Bookkeeping Implementation

Sebastien Ponce

March 12, 2004

Internal Note

Issue :	1
Revision :	0
Reference :	-
Created :	September 1, 2003
Last modified :	March 12, 2004

Contents

0.1 Introduction

This document describes the implementation of the bookkeeping tools and services in LHCb.

We first present the persistent data model, which describes the tables and columns declared in the database to store the data.

We then deal with the description of the transient data model and the way applications can access to the data. This includes the definition of abstract interfaces for both data access and data writing.

Finally we present the python and web tools deployed around the main system for easy access by the end user. We also present some tools that were used to transfer, create or update the main database.

Before jumping into the details, here are some prerequisites :

- The core system was implemented in Java. Thus the interfaces described in the "Data Management" document were adapted to this language. Details are given in section **??**.
- Since Gaudi is only available in C++ at this time, the system has no link to it. In particular, the interfaces do not inherit from the *Interface* abstract interface and the services do not inherit from the IService abstract interface. However, the needed Gaudi components were recreated in Java.
- The database used in the implementation is not precisely defined in the rest of this chapter. This is because it may be any relational database having a JDBC interface. MySQL and Oracle are the two databases that were actually tested and the final system is running mainly under Oracle.

Chapter 1

Data description and access

1.1 Persistent Data

1.1.1 The Data Model

The persistent data model (Figure 1.1) describes the design of the database used to store book-keeping data. This database is supposed to be relational. Thus the description is the description of the tables used and of their colums. Each bubble of the figure is a table and the list of column is given inside each one. The associations drawn as lines show where IDs are supposed to match. As an example, the Job_ID in _Job and in _JobOptions match. The dotted line indicates that on the FilesParam side, the EvtType_ID is actually stored in the Value column when the Name column contains "EventtType" and not as a separate column.

