Core Provenance Library

# Introduction

Provenance is metadata that describes the history of a digital object: where it came from, how it came to be in its present state, who or what acted upon it, etc. It is especially important in computational science, where it enables the researches to precisely track how each document came into existence, provides a means to experimental reproducibility, and aids them in debugging what went wrong during a computation.

The adoption of provenance among computational scientists is low, because most existing systems require the users to adopt a particular tool set in order to benefit from their functionality, such as the requirement to use a particular programming language, operating system, or a workflow engine. Core Provenance Library (CPL) takes the opposite approach by enabling the scientists to easily integrate provenance collection to their existing tools. We believe that this approach would increase the adoption of provenance in computational science.

Core Provenance Library is designed to run on a variety of platforms, work with multiple programming languages, and be able to use a several different database backends. An application would use the library’s API to disclose its provenance by creating provenance objects and disclosing data and control flow between the objects. The library would take care of persistently storing the provenance, detecting and breaking the cycles, and providing an interface to query and visualize the collected provenance.

This document describes the API specification and the design of Core Provenance Library.

# Important Concepts

Provenace Object: An object for which the provenance is tracked. It can correspond to a persistent object in the system, such as a file or a database table, or to a transient object, such as a process, socket, or a pipe. An object is uniquely identified by an object ID (equivalent to the *pnode number* in PASS), which is automatically assigned to an object by CPL upon creation. An object is required to have the following properties, which can be used to look up the object ID in the database:

* Originator: The application that created the object. The application is responsible for unique naming of its objects. The *originator* field thus acts as a namespace.
* Name: The object name. If the name is not unique, CPL returns the most recently created object with the requested name when performing a lookup. However, we recommend that an application uses unique names for its objects.
* Type: The object type, such as a file, a process, or a database table.

Version: The version of a provenance object. The combination of the object ID and the version number uniquely identifies a node in the provenance graph. CPL versions its objects in order to avoid cycles in the provenance graph. For example, if process P read file F and then wrote back to file F, we would normally get a cycle (the edges in the picture are *dependency edges* – i.e. in the opposite direction of the data flow):

**F**

Provenance cycles are unacceptable, because they imply that a state of an object in the past depends on the future state of another object. We can avoid the cycle using versions. In our example, we thus record that P read an older version of F and created a new version of F:

**F** ver. 1

**F** ver. 2

Data Dependency: The data associated with object A (or an in-memory state in the case of transient objects) depend on the data associated with object B. We recognize several types of data dependencies:

* Input: The most generic type of data dependency.
* Translate: Object A is a translation of object B. For example, a database table is a translation of a database file on disk that stores the table.
* Copy: Object A is an exact copy of object B.
* IPC: Data flow between two processes. This type of data dependency can be treated equivalently to the *input* dependency.

Control Dependency: A relationship between two processes. We recognize several kinds of control dependencies:

* Parent (START): Process A was started by process B. This is similar to the *fork-parent edge* in PASS.
* Control: Process A was controlled by process B, such as by sending “pause” or “resume” commands, but without transferring any data between the two processes.

CONTAINER: A provenance object that is composed from multiple smaller objects. For example, a database is a container composed of one or more database tables.

Provenance of Provenance: The source of the given provenance record. CPL automatically keeps track of which records were created by which application.

# API Specificiation

This section describes the API that an application can use to disclose its provenance. We include only C bindings; bindings to other programming languages are omitted for brevity.

cpl\_initialize: Initialize the CPL bindings for the current process.

Arguments:

* cpl\_backend\_t\* backend: The interface to an initialized database backend (described below).

Returns: CPL\_OK or an error code.

cpl\_destroy: De-initialize the CPL bindings for the current process.

Returns: CPL\_OK or an error code.

cpl\_create\_object: Create a new provenance object.

Arguments:

* const char\* originator: The unique ID of an application that creates the object.
* const char\* name: The object name.
* const char\* type: The object type.
* cpl\_id\_t container: The object ID of the container to which the object belongs, or CPL\_NONE.

