The ALICE Offline Bible

Version 0.00

(Rev. 22)

1 Table of content

2 Introduction

Purpose of this document.

2.1 About this document

History of the document, origin of the different parts and authors.

2.2 Acknowledgements

All those who helped

2.3 History of modifications

#	Who	When	What
1	F. Carminati	30/1/07	Initial merge of documents
2	F. Carminati	19/3/07	Inserted MetaData note
3	G. Bruckner	1/7/07	

3 The AliRoot primer

3.1 About this primer

The aim of this primer is to give some basic information about the ALICE offline framework (AliRoot) from the users' perspective. We explain in detail the installation procedure, and give examples of some typical use cases: detector description, event generation, particle transport, generation of "summable digits", event merging, reconstruction, particle identification, and generation of event summary data.

The primer also includes some examples of analysis, and short description of the existing analysis classes in AliRoot. An updated version of the document can be downloaded from:

http://aliceinfo.cern.ch/Offline/AliRoot/Manual.html

For the reader interested by the AliRoot architecture and by the performance studies done so far, a good starting point is Chapter 4 of the ALICE Physics Performance Report [1]. Another important document is the ALICE Computing Technical Design Report [2].

Some information contained there has been included in the present document, but most of the details have been omitted.

AliRoot uses the ROOT [3] system as a foundation on which the framework for simulation, reconstruction and analysis is built. The Geant3 [4] or FLUKA [5] packages perform the transport of particles through the detector and simulate the energy deposition from which the detector response can be simulated. Support for Geant4 [6] transport package is coming soon.

Except for large existing libraries, such as Pythia6 [7] and HIJING [8], and some remaining legacy code, this framework is based on the Object Oriented programming paradigm, and is written in C++.

The following packages are needed to install the fully operational software distribution:

- ROOT, available from http://root.cern.ch or using the ROOT SVN repository: http://root.cern.ch/svn/root/
- AliRoot from the ALICE offline SVN repository:

  https://alisoft.cern.ch/AliRoot/
- transport packages:
  - GEANT 3 is available from the ROOT SVN repository
  - FLUKA library can be obtained after registration from http://www.fluka.org

- GEANT~4 distribution from http://cern.ch/geant4.

The access to the GRID resources and data is provided by the AliEn [9] system.

The installation details are explained in Section 3.3.

3.2 AliRoot framework

In HEP, a framework is a set of software tools that enables data processing. For example the old CERN Program Library was a toolkit to build a framework. PAW was the first example of integration of tools into a coherent ensemble, specifically dedicated to data analysis. The role of the framework is shown in Figure 1.

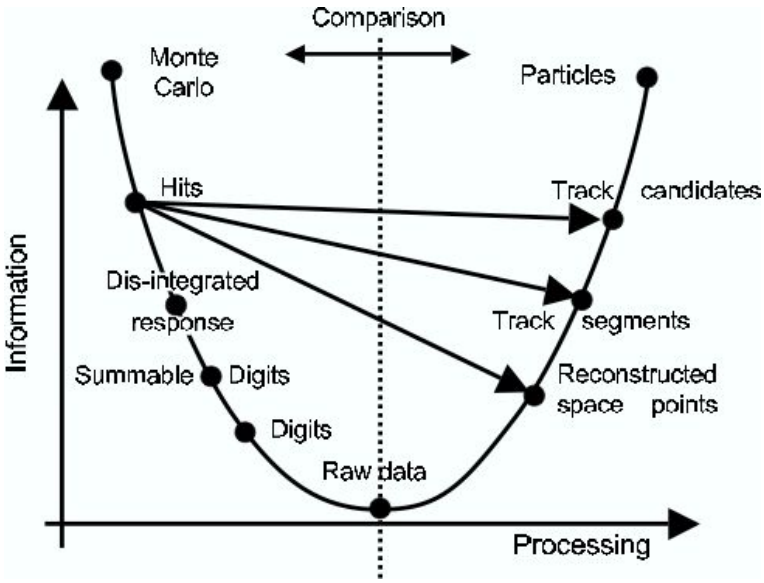


Figure 1: Data processing framework

The primary interactions are simulated via event generators, and the resulting kinematic tree is then used in the transport package. An event generator produces set of "particles" with their momenta. The set of particles, where one maintains the production history (in form of mother-daughter relationship and production vertex), forms the kinematic tree. More details can be found in the ROOT documentation of class **TParticle**. The transport package transports the particles through the set of detectors, and produces **hits**, which in ALICE terminology means energy deposition at a given point. The hits contain also information ("track labels") about the particles that have generated them. In case of calorimeters (PHOS and EMCAL) the hit is the energy deposition in the whole active volume of a detecting element. In some detectors the energy of the hit is used only for comparison with a given threshold, for example in TOF and ITS pixel layers.

At the next step, the detector response is taken into account, and the hits are transformed into **digits**. As mentioned previously, the hits are closely related to the tracks that generated them. The transition from hits/tracks to digits/detectors is marked on the picture as "disintegrated response", the tracks are "disintegrated" and only the labels carry the Monte Carlo information.

There are two types of digits: “summable digits”, where one uses low thresholds and the result is additive, and “digits”, where the real thresholds are used, and the result is similar to what one would get in the real data taking. In some sense the “summable digits” are precursors of the “digits”. The noise simulation is activated when “digits” are produced. There are two differences between “digits” and the “raw” data format produced by the detector: firstly, the information about the Monte Carlo particle generating the digit is kept as data member of the class **AliDigit**, and secondly, the raw data are stored in binary format as “payload” in a ROOT structure, while the digits are stored in ROOT classes. Two conversion chains are provided in AliRoot:

hits → summable digits → digits

hits → digits

The summable digits are used for the so-called “event merging”, where a signal event is embedded in a signal-free underlying event. This technique is widely used in heavy-ion physics and allows reusing the underlying events with substantial economy of computing resources. Optionally, it is possible to perform the conversion

digits → raw data

which is used to estimate the expected data size, to evaluate the high level trigger algorithms and to carry on the so called computing data challenges. The reconstruction and the HLT algorithms can both work with digits and with raw data. There is also the possibility to convert the raw data between the following formats: the format coming from the front-end electronics (FEE) through the detector data link (DDL), the format used in the data acquisition system (DAQ) and the “rootified” format. More details are given in section 3.5.

After the creation of digits, the reconstruction and analysis chains can be activated to evaluate the software and the detector performance, and to study some particular signatures. The reconstruction takes as input digits or raw data, real or simulated.

The user can intervene into the cycle provided by the framework and replace any part of it with his own code or implement his own analysis of the data. I/O and user interfaces are part of the framework, as are data visualization and analysis tools and all procedures that are considered of general interest to be introduced into the framework. The scope of the framework evolves with time as do the needs of the physics community.

The basic principles that have guided the design of the AliRoot framework are reusability and modularity. There are almost as many definitions of these concepts as there are programmers. However, for our purpose, we adopt an operative heuristic definition that expresses our objective to minimize the amount of unused or rewritten code and maximize the participation of the physicists in the development of the code.

Modularity allows replacement of parts of our system with minimal or no impact on the rest. We do not expect to replace every part of our system. Therefore we focus on modularity directed at those elements that we expect to change. For example, we require the ability to change the event generator or the transport Monte Carlo without affecting the user code. There are elements that we do not plan to subject to major modifications, but rather to evolve them in collaboration with their authors such as the

ROOT I/O subsystem or the ROOT User Interface (UI). Whenever an element is chosen to become a modular one, we define an abstract interface to it. Code contributions from different detectors are independent so that different detector groups can work concurrently on the system while minimizing the interference. We understand and accept the risk that, at some point, the need may arise to modularize a component that was not initially designed to modular. For these cases, we have elaborated a development strategy that can handle design changes in production code.

Reusability is the protection of the investment made by the physicist programmers of ALICE. The code embodies a large amount of scientific knowledge and experience and thus is a precious resource. We preserve this investment by designing a modular system in the sense above and by making sure that we maintain the maximum amount of backward compatibility while evolving our system. This naturally generates requirements on the underlying framework prompting developments such as the introduction of automatic schema evolution in ROOT.

Support of the AliRoot framework is a collaborative effort within the ALICE experiment. Questions, suggestions, topics for discussion and messages are exchanged in the mailing list alice-off@cern.ch. Bug reports and tasks are submitted on the Savannah page http://savannah.cern.ch/projects/aliroot.

3.3 Installation and development tools

3.3.1 Platforms and compilers

The main development and production platform is Linux on Intel 32 bit processors. The official Linux [10] distribution at CERN is Scientific Linux SLC [11]. The code works also on RedHat [12] version 7.3, 8.0, 9.0, Fedora Core [13] 1 – 5, and on many other Linux distributions. The main compiler on Linux is gcc [14]: the recommended version is gcc 3.2.3 – 3.4.6. Older releases (2.91.66, 2.95.2, 2.96) have problems in the FORTRAN optimization that has to be switched off for all FORTRAN packages. AliRoot can be used with gcc 4.0.X where the FORTRAN compiler g77 is replaced by g95. The last release series of gcc (4.1) work with gfortran as well. As an option you can use Intel icc [15] compiler, which is also supported. You can download it from http://www.intel.com and use it free of charge for non-commercial projects. Intel also provides free of charge the VTune [16] profiling tool which is one of the best available so far.

AliRoot is supported on Intel 64 bit processors [17] running Linux. Both the gcc and Intel icc compilers can be used.

On 64 bit AMD [18] processors, such as Opteron, AliRoot runs successfully with the gcc compiler.

The software is also regularly compiled and run on other Unix platforms. On Sun (SunOS 5.8) we recommend the CC compiler Sun WorkShop 6 update 1 C++ 5.2. The WorkShop integrates nice debugging and profiling facilities that are very useful for code development.

On Compaq alpha server (Digital Unix V4.0) the default compiler is cxx (Compaq C++

V6.2-024 for Digital UNIX V4.0F). Alpha also provides its profiling tool pixie, which works well with shared libraries. AliRoot works also on alpha server running Linux, where the compiler is gcc.

Recently AliRoot was ported to MacOS (Darwin). This OS is very sensitive to the circular dependences in the shared libraries, which makes it very useful as test platform.

3.3.2 Essential SVN information

SVN stadns for Subversion - it is a version control system that enables a group of people to work together on a set of files (for instance program sources). It also records the history of files, which allows backtracking and file versioning. Subversion succeded Concurrent Version Sysytem, and therefore is mostly-compatible to CVS.

The official SVN Web page is http://subversion.tigris.org/ .

SVN has a host of features, among them the most important are:

- SVN facilitates parallel and concurrent code development;
- it provides easy support and simple access;

SVN has rich set of commands, the most important are described below. There exist several tools for visualization, logging and control that work with SVN. More information is available in the SVN documentation and manual [19].

Usually the development process with SVN has the following features:

- All developers work on their *own* copy of the project (in one of their directories);
- They often have to *synchronize* with a global repository both to update with modifications from other people and to commit their own changes;
- In case of conflicts, it is the developer's responsibility to resolve the situation, because the SVN tool can only perform a purely mechanical merge.

