

RCP Database Interactions

Version v00-10-00

Marc Paterno
CD/CPD/APS, FNAL

October 9, 2002

Abstract

This document presents the database interface of the **RCP** system. The interface is used by all **RCP** database implementations.

1 Recent Modifications

This is the first version of the most recent release, **RCP** v00-10-00. The major new feature in this release is the addition of untracked (also called transient) parameters. See section 7 for details.

2 Introduction and Terminology

This document specifies the interactions of the RCP system with all RCP database implementations. In the following, we carefully distinguish between RCPs (instances of the class *RCP*); RCPValues (instances of the class *RCPValue*), RCP scripts (text files used to describe an RCP) and RCP database entries (the representations of RCP objects in a specific physical database).

2.1 Glossary

The following list of terms is presented to help clarify terminology.

Parameter set The collection of name/value pairs in an *RCPValue* object.

RCPName A class; it includes a set of four strings (package name, object name, version tag, and database name), used to uniquely identify a parameter set.

RCPID A class; it provides a unique (within a single database piece) identifier for a parameter set.

RCPValue A class; it contains a parameter set, an *RCPID*, and an *RCPName*.

RCP A class; it is an *RCPValue* object, wrapped in a read-only shell. This is form in which parameter sets are presented to users.

RCPHashKey A class; it represents a not-quite-unique identifier, calculated by an *RCPValue*, from the contents of its parameter set. Untracked (transient) entries are not included in the generation of an *RCPHashKey*.

Database piece A group of tables in a database, which contain related *RCPValue* objects. Different database pieces may or may not reside in different physical databases.

Concrete database object An instance of a subclass of *AbsRCPDatabase*. Each concrete database object is the C++ representation of a single database piece.

3 RCP Classes Relevant to the Databases

The database interactions of the RCP package are encapsulated in a single abstract base class, *AbsRCPDatabase*, from which concrete RCP database classes are derived by subclassing. These subclasses are used, through the *AbsRCPDatabase* interface, by the class *RCPDatabaseServices*. The RCP database objects will be constructed by a factory on behalf of the *RCPManager*, which is responsible for managing all interactions with the physical database represented, within a program, by an RCP database object. Users of the RCP system do not interact directly with the database; they interact only with the class *RCPManager*. The purpose of the *RCPDatabaseServices* class is to provide the database services needed by the *RCPManager*, through a simple interface. It exists to factor out the “business rules” shared by all RCP databases. The interface class *AbsRCPDatabase* expresses the functionality required by each database backend. The things manipulated by the *AbsRCPDatabase* interface are *RCPValue* objects. An *RCPValue* object contains the data for a single *RCP* object and an interface to query and modify the contained data. Each RCP database entry is associated with a unique identifier, in the form of an instance of the class *RCPID*. Each distinct database piece is associated with a unique (across the entire experiment) database identifier, which is also contained in the *RCPID*. We note that a copy of a database piece is not distinct from the master from which it was made because the copy can be used only in read-only mode. Write access is available only for the master version. An *RCPHashKey* represents the “hash value” associated with an *RCPValue* object. This value will be used as a non-unique (but as close to unique as can reasonably be achieved) key to find RCP database entries. The *RCPHashKey* associated with an *RCPValue* can be calculated from the *RCPValue* object alone, using only the (name, value) pairs it contains, neglecting any untracked parameters. *RCPHashKey* is currently a typedef for `unsigned long` – a 32-bit cyclic redundancy check.

4 C++ Representation of Database Pieces

In the RCP system, the representation of a database piece is an instance of a subclass of *AbsRCPDatabase* – a *concrete RCP database object*. It is critical to the functioning of the RCP system that the correct association be made between each concrete RCP database object and the physical database, or portion thereof, it represents. It is simplest to explain the requirements through an example. Consider a single Oracle database containing all the RCP database entries for the experiment. The tables in this database are organized into groups.

- One group of tables for officially released RCPs - named “official”.
- Two groups of tables, one for the Higgs physics group and one for the QCD physics group - named “higgs” and “qcd”
- Two groups of tables, one for each of two users, Jack and Jill - named “jack” and “jill”.

Each of these groups of tables is a *database piece*. Each database piece has an associated name (a string), and an associated *RCPDatabaseID*. Some master database is responsible for keeping track of all names and *RCPDatabaseIDs* for database pieces, because these must be unique across the experiment. In any program, we want to make sure that we have only one programmatic representation of each database piece. So, when we create an instance of *OracleRCPDatabase* (a subclass of *AbsRCPDatabase*), we want to connect it with a specific database piece, which we specify by giving the name of the database piece to the constructor, as follows:

```
bool writable = true;
std::string version ("SomeReleaseTagHere");
OracleRCPDatabase jacksDB("jack", writable, version);
```

The RCP system will make use of this constructor to assure that all *RCPManagers* in a single program that want to talk to a specific database piece do so through the same concrete database object, and that this database object is configured correctly.

5 Responsibilities of *RCPDatabaseServices* and *AbsRCPDatabase*

The *RCPDatabaseServices* class exists to provide a common implementation of the functions related specifically to the behavior required of the RCP system. The *AbsRCPDatabase* classe, and its concrete subclasses, are to perform only the database-specific (including the client/server nature) parts of this behavior. To be more specific:

- *RCPDatabaseServices* provides caching of *RCPValue* objects that are extracted from the *AbsRCPDatabase* object. A concrete database class does not have to implement this caching.
- When an *RCPManager* requests an *RCPValue* that matches the contents of a given *RCPValue* (by supplying a complete *RCPName* and a valid *RCPID*), it is the *RCPDatabaseServices* class that determines which of the possible matches, if any, is the correct one. The *AbsRCPDatabase* class is responsible only for (1) telling how many parameter sets match a given hash key, and (2) returning all the parameter sets matching that has key.
- A concrete subclass (or subclasses) of *AbsRCPDatabase* is responsible for providing the client/server nature of the database connection. An *RCPDatabaseServices* object talks only to an instance of a subclass of *AbsRCPDatabase* which exists in the same process. The details of how the client/server implementation is done is up to the implementer of the concrete *AbsRCPDatabase* subclass.

