently access data sets by utilizing networked resources. Such solutions can help provide faster access to data and infrastructure for transferring data across persistent stores.

[8] http://www.r-project.org

9 Conclusion & Future Work

We have presented file management infrastructure that can be integrated with workflow systems to provide strong links to data in provenance information. In addition, we have discussed how such links can be used to solve provenance queries, facilitate persistent caching, and impact scientific publishing. Finally, we have described our implementation of this system in VisTrails and its use in the ALPS project.

One important aspect that we have not addressed is how the persistent store should be managed. In theory, keeping all of the data manipulated by workflows would ensure full reproducibility, but this is impractical for large amounts of data. In future work, we plan to investigate different strategies for determining when data can be purged from the store; for example, cached data that has not been annotated. While our current implementation supports a rich class of queries over the information in the repository, we would also like to support queries that involve workflow specification and the data involved—for example, finding a workflow with a `ParseCensusData` module that accesses the `census2010.dat` file.

Another area for future study is the automatic identification of intermediate files for caching. While users can identify important way points, it can be tedious to add such modules to a large collection of workflows. By examining the timestamps of module execution in provenance, we may be able to determine which steps are time-intensive and could benefit from caching. Also, the size of the intermediate result may also be important; if a large file is generated by a time-intensive step, but the next step strips unneeded information away, it may be more efficient to store the file after the extra information has been removed.

References

1. Freire, J., Koop, D., Santos, E., Silva, C.T.: Provenance for computational tasks: A survey. Computing in Science and Engineering **10**(3) (2008) 11–21
2. Davidson, S.B., Freire, J.: Provenance and scientific workflows: challenges and opportunities. In: Proceedings of SIGMOD. (2008) 1345–1350
3. Davidson, S.B., Boulakia, S.C., Eyal, A., Ludäscher, B., McPhillips, T.M., Bowers, S., Anand, M.K., Freire, J.: Provenance in scientific workflow systems. IEEE Data Eng. Bull. **30**(4) (2007) 44–50
4. Bavoil, L., Callahan, S., Crossno, P., Freire, J., Scheidegger, C., Silva, C., Vo, H.: VisTrails: Enabling interactive multiple-view visualizations. In: Proceedings of IEEE Visualization. (2005) 135–142
5. Altintas, I., Barney, O., Jaeger-Frank, E.: Provenance collection support in the kepler scientific workflow system. In: Proceedings of IPAW. (2006) 118–132
6. Albuquerque, A., Alet, F., Corboz, P., Dayal, P., Feiguin, A., Fuchs, S., Gamper, L., Gull, E., Gürtler, S., Honecker, A., Igarashi, R., Körner, M., Kozhevnikov, M., Läuchli, A., Manmana, S., Matsumoto, M., McCulloch, I., Michel, F., Noack, R., Pawlowski, G., Pollet, L., Pruschke, T., Schollwöck, U., Todo, S., Trebst, S., Troyer, M., Werner, P., Wessel, S.: The alps project release 1.3: open source software for strongly correlated systems. J. Mag. Mag. Mat. **310** (2007) 1187
7. : git. http://git-scm.com

8. : First provenance challenge.
http://twiki.ipaw.info/bin/view/Challenge/FirstProvenanceChallenge (2006)
9. Mouallem, P., Barreto, R., Klasky, S., Podhorszki, N., Vouk, M.: Tracking files in the kepler provenance framework. In: SSDBM 2009: Proceedings of the 21st International Conference on Scientific and Statistical Database Management. (2009) 273–282
10. : Second provenance challenge.
http://twiki.ipaw.info/bin/view/Challenge/SecondProvenanceChallenge (2007)
11. Fomel, S., Claerbout, J.F.: Guest editors' introduction: Reproducible research. Computing in Science and Engineering **11** (2009) 5–7
12. Santos, E., Freire, J., Silva, C.: Information Sharing in Science 2.0: Challenges and Opportunities. CHI Workshop on The Changing Face of Digital Science: New Practices in Scientific Collaborations (2009)
13. : The VisTrails Project http://www.vistrails.org.
14. Dagotto, E., Rice, T.M.: Surprises on the Way from One- to Two-Dimensional Quantum Magnets: The Ladder Materials. Science **271**(5249) (1996) 618–623
15. Troyer, M., Tsunetsugu, H., Würtz, D.: Thermodynamics and spin gap of the heisenberg ladder calculated by the look-ahead lanczos algorithm. Phys. Rev. B **50**(18) (Nov 1994) 13515–13527
16. Todo, S., Kato, K.: Cluster algorithms for general- s quantum spin systems. Phys. Rev. Lett. **87**(4) (Jul 2001) 047203
17. : Concurrent Versions System http://www.nongnu.org/cvs.
18. : Subversion http://subversion.tigris.org.
19. : The Taverna Project http://taverna.sourceforge.net.
20. : The Kepler Project http://kepler-project.org.
21. : Third provenance challenge.
http://twiki.ipaw.info/bin/view/Challenge/ThirdProvenanceChallenge (2008)
22. Moreau, L., Freire, J., Futrelle, J., McGrath, R.E., Myers, J., Paulson, P.: The open provenance model: An overview. In: IPAW. (2008) 323–326
23. Cheney, J., Chiticariu, L., Tan, W.C.: Provenance in databases: Why, how, and where. Foundations and Trends in Databases **1**(4) (2009) 379–474
24. Plale, B., Alameda, J., Wilhelmson, B., Gannon, D., Hampton, S., Rossi, A., Droegemeier, K.: Active management of scientific data. IEEE Internet Computing **9**(1) (2005) 27–34
25. Simmhan, Y., Barga, R., van Ingen, C., Lazowska, E., Szalay, A.: Building the trident scientific workflow workbench for data management in the cloud. In: International Conference on Advanced Engineering Computing and Applications in Sciences. (2009) 41–50
26. Salamone, S.: Lsid: An informatics lifesaver. Bio-ITWorld (2004)
27. Paskin, N.: Digital object identifiers for scientific data. Data Science Journal **4** (2005) 12–20
28. Hasan, R., Sion, R., Winslett, M.: The case of the fake picasso: preventing history forgery with secure provenance. In: FAST '09: Proccedings of the 7th conference on File and storage technologies. (2009) 1–14
29. Peng, R.S., Eckel, S.P.: Distributed reproducible research using cached computations. Computing in Science & Engineering **11**(1) (2009) 28–34
30. Allcock, W., Bester, J., Bresnahan, J., Chervenak, A., Liming, L., Tuecke, S.: Gridftp: Protocol extensions to ftp for the grid. Global Grid Forum (2001) 3
31. Shoshani, A., Sim, A., Gu, J. In: Storage resource managers: essential components for the Grid. Kluwer Academic Publishers (2004) 321–340