Instructions of using Aliroot Subversion can be found on: http://aliceinfo.cern.ch/Offline/AliRoot/HowTO_SVN.html

3.3.3 Main SVN commands

- svn checkout — Check out a working copy from a repository.

```
% svn co https://alisoft.cern.ch/AliRoot/trunk AliRoot
```

- svn update — Update your working copy. This command should be called from inside the working directory to update it. The first character in line for each updated item, stands for the action taken for this item. The character have the following meaning:

  A - added

  D - deleted

U - updated

C - conflict

G - merged

```
% svn update
```

- svn diff — Display the differences between two paths.

```
% svn diff -r 20943 Makefile
```

- svn add - Add files, directories, or symbolic links.

```
% svn -qz9 add AliTPCseed.*
```

- svn delete - Delete an item from a working copy or repository.

```
% svn delete -f CASTOR
```

- svn commit checks in the local modifications to the repository, send changes of the working copy and increments the version numbers of the files. In the example below all the changes made in the different files of the module STEER will be committed to the repository. The -m option is followed by the log message. In case you don't provide it an editor window will prompt you to enter your description. No commit is possible without the log message that explains what was done.

```
% svn ci newname.cxx
```

- svn status — Print the status of working copy files and directories.With --show-updates, add working revision and server out-of-date information. With --verbose, print full revision information on every item.

```
% svn status Makefile
```

-

3.3.4 Environment variables

Before the installation of AliRoot, the user has to set certain environment variables. In the following examples, we assume that the user is working on Linux and the default shell is bash. It is sufficient to add to ~/.bash_profile the lines shown below:

```
# ROOT
export ROOTSYS=<ROOT installation directory>
export PATH=$PATH\:$ROOTSYS/bin
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH\:$ROOTSYS/lib

# AliRoot
export ALICE=<AliRoot installation directory>
export ALICE_ROOT=$ALICE/AliRoot
export ALICE_TARGET=`root-config --arch`
export PATH=$PATH\:$ALICE_ROOT/bin/tgt_${ALICE_TARGET}
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH\:$ALICE_ROOT/lib/tgt_$
{ALICE_TARGET}
```

```
# Geant3
export PLATFORM=`root-config --arch`
# Optional, defined otherwise in Geant3 Makefile

export  LD_LIBRARY_PATH=$LD_LIBRARY_PATH\:$ALICE/geant3/lib/tgt_$
{ALICE_TARGET}

# FLUKA
export FLUPRO=$ALICE/fluka # $FLUPRO is used in TFluka
export PATH=$PATH\:$FLUPRO/flutil

# Geant4: see the details later
```

The meaning of the environment variables is the following:

ROOTSYS directory of the ROOT package;

ALICE top directory for all software packages used in ALICE;

ALICE_ROOT directory where the AliRoot package is located, usually a subdirectory of ALICE;

ALICE_TARGET specific platform name. Up to release v4-01-Release this variable was set to the result of “uname” command. Starting from AliRoot v4-02-05 the ROOT naming schema was adopted, and the user has to use the “root-config --arch” command in order to know the proper value.

PLATFORM same as ALICE_TARGET for the GEANT 3 package. Until GEANT 3 v1-0, the user had to use `uname` to specify the platform. From version v1-0 on, the ROOT platform is used instead ("root-config --arch"). This environment variable is set by default in the Geant 3 Makefile.

3.4Software packages

3.4.1 AliEn

The installation of AliEn is the first one to be done if you plan to access the GRID or need GRID-enabled ROOT. You can download the AliEn installer and use it in the following way:

Get the installer:

```
% wget http://alien.cern.ch/alien-installer
% chmod +x alien-installer
% ./alien-installer
```

The alien-installer runs a dialog that prompts for the default selection and options. The default installation place for AliEn is /opt/alien, and the typical packages one has to

install are “client” and “gshell”.

3.4.2 ROOT

All ALICE offline software is based on ROOT [197]. The ROOT framework offers a number of important elements that are exploited in AliRoot:

- a complete data analysis framework including all the PAW features;
- an advanced Graphic User Interface (GUI) toolkit;
- a large set of utility functions, including several commonly used mathematical functions, random number generators, multi-parametric fit and minimization procedures;
- a complete set of object containers;
- integrated I/O with class schema evolution;
- C++ as a scripting language;
- documentation tools.

There is a useful ROOT user's guide that incorporates important and detailed information. For those who are not familiar with ROOT, a good starting point is the ROOT Web page at http://root.cern.ch. Here, the experienced user may find easily the latest version of the class descriptions and search for useful information.

The recommended way to install ROOT is from the SVN sources, as it is shown below:

1. Get a specific version (>= 2.25/00), e.g.: version 2.25/03:

```
prompt% svn co http://root.cern.ch/svn/root/tags/v2-25-03 root
```

2. Alternatively, checkout the head (development version) of the sources:

```
prompt% svn co http://root.cern.ch/svn/root/trunk root
```

In both cases you should have a subdirectory called "root" in the directory you ran the above commands in.

The appropriate combinations of ROOT, Geant 3 and AliRoot versions can be found at http://aliceinfo.cern.ch/Offline/AliRoot/Releases.html

The code is stored in the directory “root”. You have to go there, set the ROOTSYS environment variable (if this is not done in advance), and configure ROOT. The ROOTSYS contains the full path to the ROOT directory.

ROOT Configuration

```
#!/bin/sh

cd root
export ROOTSYS=`pwd`

ALIEN_ROOT=/opt/alien

./configure \
    --with-pythia6-uscore=SINGLE \
```

```
    --enable-cern --enable-rfio \
    --enable-mathmore --enable-mathcore --enable-roofit \
    --enable-asimage --enable-minuit2 \
    --enable-alien --with-alien-incdir=${ALIEN_ROOT}/api/include \
    --with-alien-libdir=${ALIEN_ROOT}/api/lib
```

3. Now you can compile and test ROOT

Compiling and testing ROOT

```
#!/bin/sh

make
make map
cd test
make
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:.
export PATH=$LD_LIBRARY_PATH:.
./stress
./stressFit
./stressGeometry
./stressGraphics
./stressHepix
./stressLinear
./stressShapes
./stressSpectrum
./stressVector
```

At this point the user should have a working ROOT version on a Linux (32 bit Pentium processor with gcc compiler). The list of supported platforms can be obtained by "./configure --help" command.

3.4.3 GEANT 3

The installation of GEANT~3 is needed, because it currently is the default particle transport package. A GEANT~3 description is available at http://wwwasdoc.web.cern.ch/wwwasdoc/geant_html3/geantall.html.

You can download the GEANT 3 distribution from the ROOT SVN repository and compile it in the following way:

make GEANT 3

```
  cd $ALICE
   svn co https://root.cern.ch/svn/geant3/tags/v1-9 geant3
   cd $ALICE/geant3
   export PLATFORM=`root-config --arch`
  make
```

Please note that GEANT 3 is downloaded into the $ALICE directory. Another important feature is the PLATFORM environment variable. If it is not set, the Geant 3 Makefile will set it to the result of `root-config --arch`.

3.4.4 GEANT 4

To use GEANT [197], some additional software has to be installed. GEANT 4 needs the CLHEP [20] package, the user can get the tar file (or "tarball") from http://proj-clhep.web.cern.ch/proj-clhep. Then, the installation can be done in the following way:

make CLHEP

```
tar zxvf clhep-2.0.3.1.tgz
cd 2.0.3.1/CLHEP
./configure --prefix=$ALICE/CLHEP # Select the place to install CLHEP
make
make check
make install
```

Another possibility is to use the CLHEP CVS repository:

make CLHEP from CVS

```
cvs -d :pserver:anonymous@clhep.cvs.cern.ch:/cvs/CLHEP login
# Empty password
cvs -d :pserver:anonymous@clhep.cvs.cern.ch:/cvs/CLHEP \
      co -r CLHEP_2_0_3_1 CLHEP
cd CLHEP
./bootstrap
./configure --prefix=$ALICE/CLHEP # Select the place to install CLHEP
make
make check
make install
```

Now the following lines should be added to ~/.bash_profile:

```
% export CLHEP_BASE_DIR=$ALICE/CLHEP
```

The next step is to install GEANT 4. The GEANT 4 distribution is available from http://geant4.web.cern.ch/geant4. Typically the following files will be downloaded (the current versions may differ from the ones below):

- geant4.8.1.p02.tar.gz: source tarball
- G4NDL.3.9.tar.gz: G4NDL version 3.9 neutron data files with thermal cross sections
- G4EMLOW4.0.tar.gz: data files for low energy electromagnetic processes - version 4.0
- PhotonEvaporation.2.0.tar.gz: data files for photon evaporation - version 2.0
- RadiativeDecay.3.0.tar.gz: data files for radioactive decay hadronic processes - version 3.0
- G4ELASTIC.1.1.tar.gz: data files for high energy elastic scattering processes -

version 1.1

Then the following steps have to be executed:

make GEANT4

```
tar zxvf geant4.8.1.p02.tar.gz
cd geant4.8.1.p02
mkdir data
cd data
tar zxvf ../../G4NDL.3.9.tar.gz
tar zxvf ../../G4EMLOW4.0.tar.gz
tar zxvf ../../PhotonEvaporation.2.0.tar.gz
tar zxvf ../../RadiativeDecay.3.0.tar.gz
tar zxvf ../../G4ELASTIC.1.1.tar.gz
cd ..

# Configuration and compilation

./Configure -build
```

As answer choose the default value, except of the following:

Geant4 library path:	$ALICE/geant4
Copy all Geant4 headers in one directory?	Y
CLHEP location:	$ALICE/CLHEP
build 'shared' (.so) libraries?	Y
G4VIS_BUILD_OPENGLX_DRIVER and G4VIS_USE_OPENGLX	Y
G4LIB_BUILD_G3TOG4	Y

Now a long compilation...

For installation in the selected place ($ALICE/geant4):

```
./Configure -install
```

Environment variables - The execution of the env.sh script can be done from the ~/.bash_profile to have the GEANT~4 environment variables initialized automatically. Please note the "dot" in the beginning:

```
# The <platform> has to be replaced by the actual value
. $ALICE/geant4/src/geant4/.config/bin/<platform>/env.sh
```

3.4.5 FLUKA

The installation of FLUKA [197] consists of the following steps:

1. register as FLUKA user at http://www.fluka.org if you have not yet done so. You will receive your “fuid” number and will set you password;
2. download the latest FLUKA version from http://www.fluka.org. Use your “fuid” registration and password when prompted. You will obtain a tarball containing the FLUKA libraries, for example fluka2006.3-linuxAA.tar.gz
3. install the libraries;
4. install FLUKA

```
# Make fluka subdirectory in $ALICE
cd $ALICE
mkdir fluka

# Unpack the FLUKA libraries in the $ALICE/fluka directory.
# Please set correctly the path to the FLUKA tarball.
cd fluka
tar zxvf  <path_to_fluka_tarball>/fluka2006.3-linuxAA.tar.gz

# Set the environment variables
export FLUPRO=$ALICE/fluka
```

5. compile TFluka;

```
% cd $ALICE_ROOT
% make all-TFluka
```

6. run AliRoot using FLUKA;

```
% cd $ALICE_ROOT/TFluka/scripts
% ./runflukageo.sh
```

7. This script creates the directory ‘tmp’ as well as all the necessary links for data and configuration files, and starts AliRoot. For the next run, it is not necessary to run the script again. The ‘tmp’ directory can be kept or renamed. The user should run AliRoot from within this directory.
8. From the AliRoot prompt start the simulation;

```
root [0] AliSimulation sim;
root [1] sim.Run();
```

9. You will get the results of the simulation in the ‘tmp’ directory.
10. reconstruct the simulated event;

```
% cd tmp
% aliroot
```

11. and from the AliRoot prompt

```
root [0] AliReconstruction rec;
root [1] rec.Run();
```

12. report any problem you encounter to the offline list alice-off@cern.ch.