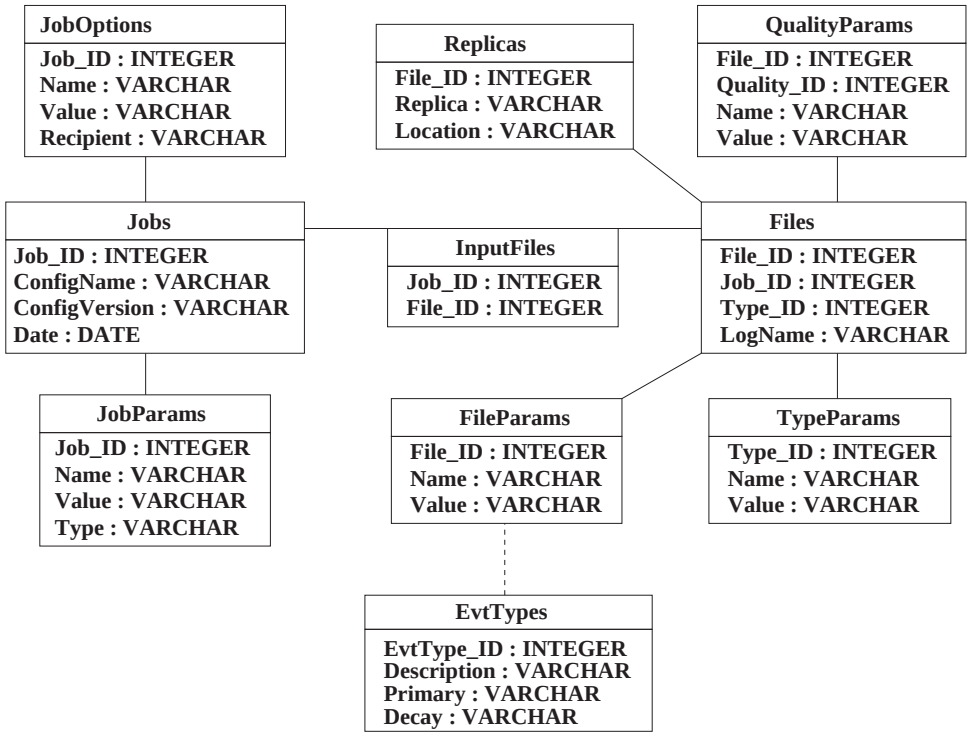

Figure 1.1: The design of the database tables

The whole design was intended to build thin tables on the database side by removing all columns that would not be absolutely necessary for all cases. Lines in one of the parameter tables replace these columns. As an example, the number of event of a given file is only relevant in the case of a file containing data (not in the case of a log file). Thus, is was not added in the _Files table but will be a line in the _FileParams table with name "NbEvent" and value the actual number of event of the associated file.

This architecture also allows taking advantage of the indexing capabilities of the underlying database to search for objects. As an example, indexes should be created in the _InputFile table, based on the file_ID and in the _Files table based on the Job_ID. This allows an efficient browse of the hierarchy of jobs and files.

In principle, every object in the data model maps to a table and every relation between objects to an ID. There are however some exceptions:

- The Type and Quality objects have actually no members. These are stored as parameters. The names of these parameters are "Name" and "Version" for Type and "Group" and "Flag" for Quality.

- Types are not mapped to a table since this table would have had no column except the ID one. They only exist through IDs in other tables. As an example, the Type_ID is used in the _TypeParams and in the _Files tables.

- Qualities are not mapped to a table. However, the table would have had two columns: one for File_ID and one for Quality_ID. For optimization purposes, it was decided that the File_ID would be replicated in every line of the _QualityParams table and that the _Qualities table would be suppressed. However, all the quality parameters defined for a given file stay grouped by the Quality_ID and will map to parameters of the same Quality object in the transient world.

- The many to many link that exists for input files requires a specific link table: _InputFiles. Each line of this table describes one link between a job and a file, which is seen as an input file of this job. Note that the output files relation is a regular relation and is thus mapped to an ID, namely the Job_ID in _Files.

1.1.2 Constraints

The "Type" column of table "JobParams" cannot be filled with any value. Table 2 give a list of the allowed values and their meaning.

Value	Meaning
"Info"	This parameter gives information on the conditions under which the job ran
"Script"	This is a script that was launched before the job was run.
"Environment variable"	This is an environment variable that was defined before the job was run.
"Error"	This is a parameter created by the book process when it crashes. It stores what could not be put anywhere else du to the crash.

Many parameter tables are defined in this design that may keep most of the relevant information. It is thus important to agree on the name of the parameters to be able to retrieve the information. Here are the parameters and name that are defined by default. More parameters may be defined in the future. Note that in the case of the "JobParams" table, these are "Info" parameters.

Table	Name	Description	Needed
JobParams	Location	The laboratory where the job was executed	No
	Host	The machine where the job was executed	No
	Run	The run number for data files that have one	No
	Seed1	First random number seed used for this job, if any	No
	Seed2	Second random number seed used for this job, if any	No
	XmlDDDBVersion	Version of the geometry database used for this job, if the XML database is used	No
	DbaseVersion	Version of the geometry database used for this job, if the old dbase is used	No
	Name	The job name. By convention XXXXXXXX_YYYYYYYY_Z where XXXXXXXX is the production number, with 8 digit and prefixed with 0s, YYYYYYYY is the job number (same convention) and Z is the step number	No
	Production	The production number	No
	Job	The job number in the production	No
	ProgramName	The name of the program that was run	No
	ProgramVersion	The version of the program that was run	No
FileParams	EvtType	The type of events contained in this file (Only in case of a data file). Only an id is given here. The description of the event type can be found in the EvtType table	No
	EvtStat	The number of events in this file (Only in case of a data file)	No
	Size	The size of the file in KB	No
TypeParams	Name	The name of this type. Example: "RAW", "DST" or "Log"	Yes
	Version	The version of this type. Usually a number.	Yes
	Description	A short description of this type.	No
QualityParams	Group	The group concerned by this Quality.	Yes
	Flag	The quality by itself. Allowed values are "Good", "Fair", "Bad" and "Not checked".	Yes

1.1.3 How to generate unique IDs

As you can see in the tables of Figure 1.1, many IDs are defined in the database schema. Each Job, File, Type and Quality can thus be identified by a single and unique number. The problem is to ensure the uniqueness of this number.