Returns: The object ID (a positive number), or an error code (negative).

cpl\_lookup\_object: Lookup an object in the database.

Arguments:

* const char\* originator: The unique ID of an application that created the object.
* const char\* name: The object name.
* const char\* type: The object type.

Returns: The object ID (a positive number), or an error code (negative).

cpl\_data\_flow: Disclose a data flow. CPL would translate this into a data dependency. We decided to define the API in terms of data flow rather than the data dependencies, because we believe that it would be easier to use by our intended audience. We might need to revisit this choice in the future depending on the feedback that we would get.

Arguments:

* cpl\_id\_t data\_source: The object ID of a data source (from which the data is flowing).
* cpl\_id\_t data\_destination: The ID of the object to which the data is flowing.
* int type: The type of data flow (data dependency), such as CPL\_DATA\_INPUT, CPL\_DATA\_TRANSLATION, or CPL\_DATA\_COPY.

Returns: CPL\_OK or an error code.

cpl\_control: Disclose a control operation. CPL would translate this into a control dependency.

Arguments:

* cpl\_id\_t controller: The ID of the object that originated the control operation.
* cpl\_id\_t controlled: The ID of the object that received the control operation.
* int type: The type of the control operation, such as CPL\_CONTROL\_START or CPL\_CONTROL\_OP.

Returns: CPL\_OK or an error code.

# Database Backend Interface

CPL communicates with the database backend using the interface cpl\_backend\_t described in this section. CPL requires that each individual operation is atomic and durable, and that it preserves the database consistency. It does not require the atomic guarantee across multiple operations. Each interface function accepts cpl\_backend\_t\* backend as an argument in addition to the arguments listed below.

cpl\_db\_destroy: De-initialize the database backend.

Returns: CPL\_OK or an error code.

cpl\_db\_create\_object: Create a new provenance object (including its 0th version) and generate a new unique ID.

Arguments:

* const char\* originator: The unique ID of an application that creates the object.
* const char\* name: The object name.
* const char\* type: The object type.
* cpl\_id\_t container\_id: The object ID of the container to which the object belongs, or CPL\_NONE.
* cpl\_version\_t container\_version: The version of the container object.
* cpl\_id\_t record\_originator: The ID of the process that generated this provenance record (provenance of provenance).

Returns: The object ID (a positive number), or an error code (negative).

cpl\_db\_lookup\_object: Lookup an object in the database.

Arguments:

* const char\* originator: The unique ID of an application that creates the object.
* const char\* name: The object name.
* const char\* type: The object type.

Returns: The object ID (a positive number), or an error code (negative).

cpl\_db\_create\_version: Create a new version of the provenance object.

Arguments:

* cpl\_id\_t object\_id: The object ID.
* cpl\_version\_t version: The version number to create.
* cpl\_id\_t record\_originator: The ID of the process that generated this provenance record (provenance of provenance).

Returns: CPL\_OK or an error code.

cpl\_db\_get\_version: Get the latest version of the given object.

Arguments:

* cpl\_id\_t object\_id: The object ID.

Returns: The version number (a nonnegative number), or an error code (negative).

cpl\_db\_get\_record\_originator: Get the ID of the process that created the given provenance record (i.e. provenance of provenance).

Arguments:

* cpl\_id\_t object\_id: The object ID.
* cpl\_version\_t version: The object version.

Returns: The record originator ID (a positive number), or an error code (negative).

cpl\_db\_get\_immediate\_ancestors: Get list of immediate ancestors for all versions of the given object.

Arguments:

* cpl\_id\_t object\_id: The object ID.
* cpl\_version\_t version\_hint: The version number to create if CPL knows what is the latest version of the object (which is usually the case if CPL is running as a service rather than as a library – see below), or CPL\_VERSION\_NONE otherwise.
* cpl\_id\_version\_t\* buffer: The output buffer.
* size\_t buffer\_size: The size of the output buffer.