3.4.6 AliRoot

The AliRoot distribution is taken from the SVN repository and then

```
% cd $ALICE
% svn co https://alisoft.cern.ch/AliRoot/trunk AliRoot
% cd $ALICE ROOT
% make
```

The AliRoot code (the above example retrieves the HEAD version from CVS) is contained in $ALICE_ROOT directory. The $ALICE_TARGET is defined automatically in the .bash_profile} via the call to `root-config --arch`.

3.4.7 Debugging

While developing code or running some ALICE program, the user may be confronted with the following execution errors:

- Floating exceptions: division by zero, sqrt from negative argument, assignment of NaN, etc.
- Segmentation violations/faults: attempt to access a memory location which it is not allowed to access by the operating system.
- Bus error: attempt to access memory that the computer cannot address.

In this case, the user will have to debug the programme in order to determine the source of the problem and fix it. There are several debugging techniques, which are briefly listed below:

- using printf(...), std::cout, assert(...) and AliDebug.
  - often this is the only easy way to find the origin of the problem;
  - assert(...) aborts the program execution if the argument is FALSE. It is a macro from cassert, it can be inactivated by compiling with -DNDEBUG.
- using the GNU Debugger (gdb), see http://sourceware.org/gdb/
  - gdb needs compilation with -g option. Sometimes -O2 -g prevents from exact tracing, so it is save to use compilation with -O0 -g for debugging purposes;
  - One can use it directly (gdb aliroot) or attach it to a process (gdb aliroot 12345 where 12345 is the process id).

Below we report the main gdb commands and their descriptions:

- **run** starts the execution of the program;
- **Control-C** stops the execution and switches to the gdb shell;
- **where <n>** prints the program stack. Sometimes the program stack is very long. The user can get the last n frames by specifying n as a parameter to

where;

- **print** prints the value of a variable or expression;

```
(gdb) print *this
```

- **up** and **down** are used to navigate through the program stack;
- **quit** exits the gdb session;
- **break** sets break point;

```
(gdb) break AliLoader.cxx:100
(gdb) break 'AliLoader::AliLoader()'
```

The automatic completion of the class methods via tab is available in case an opening quote (`) is put in front of the class name.

- **cont** continues the run;
- **watch** sets watchpoint (very slow execution). The example below shows how to check each change of fData;

```
(gdb) watch *fData
```

- **list** shows the source code;
- **help** shows the description of commands.

3.4.8 Profiling

Profiling is used to discover where the program spends most of the time, and to optimize the algorithms. There are several profiling tools available on different platforms:

- Linux tools:
  - gprof: compilation with -pg option, static libraries
  - oprofile: uses kernel module
  - VTune: instruments shared libraries.
- Sun: Sun workshop (Forte agent). It needs compilation with profiling option (-pg)
- Compaq Alpha: pixie profiler. Instruments shared libraries for profiling.

On Linux AliRoot can be built with static libraries using the special target “profile”

```
% make profile
# change LD_LIBRARY_PATH to replace lib/tgt_linux with
lib/tgt_linuxPROF
# change PATH to replace bin/tgt_linux with bin/tgt_linuxPROF
% aliroot
root [0] gAlice->Run()
root [1] .q
```

At the end of an AliRoot session, a file called ‘gmon.out’ will be created. It contains the profiling information that can be investigated using gprof.

```
% gprof `which aliroot` | tee gprof.txt
% more gprof.txt
```

VTune profiling tool

VTune is available from the Intel Web site http://www.intel.com/software/products/index.htm. It is free for non-commercial use on Linux. It provides possibility for call-graph and sampling profiling. VTune uses shared libraries and is enabled via the ‘-g’ option during compilation. Here is an example of call-graph profiling:

```
# Register an activity
% vtl activity sim -c callgraph -app aliroot," -b -q sim.C" -moi
aliroot
% vtl run sim
% vtl show
% vtl view sim::r1 -gui
```

3.4.9 Detection of run time errors

The Valgrind tool can be used for detection of run time errors on linux. It is available from http://www.valgrind.org. Valgrind is equipped with the following set of tools which are important for debugging AliRoot:

- memcheck for memory management problems;
- cachegrind: cache profiler;
- massif: heap profiler;

Here is an example of Valgrind usage:

```
% valgrind --tool=addrcheck --error-limit=no aliroot -b -q sim.C
```

ROOT memory checker

The ROOT memory checker does not work with the current version of ROOT. This section is for reference only.

The ROOT memory checker provides tests of memory leaks and other problems related to new/delete. It is fast and easy to use. Here is the recipe:

- link aliroot with -lNew. The user has to add ‘--new’ before ‘--glibs’ in the ROOTCLIBS variable of the Makefile;

- Root.MemCheck: 1 in .rootrc
- run the program: aliroot -b -q sim.C
- run memprobe -e aliroot
- Inspect the files with .info extension that have been generated.

3.4.10Useful information LSF and CASTOR

The information in this section is included for completeness: users are strongly advised to rely on the GRID tools for productions and data access.

LSF (Load Sharing Facility) is the batch system at CERN. Every user is allowed to submit jobs to the different queues. Usually the user has to copy some input files (macros, data, executables, libraries) from a local computer or from the mass-storage system to the worker node on lxbatch, to execute the program, and to store the results on the local computer or in the mass-storage system. The methods explained in the section are suitable, if the user doesn't have direct access to a shared directory, for example on AFS. The main steps and commands are described below.

In order to have access to the local desktop and to be able to use scp without password, the user has to create pair of SSH keys. Currently lxplus/lxbatch uses RSA1 cryptography. After a successful login to lxplus, the following has to be done:

```
% ssh-keygen -t rsa1
# Use empty password
% cp .ssh/identity.pub public/authorized_keys
% ln -s ../public/authorized_keys .ssh/authorized_keys
```

A list of useful LSF commands is given below:

- **bqueues** – shows available queues and their status;
- **bsub -q 8nm job.sh** – submits the shell script ‘job.sh’ to the queue 8nm, where the name of the queue indicates the “normalized CPU time” (maximal job duration 8 min of normalized CPU time);
- **bjobs** – lists all unfinished jobs of the user;
- **lsrun -m lxbXXXX xterm** – returns a xterm running on the batch node lxbXXXX. This allows you to inspect the job output and to debug a batch job.

Each batch job stores the output in directory ‘LSFJOB_XXXXXX’, where ‘XXXXXX’ is the job id. Since the home directory is on AFS, the user has to redirect the verbose output, otherwise the AFS quota might be exceeded and the jobs will fail.

The CERN mass storage system is CASTOR2 [21]. Every user has his/her own CASTOR2 space, for example /castor/cern.ch/user/p/phristov.

The commands of CASTOR2 start with prefix “ns” of “rf”. Here is very short list of useful commands:

- **nsls /castor/cern.ch/user/p/phristov** lists the CASTOR space of user phristov;
- **rfdir /castor/cern.ch/user/p/phristov** the same as above, but the output is in long format;
- **nsmkdir test** creates a new directory 'test' in the CASTOR space of the user;
- **rfcp /castor/cern.ch/user/p/phristov/test/galice.root** copies the file from CASTOR to the local directory. If the file is on tape, this will trigger the stage-in procedure, which might take some time.
- **rfcp AliESDs.root /castor/cern.ch/p/phristov/test** copies the local file 'AliESDs.root' to CASTOR in the subdirectory 'test' and schedules it for migration to tape.

The user also has to be aware, that the behavior of CASTOR depends on the environment variables RFIO_USE_CASTOR_V2 (=YES), STAGE_HOST (=castoralice) and STAGE_SVCCLASS (=default). They are set by default to the values for the group (z2 in case of ALICE).

Below the user can find a job example, where the simulation and reconstruction are run using the corresponding macros 'sim.C' and 'rec.C'.

An example of such macros will be given later.

LSF example job

```
#! /bin/sh
# Take all the C++ macros from the local computer to the working
directory
command scp phristov@pcepalice69:/home/phristov/pp/*.C .

# Execute the simulation macro. Redirect the output and error streams
command aliroot -b -q sim.C > sim.log 2>&1

# Execute the reconstruction macro. Redirect the output and error
streams
command aliroot -b -q rec.C > rec.log 2>&1

# Create a new CASTOR directory for this job ($LSB_JOBID)
command rfmkdir /castor/cern.ch/user/p/phristov/pp/$LSB_JOBID

# Copy all log files to CASTOR
for a in *.log; do rfcp $a /
castor/cern.ch/user/p/phristov/pp/$LSB_JOBID; done
# Copy all ROOT files to CASTOR
for a in *.root; do rfcp $a /
castor/cern.ch/user/p/phristov/pp/$LSB_JOBID; done
```

3.5Simulation

3.5.1 Introduction

Heavy-ion collisions produce a very large number of particles in the final state. This is a challenge for the reconstruction and analysis algorithms. Detector design and development of these algorithms require a predictive and precise simulation of the detector response. Model predictions, as discussed in the first volume of Physics Performance Report for the charged multiplicity at LHC in Pb–Pb collisions, vary from 1400 to 8000 particles in the central unit of rapidity. The experiment was designed when the highest available nucleon–nucleon center-of-mass energy heavy-ion interactions was at 20 GeV per nucleon–nucleon pair at CERN SPS, i.e. a factor of about 300 less than the energy at LHC. Recently, the RHIC collider came online. Its top energy of 200 GeV per nucleon–nucleon pair is still 30 times less than the LHC energy. The RHIC data seem to suggest that the LHC multiplicity will be on the lower side of the interval. However, the extrapolation is so large that both the hardware and software of ALICE have to be designed for the highest multiplicity. Moreover, as the predictions of different generators of heavy-ion collisions differ substantially at LHC energies, we have to use several of them and compare the results.

The simulation of the processes involved in the transport through the detector of the particles emerging from the interaction is confronted with several problems:

- Existing event generators give different answers on parameters such as expected multiplicities, p_T-dependence and rapidity dependence at LHC energies.
- Most of the physics signals, like hyperon production, high-p_t phenomena, open charm and beauty, quarkonia etc. are not exactly reproduced by the existing event generators.
- Simulation of small cross-sections would demand prohibitively high computing resources to simulate a number of events that is commensurable with the expected number of detected events in the experiment.
- Existing generators do not provide for event topologies like momentum correlations, azimuthal flow etc.

Nevertheless, to allow efficient simulations, we have adopted a framework that allows for a number of options:

- The simulation framework provides an interface to external generators, like HIJING [197] and DPMJET [[22]].
- A parameterized, signal-free, underlying event where the produced multiplicity can be specified as an input parameter is provided.
- Rare signals can be generated using the interface to external generators like PYTHIA or simple parameterizations of transverse momentum and rapidity spectra defined in function libraries.