The complete interface for *AbsRCPDatabase* follows.

```

class RCPName;
typedef std::list<RCPValue> RCPValueCollection;
class AbsRCPDatabase {
public:
    AbsRCPDatabase(const std::string& version = std::string());
    virtual ~AbsRCPDatabase() = 0;

    //
    // Testing
    //
    // Return true if the db contains an entry with this ID.
    virtual bool has(const edm::RCPID& id) const = 0;
    // Return the number of db entries matching this name. Note
    // that the name may be incomplete; this is why more than
    // one match is possible.
    virtual size_t count(const RCPName& name) const = 0;
    // Return the number of db entries matching this hash key.
    virtual size_t count(const RCPHashKey& name) const = 0;
    // Return true if this db may be written to.
    virtual bool isWritable() const = 0;

    //
    // Manipulation and access.
    //
    // Return the name of this db
    virtual DBName dbName() const = 0;
    // Return the class name of this database

```

```

virtual std::string className() const = 0;
// Fill the given RCPValue with the parameter set
// specified by the given RCPID, insert the appropriate
// RCPID and RCPName, and return true. If no match is found,
// return false and do not modify val.
virtual bool get(const edm::RCPID& id,
                RCPValue& val) const = 0;
// Fill the collection with RCPValues that match the given
// hash key, and return true. If no matching RCPValues are
// found, return false and do not modify the collection
// values.
virtual bool get(const RCPHashKey& key,
                RCPValueCollection& values) const = 0;
// Fill the collection with RCPValues that match the given
// (possibly incomplete) RCPName. If no matching RCPValues
// are found, return false and do not modify the collection
// values.
virtual bool get(const RCPName& name,
                RCPValueCollection& values) const = 0;

// Add a parameter set equal to the one within val to the db.
// Modify val to have a valid RCPID (issued by the db, to
// assure uniqueness) and to have a complete RCPName, and
// return true. The RCPName may have to be completed by the
// db, by inserting the database name, again to assure
// uniqueness. If the new parameter set cannot be added to
// the database, or if a new unique RCPID cannot be issued,
// or if the RCPName cannot be completed uniquely, return
// false and do not modify val.
virtual bool put(RCPValue& val) = 0;

```

```

// Try to add the given name to an already existing parameter
// set. Return false on error, true on success. Successful
// completion of this function means that the database has
// completed the name (if necessary), and given it a
// timestamp, and inserted it into the RCPValue. The database
// has also updated itself so that it will recognize this
// name as associated with this RCPValue.
// Preconditions: it is guaranteed that the RCPValue will
// have a valid RCPID, and that this RCPID will be known to
// the database.
virtual bool addName(RCPValue& val, const RCPName& name) = 0;
// Try to add the given name to an already existing parameter
// set. Return false on error, true on success. Successful
// completion of this function means that the database has
// completed the name (if necessary), and given it a
// timestamp, and inserted it into the RCPValue. The database
// has also updated itself so that it will recognize this
// name as associated with this RCPValue.
bool tryToAddName(RCPValue& val, const RCPName& name);
// Remove the given name from this database. If any
// parameter set is associated with no other name, remove the
// parameter set as well. Note that this is a destructive
// operation, and should never be done after a database has
// been in use; this function is to be use, with care, only
// by database managers.
// Return codes:\
// -2: removal failed; unable to remove parameter set
// -1: removal failed; nothing by this name was found
// 0: removed name
// 1: removed name, and also removed parameter set
virtual int remove(const std::string& pkgName,
                  const std::string& objName,
                  const std::string& version) = 0;

//
// Output
//
// Print output useful for debugging to stream os.
virtual void dump(std::ostream& os) const = 0;

```

```

// Return a new version tag. This will give either the fixed
// version this database has been told to produce, or a
// unique timestamp.
std::string generateVersion() const;
// Return true if this database is to produce a fixed
// version tag, and false if it is to produce timestamps.
bool hasFixedVersionTag() const;
};

```

6 Uniqueness of RCPNames and RCPIDs

The class *RCPName* exists in order to provide a human-friendly method of referring to a particular parameter set. The class *RCPID* exists in order to provide a concise and unique method of referring to a particular parameter set. The mapping from *RCPID* to parameter set is one-to-one: each parameter set has exactly one *RCPID*, and each *RCPID* refers to one parameter set. The mapping of *RCPName* to parameter set is many-to-one: each parameter set can have many names (it must have at least one), but each *RCPName* must refer to a single parameter set. An *RCPName* object has several components, each of which is a string: a *package name*, and *object name*, a *version*, and a *database name*. For parameter sets which are entered into the database through the release mechanism, the *version* means the version name of the experiment's software release. For parameter sets entered by any other mechanism (such as as a the planned web interface), the assignment of the version name will be done by the database. The *database name* is the name associated with the database piece in which the parameter set resides.

7 Untracked Parameters

Untracked parameters are never stored in a database. A parameter set presented for completion to the database may contain untracked parameters (*e.g.* if the parameter set was formed by reading a script), but they are not to be stored. The equality test function (`operator==`) for two *RCPValue* does not consider untracked parameters in its comparison. If a parameter set containing untracked parameters is presented to a database for completion, the untracked parameters must remain untouched by the database upon return of the completed *RCPValue*.