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, a combination of which uniquely identifies the object in the database:

* Originator: The application that created the object. The application is responsible for unique naming of its objects, so the *originator* field thus acts as a namespace. We strongly encourage, but do not require, that the originator field follows the same naming convention as Java packages – a hierarchical naming scheme, in which the top level domain comes first, and the domain names are separated by dots. For example, CPL’s originator name is edu.harvard.pass.cpl.
* Name: The object name. We recommend that the object names are unique within the *originator*, but we do not enforce this rule by default, since several provenance-aware applications that were originally designed for PASS depend on this feature. If there are multiple objects with the same name, type, and originator, CPL’s lookup function returns the most recently created object.
* 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 (or the combination of the originator, name, type, and the version number if the names are unique) 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

The library uses the *Cycle Avoidance* algorithm by default, because it is both fast and light on system resources.

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.
* IPC: Possible data flow between two processes. This type of data dependency can be treated equivalently to the *input* 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.

An IPC message sent from one process to another process can be a control message, a data flow, or both. If we do not know the nature of the IPC, it is safer to assume that it involved a data flow, which is why CPL considers IPC to be a data dependency. If the nature of the IPC message is known, we strongly encourage the developers label the data dependency using an appropriate dependency type such as *Input* or *Control* (described below).

The best way to explain the Translate edge is on a relational database. Let F be a database file, and let T be the table that F represents. Ideally, we would like to say that T stands for F – which CPL currently does not support (*stands-for* edges have been proposed in the literature, but they are still a subject to research). Instead, the *Translate* edge allows you to assert that T is a translation of F – essentially the same object as F, but in a different format. For example, when a database program starts up, it reads file F and creates object T based on the contents of the file. T is thus an in-memory representation of F. When the database process writes T back to F, it translates T into F. The provenance graph thus looks like this:

**F** ver. 1

**F** ver. 2

**T** ver. 1

The database is responsible for creating an *Input* or a *Copy* edge between this instance of T and the corresponding instance of the table from the previous execution of the database process.

Control Dependency: A control 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. No data was transferred 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. Containers in CPL are just object attributes, and they do not participate in data flow, control flow, or cycle detection. Containers in provenance are still an active research topic. Full support of containers in CPL is thus a subject to future work.

Session: A running instance of a provenance-aware application. CPL automatically keeps track of all sessions in the database, recording the application name, PID, user name, and the MAC address of the computer on which the application is running.

Provenance of Provenance (PoP): The source of the given provenance record. CPL automatically keeps track of which records were created in which session; this information is not disclosed via the public API.

# C 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.

## Data Types

cpl\_id\_t: A 128-bits-long globally unique identifier used to identify provenance objects and sessions. The ID can be copied in C using cpl\_id\_copy() or using a simple variable assignment. IDs can be compared using cpl\_id\_cmp() in C or using the standard overloaded comparison operators in C++. The variable CPL\_NONE represents the null ID.

cpl\_session\_t: An alias of cpl\_id\_t used to identify sessions. It can be thus manipulated using the same functions and operators as cpl\_id\_t.

cpl\_version\_t: A 32-bit signed version number; only nonnegative values are valid. Macro CPL\_VERSION\_NONE represents an invalid version.

cpl\_return\_t: A 32-bit signed integer, which is returned as a result of every function in the C API. Macro CPL\_OK signifies no error, and macros with the CPL\_E\_ prefix are different error codes. We recommend that the return values are checked using the macro CPL\_IS\_OK(x) rather than comparing them directly to CPL\_OK, just in the case that CPL would be able to use different return codes that do not signify errors in the future.

## Error Codes

|  |  |
| --- | --- |
| Return Code | Description |
| CPL\_OK | No error |
| CPL\_E\_ALREADY\_EXISTS | The requested object, file, or node already exists |
| CPL\_E\_ALREADY\_INITIALIZED | The library has already been initialized, which usually means that you are calling cpl\_attach() more than once |
| CPL\_E\_BACKEND\_INTERNAL\_ERROR | Internal error inside the database backend or the database driver |
| CPL\_E\_DB\_CONNECTION\_ERROR | Failure to connect to the database backend or if the database connection was closed unexpectedly |
| CPL\_E\_INSUFFICIENT\_RESOURCES | Insufficient system resources, especially insufficient memory |
| CPL\_E\_INTERNAL\_ERROR | Internal error inside CPL (ideally, this should never happen) |
| CPL\_E\_INVALID\_ARGUMENT | Invalid argument, such as passing NULL instead of a required string argument, using a negative version number, or using CPL\_NONE instead of a required object ID |
| CPL\_E\_NOT\_FOUND | The requested provenance object, session, or file was not found |
| CPL\_E\_NOT\_IMPLEMENTED | The called function, or one of the functions that it depends on, has not yet been implemented |
| CPL\_E\_NOT\_INITIALIZED | The library has not been initialized; please call cpl\_attach() |
| CPL\_E\_PLATFORM\_ERROR | Internal error inside the platform compatibility layer, signifying that it was not able to resolve an error returned from the native API |
| CPL\_E\_PREPARE\_STATEMENT\_ERROR | If the database supports prepared statements – the compilation of a prepared statement (query) failed |
| CPL\_E\_STATEMENT\_ERROR | The execution of a database update statement or a query failed |