- The framework provides a tool to assemble events from different signal generators (event cocktails).
- The framework provides tools to combine underlying events and signal events at the primary particle level (cocktail) and at the summable digit level (merging).
- “afterburners” are used to introduce particle correlations in a controlled way. An afterburner is a program which changes the momenta of the particles produced by another generator, and thus modifies the multi-particle momentum distributions, as desired.

The implementation of this strategy is described below. The results of different Monte Carlo generators for heavy-ion collisions are described in section 3.5.4.

3.5.2 Simulation framework

The simulation framework covers the simulation of primary collisions and generation of the emerging particles, the transport of particles through the detector, the simulation of energy depositions (hits) in the detector components, their response in form of so called summable digits, the generation of digits from summable digits with the optional merging of underlying events and the creation of raw data.

The **AliSimulation** class provides a simple user interface to the simulation framework. This section focuses on the simulation framework from the (detector) software developer point of view.

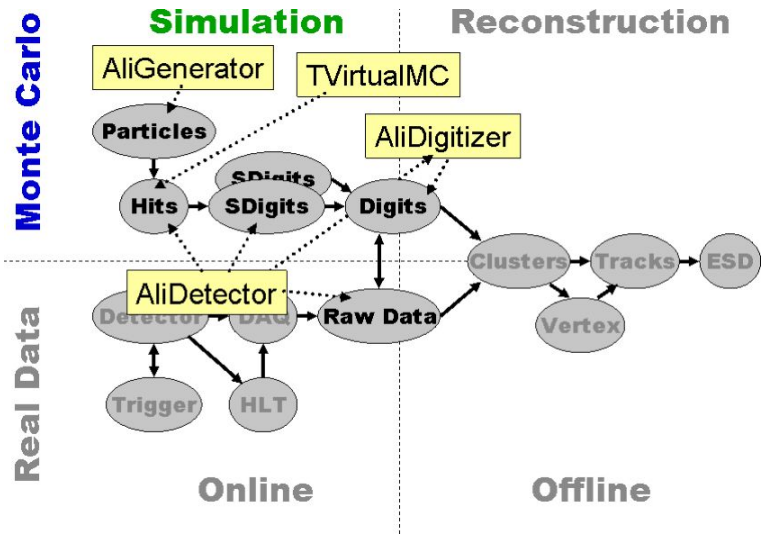


Figure 2: Simulation framework.

Generation of Particles

Different generators can be used to produce particles emerging from the collision. The class **AliGenerator** is the base class defining the virtual interface to the generator programs. The generators are described in more detail in the ALICE PPR Volume 1 and in the next chapter.

Virtual Monte Carlo

A class derived from **TVirtualMC** performs the simulation of particles traversing the detector components. The Virtual Monte Carlo also provides an interface to construct the geometry of detectors. The geometrical modeller **TGeo** does the task of the geometry description. The concrete implementation of the virtual Monte Carlo application **TVirtualMCApplication** is **AliMC**. The Monte Carlos used in ALICE are GEANT 3.21, GEANT 4 and FLUKA. More information can be found on the VMC Web page: http://root.cern.ch/root/vmc.

As explained above, our strategy was to develop a virtual interface to the detector simulation code. We call the interface to the transport code 'virtual Monte Carlo'. It is implemented via C++ virtual classes and is schematically shown in Figure 3. Implementations of those abstract classes are C++ programs or wrapper classes that interface to FORTRAN programs.

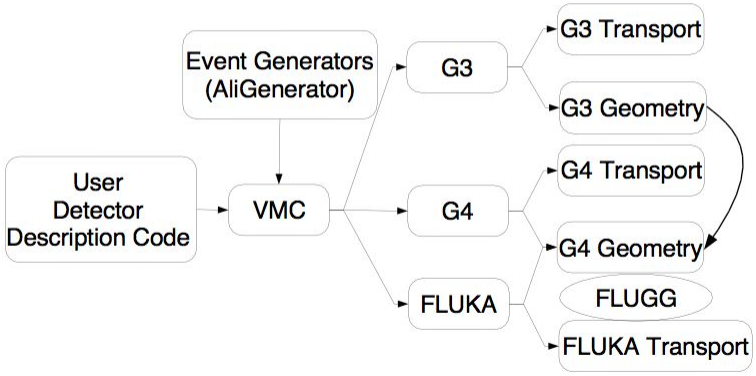


Figure 3. Virtual Monte Carlo

Thanks to the virtual Monte Carlo, we have converted all FORTRAN user code developed for GEANT 3 into C++, including the geometry definition and the user scoring routines, StepManager. These have been integrated in the detector classes of the AliRoot framework. The output of the simulation is saved directly with ROOT I/O, simplifying the development of the digitization and reconstruction code in C++.

Modules and Detectors

A class derived from **AliModule** describes each module of the ALICE detector. Classes for active modules (i.e. detectors) are not derived directly from **AliModule** but from its subclass **AliDetector**. These base classes define the interface to the simulation framework via a set of virtual methods.

Configuration File (Config.C)

The configuration file is a C++ macro that is processed before the simulation starts. It creates and configures the Monte Carlo object, the generator object, the magnetic field map and the detector modules. A detailed description is given below.

Detector Geometry

The virtual Monte Carlo application creates and initializes the geometry of the detector modules by calling the virtual functions CreateMaterials, CreateGeometry, Init and

BuildGeometry.

Vertexes and Particles

In case the simulated event is intended to be merged with an underlying event, the primary vertex is taken from the file containing the underlying event by using the vertex generator **AliVertexGenFile**. Otherwise, the primary vertex is generated according to the generator settings. Then the particles emerging from the collision are generated and put on the stack (an instance of **AliStack**). The Monte Carlo object performs the transport of particles through the detector. The external decayer **AliDecayerPythia** usually handles the decay of particles.

Hits and Track References

The Monte Carlo simulates the transport of a particle step by step. After each step the virtual method StepManager of the module in which the particle currently is located is called. In this StepManager method, calling AddHit creates the hits in the detector. Optionally also track references (location and momentum of simulated particles at selected places) can be created by calling AddTackReference. AddHit has to be implemented by each detector whereas AddTackReference is already implemented in **AliModule**. The detector class manages the container and the branch for the hits – and for the (summable) digits – via a set of so-called loaders. The relevant data members and methods are fHits, fDigits, ResetHits, ResetSDigits, ResetDigits, MakeBranch and SetTreeAddress.

For each detector methods like PreTrack, PostTrack, FinishPrimary, FinishEvent and FinishRun are called during the simulation when the conditions indicated by the method names are fulfilled.

Summable Digits

Calling the virtual method Hits2SDigits of a detector creates summable digits. This method loops over all events, creates the summable digits from hits and stores them in the sdigits file(s).

Digitization and Merging

Dedicated classes derived from **AliDigitizer** are used for the conversion of summable digits into digits. Since **AliDigitizer** is a **TTask**, this conversion is done for the current event by the Exec method. Inside this method the summable digits of all input streams have to be added, combined with noise, converted to digital values taking into account possible thresholds, and stored in the digits container.

An object of type **AliRunDigitizer** manages the input streams (more than one in case of merging) as well as the output stream. The methods GetNinputs, GetInputFolderName and GetOutputFolderName return the relevant information. The run digitizer is accessible inside the digitizer via the protected data member fManager. If the flag fRegionOfInterest is set, only detector parts where summable digits from the signal event are present should be digitized. When Monte Carlo labels are assigned to digits, the stream-dependent offset given by the method GetMask is added to the label of the summable digit.

The detector specific digitizer object is created in the virtual method CreateDigitizer of

the concrete detector class. The run digitizer object is used to construct the detector digitizer. The Init method of each digitizer is called before the loop over the events is started.

A direct conversion from hits directly to digits can be implemented in the method Hits2Digits of a detector. The loop over the events takes place inside the method. Of course merging is not supported in this case.

An example of a simulation script that can be used for simulation of proton-proton collisions is provided below:

Simulation run

```
void sim(Int_t nev=100) {
 AliSimulation simulator;
// Measure the total time spent in the simulation
 TStopwatch timer;
 timer.Start();
// List of detectors, where both summable digits and digits are
provided
 simulator.SetMakeSDigits("TRD TOF PHOS EMCAL HMPID MUON ZDC PMD FMD
T0 VZERO");
// Direct conversion of hits to digits for faster processing (ITS TPC)
 simulator.SetMakeDigitsFromHits("ITS TPC");
 simulator.Run(nev);
 timer.Stop();
 timer.Print();
}
```

The following example shows how one can do event merging:

Event merging

```
void sim(Int_t nev=6) {
 AliSimulation simulator;
// The underlying events are stored in a separate directory.
// Three signal events will be merged in turn with each
// underlying event
 simulator.MergeWith("../backgr/galice.root",3);
 simulator.Run(nev);
 }
```

Raw Data

The digits stored in ROOT containers can be converted into the DATE [23] format that will be the `payload' of the ROOT classes containing the raw data. This is done for the current event in the method Digits2Raw of the detector.

The class **AliSimulation** manages the simulation of raw data. In order to create raw data DDL files, it loops over all events. For each event it creates a directory, changes to this directory and calls the method Digits2Raw of each selected detector. In the Digits2Raw method the DDL files of a detector are created from the digits for the current event.

For the conversion of the DDL files to a DATE file the **AliSimulation** class uses the tool dateStream. To create a raw data file in ROOT format with the DATE output as payload the program alimdc is utilized.

The only part that has to be implemented in each detector is the Digits2Raw method of the detectors. In this method one file per DDL has to be created following the conventions for file names and DDL IDs. Each file is a binary file with a DDL data header in the beginning. The DDL data header is implemented in the structure **AliRawDataHeader**. The data member fSize should be set to the total size of the DDL raw data including the size of the header. The attribute bit 0 should be set by calling the method to indicate that the data in this file is valid. The attribute bit 1 can be set to indicate compressed raw data.

The detector-specific raw data is stored in the DDL files following the DDL data header. The format of this raw data should be as close as possible to the one that will be delivered by the detector. This includes the order in which the channels will be read out.