Since every database has its own way of solving this problem (if any), it was decided to do it by hand via the use of an additional table called "NextBookkeepingIDs". This table has only two columns: "TableName" and "ID". It contains only four lines (with TableNames "Job", "File", "Type" and "Quality"), which keep track of the next ID that should be used when creating a Job, a File, a Type or a Quality. The IBookkeepingEditor interface implementations have to take care that this table is maintained up to date. Note that the table contains a fifth line called "IndexName". This index is used for the bookkeeping updates, see section **??**.

1.2 The transient Data Model

A preliminary version of the transient data model was exposed in the document "Data Management". We describe here its current implementation. Lsiting 1.1 shows the implementation of every class used in this model. The code should be pretty obvious to understand.

```
public class Job {
    public Job (int jobID, String configName,
                String configVersion, Date date);
    public int jobID();
    public String configName();
    public String configVersion();
    public Date date();
}
public class File {
    public File (int fileID, int jobID,
                 int typeID, String logName);
    public int fileID();
    public int jobID();
    public int typeID();
    public String logName();
}
public class Type {
    public Type (int typeID, String name, String version);
    public int typeID();
    public String name();
    public String version();
}
public class Quality {
    public Quality (int fileID, int qualityID,
                    String group, String flag);
    public int qualityID();
    public int fileID();
    public String group();
    public String flag();
}
public class Param {
    public Param (int ID, String name, String value);
    public int ID();
    public String name();
    public String value();
}
public class JobOption extends Param {
    public JobOption (int jobID, String name,
                      String recipient, String value);
```

```
    public String recipient();
}
public class TypedParam extends Param {
    public TypedParam (int jobID, String name,
                       String value, String type);
    public String type();
}
public class EvtType {
    public EvtType (int id, String desc,
                    String prim, String decay);
    public int ID();
    public String description();
    public String primary();
    public String decay();
}
```

Listing 1.1: Transient Data Model

1.3 Access to data from Java

The interfaces are actually divided in two groups : the ones providing read access to the data and the ones providing write access. The first ones have their name ending with “Info” while the second ones have their name ending with “Editor”.

The data were split into three pieces : the event types, the replica catalog and the rest of the bookkeeping. Thus, there are 6 interfaces : IEvtTypeInfo, IReplicaInfo, IBookkeepingInfo, IEvtTypeEditor, IReplicaEditor and IBookkeepingEditor.

On top of the interface themselves, two helper classes were defined in the (tiny) Java part of Gaudi : StatusCode and Handle. They are the only objects returned by the methods of the interfaces.

1.3.1 StatusCode and Handle

The StatusCode class has the same interest in Java that it had in the C++ world: it encapsulates a status and allow to query whether the operation is success or not. The interesting part of its Java definition is shown on Listing 1.2.

```
public class StatusCode extends Object   {
  static public StatusCode FAILURE = new StatusCode(0);
  static public StatusCode SUCCESS = new StatusCode(1);
  public boolean isSuccess();
  public boolean isFailure();
};

public class Handle extends StatusCode {
    static public Handle FAILURE = new Handle(0, null);
    static public Handle SUCCESS = new Handle(1, null);
    public Object object;
};
```

Listing 1.2: Definition of class StatusCode and Handle

Besides StatusCode, a new return type has been defined that extends it by adding a member of class Object. This allows a function to return at the same time the status and the return value. This new type was called Handle. The Handle class is heavily used in the IBookkeepingInfo and IBookkeepingEditor interfaces.