## Attach/Detach

All functions return a value of type cpl\_return\_t: either CPL\_OK on success or an error code.

cpl\_attach: Initialize the CPL bindings for the current process and attach to the database backend.

Arguments:

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

cpl\_detach: Detach from the database backend and perform clean-up.

## Disclosed Provenance API

All functions return a value of type cpl\_return\_t: either CPL\_OK on success or an error code.

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

Arguments:

* const char\* originator: The string that uniquely identifies the application that is creating the object.
* const char\* name: The object name.
* const char\* type: The object type.
* const cpl\_id\_t container: The object ID of the container to which the object belongs, or CPL\_NONE.
* cpl\_id\_t\* out\_id: The pointer to store the ID of the newly created object

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

Arguments:

* const char\* originator: The string that uniquely identifies the application that is creating the object.
* const char\* name: The object name.
* const char\* type: The object type.
* cpl\_id\_t\* out\_id: The pointer to store the ID of the object

cpl\_data\_flow: Disclose a data flow. CPL translates it into a data dependency (the API is defined in terms of data flow rather than data dependency, because we believe that it would be easier to use by our intended audience)

Arguments:

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

cpl\_control: Disclose a control operation. CPL translates it into a control dependency.

Arguments:

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

# CPL Architecture

The two possible choices for CPL architecture 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.

The disadvantage is that CPL running as a library cannot cache any data, but instead, it needs to issue a large number of queries and rely only on the database cache for performance. 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. CPL also needs to issue the cpl\_db\_get\_version query each time provenance is disclosed in order to look up the version numbers of each object involved in the operation.

If CPL operates as a client of a single provenance service, the information required to answer almost all queries that are necessary for disclosing provenance can be cached by the service. The main disadvantages are that running CPL as a service complicates crash recovery and that it significantly increases the number of IPCs in the system.

CPL currently uses the “CPL as a Library” architecture because of its simplicity, but there are pros and cons to both of these approaches, as it is not entirely clear which of the two would perform better. The implementation of the Core Provenance Service, as well as determining which approach is more suitable for the use in production, is a part of our future work.

# Database Backends

CPL is designed to work with multiple databases – both graph databases and relational databases.

## Database Backend API

CPL communicates with the database backend using the interface cpl\_backend\_t described in this section. The individual functions in the interface were designed to be high level enough so that there is enough room for the database driver to implement it efficiently in the database. For example, most tasks can be performed using a single SPARQL or SQL query. At the same time, the actions are designed to be small enough, so that they can be trivially decomposed into smaller actions if necessary.

CPL requires that each individual operation is atomic and durable, and that it preserves the database consistency. We expect to eliminate the durability requirement in the future in order to improve performance. CPL 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, and all functions return a value of type cpl\_return\_t – either CPL\_OK on success or an error code.

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.

## Graph Database Backend

The graph database backend uses the following schema:

**Provenance Object**

type

name

originator

**Version Node**

creation\_time

version

creation\_timee service process dies.nning CPL as a service complicates recovery, especially in the case thatan error returned from the na

version

in-container

previous-version

input-<type code>

**Session**

mac\_address

username

pid

program

initialization\_time

session

We are currently working on the implementation of two different graph database backends, so that we would be able to support the two most commonly used APIs/protocol: Blueprint (for databases like neo4j or Jena) and RDF/SPARQL (4store and Jena with a SPARQL server).

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).

## Relational Database (ODBC) Backend

The ODBC backend uses the following relational schema:

**cpl\_objects**

id (P)

originator

name

type

creation\_time

container\_id

container\_ver

**cpl\_ancestry**

from\_id (P)

from\_version (P)

to\_id (P)

to\_version (P)

type

**cpl\_versions**

id (P)

version (P)

creation\_time

session\_id

**cpl\_sessions**

id (P)

mac\_address

username

pid

program

initialization\_time