Below we show an example of raw data creation for all the detectors:

```
void sim(Int_t nev=1) {
 AliSimulation simulator;
 // Create raw data for ALL detectors, rootify it and store in the
 // file raw,root. Do not delete the intermediate files
 simulator.SetWriteRawData("ALL","raw.root",kFALSE);
 simulator.Run(nev);
 }
```

3.5.3 Configuration: example of Config.C

The example below contains as comments the most important information:

Example of Config.C

```
// Function converting pseudorapidity
// interval to polar angle interval. It is used to set
// the limits in the generator
Float_t EtaToTheta(Float_t arg){
  return (180./TMath::Pi())*2.*atan(exp(-arg));
}

// Set Random Number seed using the current time
TDatime dat;
static UInt_t sseed = dat.Get();

void Config()
{
  gRandom->SetSeed(sseed);
  cout<<"Seed for random number generation= "<<gRandom->GetSeed()
<<endl;
```

```
  // Load GEANT 3 library. It has to be in LD_LIBRARY_PATH
  gSystem->Load("libgeant321");

  // Instantiation of the particle transport package. gMC is set
internaly
  new TGeant3TGeo("C++ Interface to Geant3");

  // Create run loader and set some properties
  AliRunLoader* rl =  AliRunLoader::Open("galice.root",
                                  AliConfig::GetDefaultEventFolderName
(),
                                  "recreate");
  if (!rl) Fatal("Config.C","Can not instatiate the Run Loader");
  rl->SetCompressionLevel(2);
  rl->SetNumberOfEventsPerFile(3);

  // Register the run loader in gAlice
  gAlice->SetRunLoader(rl);

  // Set external decayer
  TVirtualMCDecayer *decayer = new AliDecayerPythia();
  decayer->SetForceDecay(kAll); // kAll means no specific decay is
forced
  decayer->Init();

  // Register the external decayer in the transport package
  gMC->SetExternalDecayer(decayer);

  // STEERING parameters FOR ALICE SIMULATION
  // Specify event type to be transported through the ALICE setup
  // All positions are in cm, angles in degrees, and P and E in GeV
  // For the details see the GEANT 3 manual

  // Switch on/off the physics processes (global)
  // Please consult the file data/galice.cuts for detector
  // specific settings, i.e. DRAY
  gMC->SetProcess("DCAY",1); // Particle decay
  gMC->SetProcess("PAIR",1); // Pair production
  gMC->SetProcess("COMP",1); // Compton scattering
  gMC->SetProcess("PHOT",1); // Photo effect
  gMC->SetProcess("PFIS",0); // Photo fission
  gMC->SetProcess("DRAY",0); // Delta rays
  gMC->SetProcess("ANNI",1); // Positron annihilation
  gMC->SetProcess("BREM",1); // Bremstrahlung
  gMC->SetProcess("MUNU",1); // Muon nuclear interactions
  gMC->SetProcess("CKOV",1); // Cerenkov production
  gMC->SetProcess("HADR",1); // Hadronic interactions
  gMC->SetProcess("LOSS",2); // Energy loss (2=complete fluct.)
  gMC->SetProcess("MULS",1); // Multiple scattering
  gMC->SetProcess("RAYL",1); // Rayleigh scattering

  // Set the transport package cuts
```

```
  Float_t cut = 1.e-3;        // 1MeV cut by default
  Float_t tofmax = 1.e10;

  gMC->SetCut("CUTGAM", cut); // Cut for gammas
  gMC->SetCut("CUTELE", cut); // Cut for electrons
  gMC->SetCut("CUTNEU", cut); // Cut for neutral hadrons
  gMC->SetCut("CUTHAD", cut); // Cut for charged hadrons
  gMC->SetCut("CUTMUO", cut); // Cut for muons
  gMC->SetCut("BCUTE",  cut); // Cut for electron brems.
  gMC->SetCut("BCUTM",  cut); // Cut for muon brems.
  gMC->SetCut("DCUTE",  cut); // Cut for electron delta-rays
  gMC->SetCut("DCUTM",  cut); // Cut for muon delta-rays
  gMC->SetCut("PPCUTM", cut); // Cut for e+e- pairs by muons
  gMC->SetCut("TOFMAX", tofmax); // Time of flight cut

  // Set up the particle generation

  // AliGenCocktail permits to combine several different generators
  AliGenCocktail *gener = new AliGenCocktail();

  // The phi range is always inside 0-360
  gener->SetPhiRange(0, 360);

  // Set pseudorapidity range from -8 to 8.
  Float_t thmin = EtaToTheta(8);   // theta min. <--> eta max
  Float_t thmax = EtaToTheta(-8);  // theta max. <--> eta min
  gener->SetThetaRange(thmin,thmax);

  gener->SetOrigin(0, 0, 0);  // vertex position
  gener->SetSigma(0, 0, 5.3);   // Sigma in (X,Y,Z) (cm) on IP
position
  gener->SetCutVertexZ(1.);     // Truncate at 1 sigma
  gener->SetVertexSmear(kPerEvent);

  // First cocktail component: 100 ``background'' particles
  AliGenHIJINGpara *hijingparam = new AliGenHIJINGpara(100);
  hijingparam->SetMomentumRange(0.2, 999);
  gener->AddGenerator(hijingparam,"HIJING PARAM",1);

  // Second cocktail component: one gamma in PHOS direction
  AliGenBox *genbox = new AliGenBox(1);
  genbox->SetMomentumRange(10,11.);
  genbox->SetPhiRange(270.5,270.7);
  genbox->SetThetaRange(90.5,90.7);
  genbox->SetPart(22);
  gener->AddGenerator(genbox,"GENBOX GAMMA for PHOS",1);

  gener->Init(); // Initialization of the coctail generator

  // Field (the last parameter is 1 => L3 0.4 T)
  AliMagFMaps* field = new AliMagFMaps("Maps","Maps", 2, 1., 10., 1);
  gAlice->SetField(field);
```

```
  // Make sure the current ROOT directory is in galice.root
  rl->CdGAFile();

  // Build the setup and set some detector parameters

  // ALICE BODY parameters. BODY is always present
  AliBODY *BODY = new AliBODY("BODY", "ALICE envelop");

  // Start with Magnet since detector layouts may be depending
  // on the selected Magnet dimensions
  AliMAG *MAG = new AliMAG("MAG", "Magnet");

  AliABSO *ABSO = new AliABSOv0("ABSO", "Muon Absorber");       //
Absorber

  AliDIPO *DIPO = new AliDIPOv2("DIPO", "Dipole version 2");    //
Dipole magnet

  AliHALL *HALL = new AliHALL("HALL", "ALICE Hall");            //
Hall

  AliFRAMEv2 *FRAME = new AliFRAMEv2("FRAME", "Space Frame");   //
Space frame

  AliSHIL *SHIL = new AliSHILv2("SHIL", "Shielding Version 2"); //
Shielding

  AliPIPE *PIPE = new AliPIPEv0("PIPE", "Beam Pipe");           //
Beam pipe

  // ITS parameters
  AliITSvPPRasymmFMD *ITS  = new AliITSvPPRasymmFMD("ITS",
             "ITS PPR detailed version with asymmetric services");
  ITS->SetMinorVersion(2);      // don't change it if you're not an
ITS developer
  ITS->SetReadDet(kFALSE);      // don't change it if you're not an
ITS developer
  ITS->SetThicknessDet1(200.);  // detector thickness on layer 1:
[100,300] mkm
  ITS->SetThicknessDet2(200.);  // detector thickness on layer 2:
[100,300] mkm
  ITS->SetThicknessChip1(150.); // chip thickness on layer 1:
[150,300] mkm
  ITS->SetThicknessChip2(150.); // chip thickness on layer 2:
[150,300]
  ITS->SetRails(0);               // 1 -> rails in ; 0 -> rails out
  ITS->SetCoolingFluid(1);      // 1 -> water ; 0 -> freon
  ITS->SetEUCLID(0);            // no output for the EUCLID CAD system



  AliTPC *TPC = new AliTPCv2("TPC", "Default");                 // TPC

  AliTOF *TOF = new AliTOFv5T0("TOF", "normal TOF");            // TOF
```

```
  AliHMPID *HMPID = new AliHMPIDv1("HMPID", "normal
HMPID");         // HMPID

  AliZDC *ZDC = new AliZDCv2("ZDC", "normal ZDC");              // ZDC

  AliTRD *TRD = new AliTRDv1("TRD", "TRD slow simulator");      // TRD

  AliFMD *FMD = new AliFMDv1("FMD", "normal FMD");              // FMD

  AliMUON *MUON = new AliMUONv1("MUON", "default");             //
MUON

  AliPHOS *PHOS = new AliPHOSv1("PHOS", "IHEP");                //
PHOS

  AliPMD *PMD = new AliPMDv1("PMD", "normal PMD");              // PMD

  AliT0 *T0 = new AliT0v1("T0", "T0 Detector");  // T0

  // EMCAL
  AliEMCAL *EMCAL = new AliEMCALv2("EMCAL",
"SHISH_77_TRD1_2X2_FINAL_110DEG");

  AliVZERO *VZERO = new AliVZEROv7("VZERO", "normal VZERO");    //
VZERO
}
```

3.5.4 Event generation

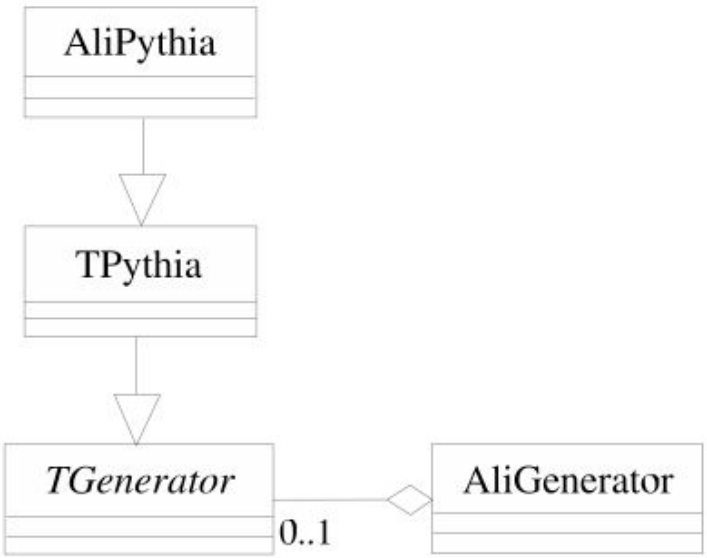


*Figure 4: **AliGenerator** is the base class, which has the responsibility to generate the*

primary particles of an event. Some realizations of this class do not generate the particles themselves but delegate the task to an external generator like PYTHIA through the ***TGenerator*** *interface.*

3.5.4.1Parametrized generation

The event generation based on parametrization can be used to produce signal-free final states. It avoids the dependences on a specific model, and is efficient and flexible. It can be used to study the track reconstruction efficiency as a function of the initial multiplicity and occupation.

AliGenHIJINGparam [24] is an example of an internal AliRoot generator, based on parametrized pseudorapidity density and transverse momentum distributions of charged and neutral pions and kaons. The pseudorapidity distribution was obtained from a HIJING simulation of central Pb–Pb collisions and scaled to a charged-particle multiplicity of 8000 in the pseudo rapidity interval $|\eta|<0.5$. Note that this is about 10% higher than the corresponding value for a rapidity density with an average dN/dy of 8.000 in the interval $|y|<0.5$.

The transverse-momentum distribution is parametrized from the measured CDF pion p_t-distribution at $\sqrt{s} = 1.8\ TeV$. The corresponding kaon p_t-distribution was obtained from the pion distribution by m_t-scaling. 197For the details of these parametrizations see [197].

In many cases, the expected transverse momentum and rapidity distributions of particles are known. In other cases, the effect of variations in these distributions must be investigated. In both situations, it is appropriate to use generators that produce primary particles and their decays sampling from parametrized spectra. To meet the different physics requirements in a modular way, the parametrizations are stored in independent function libraries wrapped into classes that can be plugged into the generator. This is schematically illustrated in Figure 5 where four different generator libraries can be loaded via the abstract generator interface.

It is customary in heavy-ion event generation to superimpose different signals on an event in order to tune the reconstruction algorithms. This is possible in AliRoot via the so-called cocktail generator (see Figure 6). This creates events from user-defined particle cocktails by choosing as ingredients a list of particle generators.