1.3.2 Main Bookkeeping data

Listings 1.3 and 1.4 gives the definition of the IBookkeepingInfo and IBookkeepingEditor interfaces in Java. Some functions are returning a Handle that encapsulates the actual return value. The return value is either a single object (see type definitions in section 1) or a java.util.Vector object containing such objects.

```
public interface IBookkeepingInfo {
    public Handle jobs ();
    public Handle jobs (Date date1, Date date2);
    public Handle jobs (String configName, String configVersion);
    public Handle jobs (File inpuFile);
    public Handle job (File outputFile);
    public Handle job (int id);
    public Handle files ();
    public Handle files (String SQLQuery);
    public Handle file (String logName);
    public Handle inputFiles (Job job);
    public Handle outputFiles (Job job);
    public Handle files (Type type);
    public Handle qualities (File file);
    public Handle quality (File file, String group, String flag);
    public Handle types ();
    public Handle type (String Name, String version);
    public Handle type (File file);
    public Handle jobParameters (Job job);
    public Handle jobParameters (Job job, String type);
    public Handle jobParameter (Job job, String name);
    public Handle jobOptions (Job job);
    public Handle jobOption (Job job, String name, String recipient);
    public Handle qualityParameters (Quality quality);
    public Handle qualityParameter (Quality quality, String name);
    public Handle fileParameters (File file);
    public Handle fileParameter (File file, String name);
    public Handle typeParameters (Type type);
    public Handle typeParameter (Type type, String name);
}
```

Listing 1.3: Definition of the IBookkeepingInfo interface

```
public interface IBookkeepingEditor extends IBookkeepingInfo {
  public Handle createJob (String configName, String configVersion,
                           Date date);
  public StatusCode modifyJobConfigName (String configName, Job job);
  public StatusCode modifyJobConfigVersion (String configVersion, Job job);
  public StatusCode modifyJobDate (Date date, Job job);
  public StatusCode deleteJob (Job job);
  public Handle createFile (String logName, Job origin, Type type);
  public StatusCode modifyFileLogName (String logName, File file);
  public StatusCode deleteFile (File file);
  public Handle createQuality (File file, String group, String flag);
  public StatusCode deleteQuality (Quality quality);
  public Handle createType (String name, String version);
  public StatusCode deleteType (Type type);
  public StatusCode createJobParameter (Job job, String name,
                                        String value, String type);
  public StatusCode modifyJobParameterValue (Job job, String name,
                                             String value);
  public StatusCode modifyJobParameterType (Job job, String name,
                                            String type);
  public StatusCode deleteJobParameter (Job job, String name);
```

```
  public StatusCode createJobOption (Job job, String name,
                                     String recipient, String value);
  public StatusCode modifyJobOptionValue (Job job, String name,
                                      String recipient, String value);
  public StatusCode deleteJobOption (Job job, String name, String recipient);
  public StatusCode createFileParameter (File file, String name,
                                     String value);
  public StatusCode modifyFileParameterValue (File file, String name,
                                       String value);
  public StatusCode deleteFileParameter (File file, String name);
  public StatusCode createQualityParameter (Quality quality, String name,
                                      String value);
  public StatusCode modifyQualityParameterValue (Quality quality,
                                        String name,
                                        String value);
  public StatusCode deleteQualityParameter (Quality quality, String name);
  public StatusCode createTypeParameter (Type type, String name,
                                     String value);
  public StatusCode modifyTypeParameterValue (Type type, String name,
                                       String value);
  public StatusCode deleteTypeParameter (Type type, String name);
  public StatusCode addInputFile (File file, Job job);
  public StatusCode removeInputFile (File file, Job job);
}
```

Listing 1.4: Definition of the IBookkeepingEditor interface

Note that the two files methods should never be used except for debug purposes. As a matter of fact, the first one would output the full list of available files, which would kill the server for lack of memory on the production system. The second method would execute any SQL query which would allow a user to delete or corrupt data.

The methods are only there for internal debug purposes and should never be exposed to the end user.