Returns: The number of returned results (a nonnegative number), or an error code (negative).

cpl\_db\_has\_immediate\_ancestor: Determine whether any version of the given object has the object query\_object (with the version number query\_version or earlier) as one of its immediate ancestors.

Arguments:

* cpl\_id\_t object\_id: The object ID.
* cpl\_version\_t version\_hint: The version number to create if CPL knows what is the latest version of the object (which is usually the case if CPL is running as a service rather than as a library – see below), or CPL\_VERSION\_NONE otherwise.
* cpl\_id\_t query\_object: The ID of the potential ancestor.
* cpl\_ version\_t max\_version: The max version number of the query\_object to consider.

Returns: A positive number if yes or zero if no – or a negative error code.

cpl\_db\_create\_dependency: Create a data or a control dependency edge. Note that this function does not take the record\_originator (provenance of provenance) argument. Instead, CPL ensures that the provenance node identified by from\_id and from\_version has the proper record\_originator and issues a freeze if necessary.

Arguments:

* cpl\_id\_t from\_id: The object ID of the source of the data dependency edge (usually the destination of the data flow or the controlled object).
* cpl\_version\_t from\_version: The version of the given object.
* cpl\_id\_t to\_id: The object ID of the target of the data dependency edge (source of the data flow).
* cpl\_version\_t to\_version: The version of the given object.
* int type: The type of the dependency, such as CPL\_DATA\_INPUT, CPL\_DATA\_TRANSLATION, CPL\_DATA\_COPY, CPL\_CONTROL\_START, or CPL\_CONTROL\_OP.

Returns: CPL\_OK or an error code.

# CPL Architecture

The two plausible choices for CPL architectures are:

1. CPL as a Library: Each application that uses CPL has its own copy of the entire system, performs its own cycle detection, and opens its own database connection:

Application

**CPL**

Database

Application

**CPL**

Application

**CPL**

1. CPL as a Service: The CPL library that an application links to is just a thin client that connects to a single instance of the Core Provenance Service (a daemon on UNIX):

Application

**CPL**

Database

Application

Application

**CPL**

**CPL**

**Core Provenance Service**

The advantage of the first approach is its simplicity from the developers’ and system administrators’ point of view, since they do not need worry about a separate process. It has the potential to reduce the number of IPC’s since there is no need to communicate with an external process, but it puts more stress on the database, and it increases the amount of data that needs to be transmitted between the database and each instance of the library.

If CPL runs as a library (the first approach), it cannot cache any data, but instead, it needs to issue a large number of queries and rely on the database cache to make sure that they perform well. For example, the *cycle avoidance* algorithm needs to query the database using cpl\_db\_has\_immediate\_ancestor every time a new dependency is added, so that it can determine whether to freeze the current version of an object. However, if CPL operates as a client of a single provenance service, the information required to answer the query can be cached by the service. Similarly, CPL needs to know the latest version of each object that it operates on, which in the former case involves a large number of cpl\_db\_get\_version queries – most of which would be unnecessary in the latter case.

There are pros and cons to both of these approaches, and it is not entirely clear which of the two would perform better. Determining which approach is more suitable for the use in production systems is a part of our future work.

# CPL RDF/SPARQL Backend

The RDF backend uses the following schema (where POP stands for “provenance of provenance”):

**Provenance Object**

Type

Name

Originator

**Version Node**

Version

POP

Timestamp

POP

Version

In-Container

Previous

Input: <type code>

The “Version” edge is included between the node that represents a provenance object and each of its version nodes. This allows an easy access to the object attributes such as Name or Type from each provenance node, and it also enables the backend driver to be able to quickly answer the cpl\_db\_has\_immediate\_ancestor query using a single SPARQL query (SPARQL does not support transitive closure).

# CPL ODBC Backend

The ODBC backend uses the following relational schema:

**cpl\_objects**

id (P)

originator

name

type

container\_id

container\_ver

**cpl\_versions**

id (P)

version (P)

pop

timestamp

**cpl\_ancestry**

from\_id (P)

from\_version (P)

to\_id (P)

to\_version (P)

type