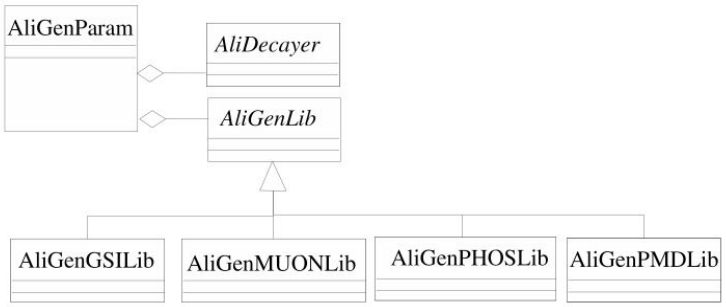


*Figure 5: **AliGenParam** is a realization that generates particles using parameterized p_t and pseudo-rapidity distributions. Instead of coding a fixed number of parameterizations directly into the class implementations, user defined parameterization libraries (AliGenLib) can be connected at run time, providing maximum flexibility.*

An example of **AliGenParam** usage is presented below:

```
// Example for J/psi Production from Parameterization
// using default library (AliMUONlib)
AliGenParam *gener = new AliGenParam(ntracks,
AliGenMUONlib::kUpsilon);
gener->SetMomentumRange(0,999); // Wide cut on the Upsilon momentum
gener->SetPtRange(0,999);  // Wide cut on Pt
gener->SetPhiRange(0. , 360.); // Full azimutal range
gener->SetYRange(2.5,4);  // In the acceptance of the MUON arm
gener->SetCutOnChild(1);  // Enable cuts on Upsilon decay products
gener->SetChildThetaRange(2,9); // Theta range for the decay products
gener->SetOrigin(0,0,0);  // Vertex position
gener->SetSigma(0,0,5.3);  // Sigma in (X,Y,Z) (cm) on IP position
gener->SetForceDecay(kDiMuon); // Upsilon->mu+ mu- decay
gener->SetTrackingFlag(0);  // No particle transport
gener->Init();
```

To facilitate the usage of different generators, we have developed an abstract generator interface called **AliGenerator**, see Figure 4. The objective is to provide the user with an easy and coherent way to study a variety of physics signals as well as a full set of tools for testing and background studies. This interface allows the study of full events, signal processes and a mixture of both, i.e. cocktail events (see example below).

Several event generators are available via the abstract ROOT class that implements the generic generator interface, **TGenerator**. By means of implementations of this abstract base class, we wrap FORTRAN Monte Carlo codes like PYTHIA, HERWIG, and HIJING that are thus accessible from the AliRoot classes. In particular the interface to PYTHIA includes the use of nuclear structure functions of LHAPDF.

3.5.4.2Pythia6

Pythia is used for simulation of proton-proton interactions and for generation of jets in case of event merging. An example of minimum bias Pythia events is presented below:

```
AliGenPythia *gener = new AliGenPythia(-1);
gener->SetMomentumRange(0,999999);
gener->SetThetaRange(0., 180.);
gener->SetYRange(-12,12);
gener->SetPtRange(0,1000);
gener->SetProcess(kPyMb);  // Min. bias events
gener->SetEnergyCMS(14000.);  // LHC energy
gener->SetOrigin(0, 0, 0);  // Vertex position
gener->SetSigma(0, 0, 5.3);  // Sigma in (X,Y,Z) (cm) on IP position
gener->SetCutVertexZ(1.);  // Truncate at 1 sigma
gener->SetVertexSmear(kPerEvent);// Smear per event
gener->SetTrackingFlag(1);  // Particle transport
gener->Init()
```

3.5.4.3HIJING

HIJING (Heavy-Ion Jet Interaction Generator) combines a QCD-inspired model of jet production [197] using the Lund model [25] for jet fragmentation. Hard or semi-hard parton scatterings with transverse momenta of a few GeV are expected to dominate high-energy heavy-ion collisions. The HIJING model has been developed with special emphasis on the role of mini jets in p–p, p–A and A–A reactions at collider energies.

Detailed systematic comparisons of HIJING results with a wide range of data demonstrate a qualitative understanding of the interplay between soft string dynamics and hard QCD interactions. In particular, HIJING reproduces many inclusive spectra, two-particle correlations, as well as the observed flavour and multiplicity dependence of the average transverse momentum.

The Lund FRITIOF [26] model and the Dual Parton Model [27] (DPM) have guided the formulation of HIJING for soft nucleus–nucleus reactions at intermediate energies: $\sqrt{NN}$; $20\,GeV$. The hadronic-collision model has been inspired by the successful implementation of perturbative QCD processes in PYTHIA [197]. Binary scattering with Glauber geometry for multiple interactions are used to extrapolate to p–A and A–A collisions.

Two important features of HIJING are jet quenching and nuclear shadowing. Under jet quenching, we understand the energy loss of partons in nuclear matter. It is responsible for an increase of the particle multiplicity at central rapidities. Jet quenching is taken into account by an expected energy loss of partons traversing dense matter. A simple colour configuration is assumed for the multi-jet system and the Lund fragmentation model is used for the hadroniation. HIJING does not simulate secondary interactions.

Shadowing describes the modification of the free nucleon parton density in the nucleus. At low-momentum fractions, x, observed by collisions at the LHC, shadowing results in a decrease of multiplicity. Parton shadowing is taken into account using a

parameterization of the modification.

Here is an example of event generation with HIJING:

```
AliGenHijing *gener = new AliGenHijing(-1);
gener->SetEnergyCMS(5500.); // center of mass energy
gener->SetReferenceFrame("CMS"); // reference frame
gener->SetProjectile("A", 208, 82); // projectile
gener->SetTarget ("A", 208, 82); // projectile
gener->KeepFullEvent(); // HIJING will keep the full parent child
chain
gener->SetJetQuenching(1); // enable jet quenching
gener->SetShadowing(1); // enable shadowing
gener->SetDecaysOff(1); // neutral pion and heavy particle decays
switched off
gener->SetSpectators(0); // Don't track spectators
gener->SetSelectAll(0); // kinematic selection
gener->SetImpactParameterRange(0., 5.); // Impact parameter range (fm)
gener->Init()
```

3.5.4.4Additional universal generators

The following universal generators are available in AliRoot:

- **AliGenDPMjet**: this is an implementation of the dual parton model [197];
- **AliGenIsajet**: a Monte Carlo event generator for p–p, $\bar{p}$–p, and e^+e^- [28];
- **AliGenHerwig**: Monte Carlo package for simulating Hadron Emission Reactions With Interfering Gluons [29].

An example of HERWIG configuration in the Config.C is shown below:

```
AliGenHerwig *gener = new AliGenHerwig(-1);
// final state kinematic cuts
gener->SetMomentumRange(0,7000);
gener->SetPhiRange(0. ,360.);
gener->SetThetaRange(0., 180.);
gener->SetYRange(-10,10);
gener->SetPtRange(0,7000);
// vertex position and smearing
gener->SetOrigin(0,0,0);  // vertex position
gener->SetVertexSmear(kPerEvent);
gener->SetSigma(0,0,5.6);  // Sigma in (X,Y,Z) (cm) on IP position
// Beam momenta
gener->SetBeamMomenta(7000,7000);
// Beams
gener->SetProjectile("P");
gener->SetTarget("P");
// Structure function
gener->SetStrucFunc(kGRVHO);
```

```
// Hard scatering
gener->SetPtHardMin(200);
gener->SetPtRMS(20);
// Min bias
gener->SetProcess(8000);
```

3.5.4.5Generators for specific studies

MevSim

MevSim [30] was developed for the STAR experiment to quickly produce a large number of A–A collisions for some specific needs; initially for HBT studies and for testing of reconstruction and analysis software. However, since the user is able to generate specific signals, it was extended to flow and event-by-event fluctuation analysis.

MevSim generates particle spectra according to a momentum model chosen by the user. The main input parameters are: types and numbers of generated particles, momentum-distribution model, reaction-plane and azimuthal-anisotropy coefficients, multiplicity fluctuation, number of generated events, etc. The momentum models include factorized p_t and rapidity distributions, non-expanding and expanding thermal sources, arbitrary distributions in y and p_t and others. The reaction plane and azimuthal anisotropy is defined by the Fourier coefficients (maximum of six) including directed and elliptical flow. Resonance production can also be introduced.

MevSim was originally written in FORTRAN. It was later integrated into AliRoot. A complete description of the AliRoot implementation of MevSim can be found on the web page (http://home.cern.ch/~radomski).

GeVSim

GeVSim is based on the MevSim [198] event generator developed for the STAR experiment.

GeVSim [31] is a fast and easy-to-use Monte Carlo event generator implemented in AliRoot. It can provide events of similar type configurable by the user according to the specific need of a simulation project, in particular, that of flow and event-by-event fluctuation studies. It was developed to facilitate detector performance studies and for the test of algorithms. GeVSim can also be used to generate signal-free events to be processed by afterburners, for example the HBT processor.

198

GeVSim generates a list of particles by randomly sampling a distribution function. The user explicitly defines the parameters of single-particle spectra and their event-by-event fluctuations. Single-particle transverse-momentum and rapidity spectra can be either selected from a menu of four predefined distributions, the same as in MevSim, or provided by user.

Flow can be easily introduced into simulated events. The parameters of the flow are defined separately for each particle type and can be either set to a constant value or parameterized as a function of transverse momentum and rapidity. Two

parameterizations of elliptic flow based on results obtained by RHIC experiments are provided.

GeVSim also has extended possibilities for simulating of event-by-event fluctuations. The model allows fluctuations following an arbitrary analytically defined distribution in addition to the Gaussian distribution provided by MevSim. It is also possible to systematically alter a given parameter to scan the parameter space in one run. This feature is useful when analyzing performance with respect to, for example, multiplicity or event-plane angle.

The current status and further development of GeVSim code and documentation can be found in [32].

HBT processor

Correlation functions constructed with the data produced by MEVSIM or any other event generator are normally flat in the region of small relative momenta. The HBT-processor afterburner introduces two particle correlations into the set of generated particles. It shifts the momentum of each particle so that the correlation function of a selected model is reproduced. The imposed correlation effects, due to Quantum Statistics (QS) and Coulomb Final State Interactions (FSI), do not affect the single-particle distributions and multiplicities. The event structures before and after the HBT processor are identical. Thus, the event reconstruction procedure with and without correlations is also identical. However, the track reconstruction efficiency, momentum resolution and particle identification are in general not identical, since correlated particles have a special topology at small relative velocities. We can thus verify the influence of various experimental factors on the correlation functions.

The method proposed by L. Ray and G.W. Hoffmann [33] is based on random shifts of the particle three-momentum within a confined range. After each shift, a comparison is made with correlation functions resulting from the assumed model of the space–time distribution and with the single-particle spectra that should remain unchanged. The shift is kept if the χ^2 -test shows better agreement. The process is iterated until satisfactory agreement is achieved. In order to construct the correlation function, a reference sample is made by mixing particles from consecutive events. Such a method has an important impact on the simulations, when at least two events must be processed simultaneously.

Some specific features of this approach are important for practical use:

- The HBT processor can simultaneously generate correlations of up to two particle types (e.g. positive and negative pions). Correlations of other particles can be added subsequently.
- The form of the correlation function has to be parameterized analytically. One and three dimensional parameterizations are possible.
- A static source is usually assumed. Dynamical effects, related to expansion or flow, can be simulated in a stepwise form by repeating simulations for different values of the space–time parameters associated with different kinematic intervals.