1.3.3 Event types

The IEvtTypeInfo and IEvtTypeEditor abstract interfaces are described in listings 1.5 and 1.6

```
public interface IEvtTypeInfo {
    public Handle evtTypes ();
    public Handle evtType (int id);
}
```

Listing 1.5: Definition of the IEvtTypeInfo interface

```
public interface IEvtTypeEditor extends IEvtTypeInfo {
    public StatusCode addEvtType (int id,
                                  String description,
                                  String primary,
                                  String decay);
    public StatusCode removeEvtType (int id);
}
```

Listing 1.6: Definition of the IEvtTypeEditor interface

1.3.4 Replicas

The IReplicaInfo and IReplicaEditor abstract interfaces are described in listings 1.7 and 1.8

```
public interface IReplicaInfo {
    public Handle replica (File file, String location);
    public Handle replicas (File file);
}
```

Listing 1.7: Definition of the IReplicaInfo interface

```
public interface IReplicaEditor extends IReplicaInfo {
    public StatusCode addReplica (File file, String replica,
                                  String location);
    public StatusCode removeReplica (File file, String location);
}
```

Listing 1.8: Definition of the IReplicaEditor interface

1.3.5 A word on the default implementation

The BookkeepingSvc class is a service that implements all Editor interfaces and as a consequence all Info interfaces. The details of the implementation won't be described here but some facts need to be known :

- All connections to the database are using JDBC. The only requirement is to give correct connection string, user name and password when creating the service. An example of an initialization string is : "jdbc:oracle:thin:@oradev:10521:D"
- Every query done on the bookkeeping database corresponds to a SQL statement. The whole list of statements used was put at the very beginning of the Java file.
- The rest of the code is made of very simple calls to JDBC.

1.4 Access to data from python and XMLRPC

All interfaces presented in section **??** are available in python through XMLRPC. The implementation of the XMLRPC server is described in details in section 2.1 of chapter 2. We describe here the interface available in python, the python objects that are defined and the way the interface should be used.

The python interface is very close to the Java interface described in section 1.3 :

- Each java object has a python counterpart which is a dictionnary having the members of the Java object as entries.
- Java vectors are converted to python lists.
- Each java method has a python counterpart with the same name and same arguments.

1.4.1 Python objects

Here is the list of the python objects used by the interface together with the members of each of them and their type :

job

- jobID : number

- configName : string
- configVersion : string
- date : python time tuple, as returned by the gmttime function

file

- fileID : number
- jobID : number
- typeID : number
- logName : string

type

- typeID : number
- name : string
- version : string

quality

- qualityID : number
- fileID : number
- group : string
- flag : string

parameter

- ID : number
- name : string
- value : string

typedparameter

- ID : number
- name : string
- value : string
- type : string

joboption

- ID : number
- name : string
- value : string
- recipient : string

evttype

- ID : number
- description : string
- primary : string
- decay : string

Take care that these are actually not objects but dictionnaries and the "members" are the entries in the dictionnary.

1.4.2 Python interface

There is a little complication in the definition of the python interface. It lies in the fact that python does not include the signature of a method in its definition. In other words, you cannot define two functions with the same name and differents arguments (even different numbers of them) in python.

The solution is to define a single function in python for all java functions with the same name which checks the number of arguments given and their types to choose the right java function. This was done in the actual interface implementation, in file python/BookkeepingSvc.py of the Bookkeeping package. However, we will show here a pseudo interface where different functions may have the same name.

Listing 1.9 gives the full list of methods available in python with the type of the arguments and return type. Note that the given return type is the one that is returned in case of success. If the call fails, a string is returned containing the error message. Thus the returned type should always be checked to detect errors.

```
list<job>          jobs(file f)
list<job>          jobs(string configName, string configVersion)
list<job>          jobs(time date1, time date2)
job                job(file outputFile)
job                job(number id)
list<file>         files(string SQLquery)
list<file>         files(type t)
file               file(string logname)
list<file>         inputFiles(job j)
list<file>         outputFiles(job j)
list<quality>      qualities(file f)
quality            quality(file f, string group, string flag)
list<type>         types()
type               type(file f)
type               type(string name, string version)
list<typedparameter> jobParameters(job j)
list<typedparameter> jobParameters(job j, string type)
typedparameter     jobParameter(job j, string name)
list<joboption>    jobOptions(job j)
joboption          jobOption(job j, string name, string recipient)
list<parameter>    qualityParameters(quality q)
parameter          qualityParameter(quality q, string name)
list<parameter>    fileParameters(file f)
parameter          fileParameter(file f, string name)
list<parameter>    typeParameters(type t)
parameter          typeParameter(type t, string name)
dictionnary<string location, string name> replicas(file f)
string             replica(file f, string location)
list<evttype>      evtTypes()
evttype            evtType(number id)
```