- Coulomb effects may be introduced by one of three approaches: Gamow factor, experimentally modified Gamow correction and integrated Coulomb wave functions for discrete values of the source radii.
- Strong interactions are not implemented.

The detailed description of the HBT processor can be found elsewhere [34].

Flow afterburner

Azimuthal anisotropies, especially elliptic flow, carry unique information about collective phenomena and consequently are important for the study of heavy-ion collisions. Additional information can be obtained studying different heavy-ion observables, especially jets, relative to the event plane. Therefore it is necessary to evaluate the capability of ALICE to reconstruct the event plane and study elliptic flow.

Since a well understood microscopic description of the flow effect is not available so far, it cannot be correctly simulated by microscopic event generators. Therefore, in order to generate events with flow, the user has to use event generators based on macroscopic models, like GeVSim [198] or an afterburner which can generate flow on top of events generated by event generators based on the microscopic description of the interaction. In the AliRoot framework such a flow afterburner is implemented.

The algorithm to apply azimuthal correlation consists in shifting the azimuthal coordinates of the particles. The transformation is given by [35]:

$$\varphi \to \varphi' = \varphi + \Delta\varphi$$

$$\Delta\varphi = \sum_n -\frac{2}{n} v_n\left(p_t, y\right) \sin n \times \left(\varphi - \phi\right)$$

where $v_n\left(p_t, y\right)$ is the flow coefficient to be obtained, n is the harmonic number and ϕ is the event-plane angle. Note that the algorithm is deterministic and does not contain any random number generation.

The value of the flow coefficient can either be constant or parameterized as a function of transverse momentum and rapidity. Two parameterizations of elliptic flow are provided as in GeVSim.

```
AliGenGeVSim* gener = new AliGenGeVSim(0);

mult = 2000; // Mult is the number of charged particles in |eta| < 0.5
vn = 0.01; // Vn

Float_t sigma_eta = 2.75; // Sigma of the Gaussian dN/dEta
Float_t etamax  = 7.00; // Maximum eta

// Scale from multiplicity in |eta| < 0.5 to |eta| < |etamax|
Float_t mm = mult * (TMath::Erf(etamax/sigma_eta/sqrt(2.)) /
TMath::Erf(0.5/sigma_eta/sqrt(2.)));

// Scale from charged to total multiplicity
mm *= 1.587;
```

```
// Define particles

// 78% Pions (26% pi+, 26% pi-, 26% p0)    T = 250 MeV
AliGeVSimParticle *pp =
new AliGeVSimParticle(kPiPlus, 1, 0.26 * mm, 0.25, sigma_eta) ;
AliGeVSimParticle *pm =
new AliGeVSimParticle(kPiMinus, 1, 0.26 * mm, 0.25, sigma_eta) ;
AliGeVSimParticle *p0 =
new AliGeVSimParticle(kPi0,  1, 0.26 * mm, 0.25, sigma_eta) ;

// 12% Kaons (3% K0short, 3% K0long, 3% K+, 3% K-) T = 300 MeV
AliGeVSimParticle *ks =
new AliGeVSimParticle(kK0Short, 1, 0.03 * mm, 0.30, sigma_eta) ;
AliGeVSimParticle *kl =
new AliGeVSimParticle(kK0Long, 1, 0.03 * mm, 0.30, sigma_eta) ;
AliGeVSimParticle *kp =
new AliGeVSimParticle(kKPlus, 1, 0.03 * mm, 0.30, sigma_eta) ;
AliGeVSimParticle *km =
new AliGeVSimParticle(kKMinus, 1, 0.03 * mm, 0.30, sigma_eta) ;

// 10% Protons / Neutrons (5% Protons, 5% Neutrons) T = 250 MeV
AliGeVSimParticle *pr =
new AliGeVSimParticle(kProton, 1, 0.05 * mm, 0.25, sigma_eta) ;
AliGeVSimParticle *ne =
new AliGeVSimParticle(kNeutron, 1, 0.05 * mm, 0.25, sigma_eta) ;

// Set Elliptic Flow properties

Float_t pTsaturation = 2. ;

pp->SetEllipticParam(vn,pTsaturation,0.) ;
pm->SetEllipticParam(vn,pTsaturation,0.) ;
p0->SetEllipticParam(vn,pTsaturation,0.) ;
pr->SetEllipticParam(vn,pTsaturation,0.) ;
ne->SetEllipticParam(vn,pTsaturation,0.) ;
ks->SetEllipticParam(vn,pTsaturation,0.) ;
kl->SetEllipticParam(vn,pTsaturation,0.) ;
kp->SetEllipticParam(vn,pTsaturation,0.) ;
km->SetEllipticParam(vn,pTsaturation,0.) ;

// Set Direct Flow properties

pp->SetDirectedParam(vn,1.0,0.) ;
pm->SetDirectedParam(vn,1.0,0.) ;
p0->SetDirectedParam(vn,1.0,0.) ;
pr->SetDirectedParam(vn,1.0,0.) ;
ne->SetDirectedParam(vn,1.0,0.) ;
ks->SetDirectedParam(vn,1.0,0.) ;
kl->SetDirectedParam(vn,1.0,0.) ;
kp->SetDirectedParam(vn,1.0,0.) ;
km->SetDirectedParam(vn,1.0,0.) ;
```

```
// Add particles to the list

gener->AddParticleType(pp) ;
gener->AddParticleType(pm) ;
gener->AddParticleType(p0) ;
gener->AddParticleType(pr) ;
gener->AddParticleType(ne) ;
gener->AddParticleType(ks) ;
gener->AddParticleType(kl) ;
gener->AddParticleType(kp) ;
gener->AddParticleType(km) ;

// Random Ev.Plane

TF1 *rpa = new TF1("gevsimPsiRndm","1", 0, 360);

gener->SetPtRange(0., 9.) ; // Used for bin size in numerical
integration
gener->SetPhiRange(0, 360);

gener->SetOrigin(0, 0, 0); // vertex position
gener->SetSigma(0, 0, 5.3); // Sigma in (X,Y,Z) (cm) on IP position
gener->SetCutVertexZ(1.);  // Truncate at 1 sigma
gener->SetVertexSmear(kPerEvent);
gener->SetTrackingFlag(1);
gener->Init();
```

Generator for e^+e^- pairs in Pb–Pb collisions

In addition to strong interactions of heavy ions in central and peripheral collisions, ultra-peripheral collisions of ions give rise to coherent, mainly electromagnetic interactions among which the dominant process is the (multiple) e^+e^- -pair production [36]:

$$AA \rightarrow AA + n\left(e^+e^-\right)$$

where n is the pair multiplicity. Most electron–positron pairs are produced into the very forward direction escaping the experiment. However, for Pb–Pb collisions at the LHC the cross-section of this process, about 230 kb, is enormous. A sizable fraction of pairs produced with large-momentum transfer can contribute to the hit rate in the forward detectors increasing the occupancy or trigger rate. In order to study this effect, an event generator for e^+e^- -pair production has been implemented in the AliRoot framework [37]. The class **TEpEmGen** is a realisation of the **TGenerator** interface for external generators and wraps the FORTRAN code used to calculate the differential cross-section. **AliGenEpEmv1** derives from **AliGenerator** and uses the external generator to put the pairs on the AliRoot particle stack.

3.5.4.6Combination of generators: AliGenCocktail

Different generators can be combined together so that each one adds the particles it has generated to the event stack via the **AliGenCocktail** class.

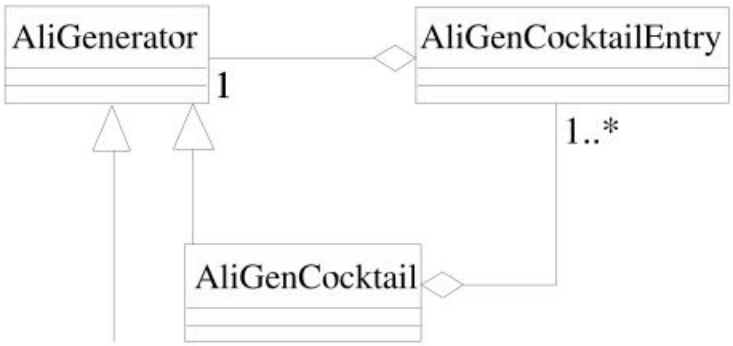


*Figure 6: The **AliGenCocktail** generator is a realization of **AliGenerator** which does not generate particles itself but delegates this task to a list of objects of type **AliGenerator** that can be connected as entries (**AliGenCocktailEntry**) at run-time. In this way, different physics channels can be combined in one event.*

Here is an example of cocktail, used for studies in the TRD detector:

```
// The cocktail generator
AliGenCocktail *gener = new AliGenCocktail();

// Phi meson (10 particles)
AliGenParam *phi =
new AliGenParam(10,new AliGenMUONlib(),AliGenMUONlib::kPhi,"Vogt
PbPb");
phi->SetPtRange(0, 100);
phi->SetYRange(-1., +1.);
phi->SetForceDecay(kDiElectron);

// Omega meson (10 particles)
AliGenParam *omega =
new AliGenParam(10,new AliGenMUONlib(),AliGenMUONlib::kOmega,"Vogt
PbPb");
omega->SetPtRange(0, 100);
omega->SetYRange(-1., +1.);
omega->SetForceDecay(kDiElectron);

// J/psi
AliGenParam *jpsi = new AliGenParam(10,new AliGenMUONlib(),
AliGenMUONlib::kJpsiFamily,"Vogt PbPb");
jpsi->SetPtRange(0, 100);
jpsi->SetYRange(-1., +1.);
jpsi->SetForceDecay(kDiElectron);
```

```
// Upsilon family
AliGenParam *ups = new AliGenParam(10,new AliGenMUONlib(),
AliGenMUONlib::kUpsilonFamily,"Vogt PbPb");
ups->SetPtRange(0, 100);
ups->SetYRange(-1., +1.);
ups->SetForceDecay(kDiElectron);

// Open charm particles
AliGenParam *charm = new AliGenParam(10,new AliGenMUONlib(),
AliGenMUONlib::kCharm,"central");
 charm->SetPtRange(0, 100);
charm->SetYRange(-1.5, +1.5);
charm->SetForceDecay(kSemiElectronic);

// Beauty particles: semi-electronic decays
AliGenParam *beauty = new AliGenParam(10,new AliGenMUONlib(),
AliGenMUONlib::kBeauty,"central");
beauty->SetPtRange(0, 100);
beauty->SetYRange(-1.5, +1.5);
beauty->SetForceDecay(kSemiElectronic);

// Beauty particles to J/psi ee
AliGenParam *beautyJ = new AliGenParam(10, new AliGenMUONlib(),
AliGenMUONlib::kBeauty,"central");
beautyJ->SetPtRange(0, 100);
beautyJ->SetYRange(-1.5, +1.5);
beautyJ->SetForceDecay(kBJpsiDiElectron);

// Adding all the components of the cocktail
gener->AddGenerator(phi,"Phi",1);
gener->AddGenerator(omega,"Omega",1);
gener->AddGenerator(jpsi,"J/psi",1);
gener->AddGenerator(ups,"Upsilon",1);
gener->AddGenerator(charm,"Charm",1);
gener->AddGenerator(beauty,"Beauty",1);
gener->AddGenerator(beautyJ,"J/Psi from Beauty",1);

// Settings, common for all components
gener->SetOrigin(0, 0, 0); // vertex position
gener->SetSigma(0, 0, 5.3); // Sigma in (X,Y,Z) (cm) on IP position
gener->SetCutVertexZ(1.);  // Truncate at 1 sigma
gener->SetVertexSmear(kPerEvent);
gener->SetTrackingFlag(1);
gener->Init();
```