Listing 1.9: python interface to the Bookkeeping service

1.4.3 XMLRPC interface usage

The usage of the XMLRPC interface is standard. One just needs the name of the service (RPC/-BookkeepingSvc) and the machine and port where to find it. Listing 1.10 shows some code displaying the replicas of a given file.

```
from xmlrpclib import Server
server = Server('http://lbnts3.cern.ch:8100/RPC/BookkeepingSvc')
f = server.file(''00000141_00000001_6.oodst'')
```

```
print server.replicas(f)
```

Listing 1.10: XMLRPC code sample

Chapter 2

Server infrastructure

The bookkeeping services are hosted by a central server that deals both with web pages and XMLRPC services. We describe here the server itself as well as its deployment as the time this document was written.

2.1 Server implementation

The Bookkeeping server is based on a piece of python code providing both an XMLRPC server and a web server. This code was initially used in the Production system and was reused with no change in the Bookkeeping. It is implemented in file python/gaudiweb.py of the Bookkeeping package. The server is then customized in file python/BookkeepingServer.py of the Bookkeeping package.

The server handles essentially three things : webpages, servlets and XMLRPC services. The webpages and XMLRPC services just need to be registered to become available, as you can see in methods *startBookkeepingWeb* and *startRPCServices*. Most of the functionnalities is actually provided by the underlying gaudiweb server.

For the servlets, the situation is a bit more complex since this concept is java specific and thus not available in python. For this purpose, the Bookkeeping server can actually not run using a regular python implementation. It actually needs Jython, a java implementation of python. On top of that, dedicated classes were defined to handle servlets correctly, reusing the gaudiweb.Service mechanism.

This servlet support is 70% of the code of BookkeepingServer.py. It uses some home made, dummy implementation of a servlet engine provided in the calsses DummyServletConfig, DummyServletResponse and DummyServletRequest. However ever, the servelt engine could be droped as soon as the servlets are hosted by an external web server, like apache or the ORACLE one.

To conclude with the Bookkeeping server, here is the list of entry points in the current server as well as their type and usage.

- Main : A servlet building the main bookkeeping web page, i.e. the one for selecting data. See section **??**.
- EvtTypes : A servlet building a list of existing event types. The types for which no events can be found are not listed. See section **??**.
- EvTypeInfo : A servlet handling the selection of data and building a web page with the results See section **??**.
- DataSets : ? Francoise, could you put something here ?

- Select A Servlet allowing to browse the database. See section **??**.
- NewConfirm : A Servlet to enter data into the database. See section **??**.
- DisplayBookFile : A servlet that displays details of database modification requests. See section **??**.
- Manager : A python servlet allowing to shutdown the server from a web interface. See section **??**.
- Bookkeeping : A directory containing files used by the generated html pages. Essentially javascripts files.
- images : A directory containing images used by the generated html pages.
- RPC/BookkeepingSvc : An XMLRPC service providing the Bookkeeping API. See section 1.4.
- RPC/BookkeepingQuery : An XMLRPC service providing the BookkeepingQuery API. This API allows to select files from the Bookkeeping. See section **??**.
- RPC/GetLogFiles : An XMLRPC service providing easy access to log files. See section **??**.

2.2 Server deployment

To be filled by Joel... Merci Joel !

Chapter 3

Booking of data

Chapter 4

The end-user web interfaces