3.5.5 Particle transport

3.5.5.1TGeo essential information

A detailed description of the ROOT geometry package is available in the ROOT User Guide [38]. Several examples can be found in $ROOTSYS/tutorials, among them assembly.C, csgdemo.C, geodemo.C, nucleus.C, rootgeom.C, etc. Here we show a simple usage for export/import of the ALICE geometry and for check for overlaps and extrusions:

```
aliroot
 root [0] gAlice->Init()
 root [1] gGeoManager->Export("geometry.root")
 root [2] .q
 aliroot
 root [0] TGeoManager::Import("geometry.root")
 root [1] gGeoManager->CheckOverlaps()
 root [2] gGeoManager->PrintOverlaps()
 root [3] new TBrowser
 # Now you can navigate in Geometry->Illegal overlaps
 # and draw each overlap (double click on it)
```

3.5.5.2Visualization

Below we show an example of VZERO visualization using the ROOT geometry package:

```
aliroot
 root [0] gAlice->Init()
 root [1] TGeoVolume *top = gGeoManager->GetMasterVolume()
 root [2] Int_t nd = top->GetNdaughters()
 root [3] for (Int_t i=0; i<nd; i++) top->GetNode(i)->GetVolume()-
>InvisibleAll()
 root [4] TGeoVolume *v0ri = gGeoManager->GetVolume("V0RI")
 root [5] TGeoVolume *v0le = gGeoManager->GetVolume("V0LE")
 root [6] v0ri->SetVisibility(kTRUE);
 root [7] v0ri->VisibleDaughters(kTRUE);
 root [8] v0le->SetVisibility(kTRUE);
 root [9] v0le->VisibleDaughters(kTRUE);
 root [10] top->Draw();
```

3.5.5.3Particle decays

We use Pythia to generate one-particle decays during the transport. The default decay channels can be seen in the following way:

```
aliroot
 root [0] AliPythia * py = AliPythia::Instance()
```

```
root [1] py->Pylist(12); >> decay.list
```

The file decay.list will contain the list of particles decays available in Pythia. Now, if we want to force the decay $\Lambda^0 \to p\pi^-$, the following lines should be included in the Config.C before we register the decayer:

```
AliPythia * py = AliPythia::Instance();
 py->SetMDME(1059,1,0);
 py->SetMDME(1060,1,0);
 py->SetMDME(1061,1,0);
```

where 1059,1060 and 1061 are the indexes of the decay channel (from decay.list above) we want to switch off.

3.5.5.4Examples

Fast simulation

This example is taken from the macro $ALICE_ROOT/FASTSIM/fastGen.C. It shows how one can create a kinematics tree which later can be used as input for the particle transport. A simple selection of events with high multiplicity is implemented.

```
void fastGen(Int_t nev = 1, char* filename = "galice.root")
{
  // Run loader
  AliRunLoader* rl = AliRunLoader::Open
("galice.root","FASTRUN","recreate");
  rl->SetCompressionLevel(2);
  rl->SetNumberOfEventsPerFile(nev);
  rl->LoadKinematics("RECREATE");
  rl->MakeTree("E");
  gAlice->SetRunLoader(rl);

  //  Create stack
  rl->MakeStack();
  AliStack* stack      = rl->Stack();

  //  Header
  AliHeader* header = rl->GetHeader();

  //  Generator
  AliGenPythia *gener = new AliGenPythia(-1);
  gener->SetMomentumRange(0,999999);
  gener->SetProcess(kPyMb);
  gener->SetEnergyCMS(14000.);
  gener->SetThetaRange(45, 135);
  gener->SetPtRange(0., 1000.);
```

```
gener->SetStack(stack);
gener->Init();
rl->CdGAFile();
//
//                        Event Loop
//
for (Int_t iev = 0; iev < nev; iev++) {

  //  Initialize event
  header->Reset(0,iev);
  rl->SetEventNumber(iev);
  stack->Reset();
  rl->MakeTree("K");

  //  Generate event
  Int_t nprim = 0;
  Int_t ntrial = 0;
  Int_t minmult = 1000;

  while(nprim<minmult) {
    // Selection of events with multiplicity
    // bigger than "minmult"
    stack->Reset();
    gener->Generate();
    ntrial++;
    nprim = stack->GetNprimary();

  }
  cout << "Number of particles " << nprim << endl;
  cout << "Number of trials " << ntrial << endl;

  //  Finish event
  header->SetNprimary(stack->GetNprimary());
  header->SetNtrack(stack->GetNtrack());

  //      I/O
  stack->FinishEvent();
  header->SetStack(stack);
  rl->TreeE()->Fill();
  rl->WriteKinematics("OVERWRITE");

} // event loop

//                         Termination
//  Generator
gener->FinishRun();
//  Stack
stack->FinishRun();
//  Write file
rl->WriteHeader("OVERWRITE");
gener->Write();
rl->Write();
```

```
}
```

Reading of kinematics tree as input for the particle transport

We suppose that the macro fastGen.C above has been used to generate the corresponding sent of files: galice.root and Kinematics.root, and that they are stored in a separate subdirectory, for example kine. Then the following code in Config.C will read the set of files and put them in the stack for transport:

```
AliGenExtFile *gener = new AliGenExtFile(-1);

 gener->SetMomentumRange(0,14000);
 gener->SetPhiRange(0.,360.);
 gener->SetThetaRange(45,135);
 gener->SetYRange(-10,10);
 gener->SetOrigin(0, 0, 0); //vertex position
 gener->SetSigma(0, 0, 5.3); //Sigma in (X,Y,Z) (cm) on IP position

 AliGenReaderTreeK * reader = new AliGenReaderTreeK();
 reader->SetFileName("../galice.root");

 gener->SetReader(reader);
 gener->SetTrackingFlag(1);

 gener->Init();
```

Usage of different generators

Numerous examples are available in $ALICE_ROOT/macros/Config_gener.C. The corresponding part can be extracted and placed in the respective Config.C file.

3.6 Reconstruction

In this section we describe the ALICE reconstruction framework and software.

3.6.1 Reconstruction Framework

This chapter focuses on the reconstruction framework from the (detector) software developers' point of view.

If not otherwise specified, we refer to the "global ALICE coordinate system" [39]. It is a right-handed coordinate system with the z-axis coinciding with the beam-pipe axis pointing away from the muon arm, the y-axis going upward, and its origin defined by the intersection point of the z-axis and the central membrane-plane of TPC.

In the following, we briefly summarize the main conceptual terms of the reconstruction framework (see also section 3.2):

- **Digit**: This is a digitized signal (ADC count) obtained by a sensitive pad of a

detector at a certain time.

- **Cluster**: This is a set of adjacent (in space and/or in time) digits that were presumably generated by the same particle crossing the sensitive element of a detector.
- **Space point** (reconstructed): This is the estimation of the position where a particle crossed the sensitive element of a detector (often, this is done by calculating the centre of gravity of the `cluster').
- **Track** (reconstructed): This is a set of five parameters (such as the curvature and the angles with respect to the coordinate axes) of the particle's trajectory together with the corresponding covariance matrix estimated at a given point in space.

The input to the reconstruction framework are digits in ROOT tree format or raw data format. First, a local reconstruction of clusters is performed in each detector. Then vertexes and tracks are reconstructed and the particle identification is carried on. The output of the reconstruction is the Event Summary Data (ESD). The **AliReconstruction** class provides a simple user interface to the reconstruction framework which is explained in the source code and.

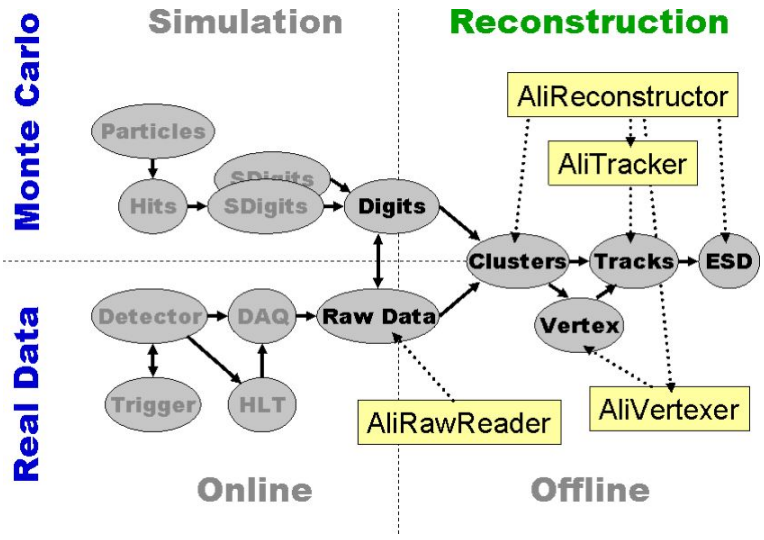


Figure 7: Reconstruction framework

Requirements and Guidelines

The development of the reconstruction framework has been guided by the following requirements and code of practice:

- The prime goal of the reconstruction is to provide the data that is needed for a physics analysis.
- The reconstruction should be aimed at high efficiency, purity and resolution.
- The user should have an easy-to-use interface to extract the required information from the ESD.

- The reconstruction code should be efficient and maintainable.
- The reconstruction should be as flexible as possible. It should be possible to do the reconstruction in one detector even if other detectors are not operational. To achieve such a flexibility each detector module should be able to
    - find tracks starting from seeds provided by another detector (external seeding),
    - find tracks without using information from other detectors (internal seeding),
    - find tracks from external seeds and add tracks from internal seeds
    - and propagate tracks through the detector using the already assigned clusters in inward and outward direction.
- Where it is appropriate, common (base) classes should be used in the different reconstruction modules.
- Interdependencies between the reconstruction modules should be minimized. If possible the exchange of information between detectors should be done via a common track class.
- The chain of reconstruction program(s) should be callable and steerable in an easy way.
- There should be no assumptions on the structure or names of files or on the number or order of events.
- Each class, data member and method should have a correct, precise and helpful html documentation.

AliReconstructor

The base class **AliReconstructor** defines the interface from the steering class **AliReconstruction** to the detector-specific reconstruction code. For each detector, there is a derived reconstructor class. The user can set options for each reconstructor as string parameter that is accessible inside the reconstructor via the method GetOption.

The detector specific reconstructors are created via plug-ins. Therefore they must have a default constructor. If no plug-in handler is defined by the user (in .rootrc), it is assumed that the name of the reconstructor for detector <DET> is Ali<DET>Reconstructor and that it is located in the library lib<DET>rec.so (or lib<DET>.so in case the libraries of the detector have not been split and are all bundled in a single one).

Input Data

If the input data is provided in format of ROOT trees, either the loaders or directly the trees are used to access the digits. In case of raw data input, the digits are accessed via a raw reader.

If a galice.root file exists, the run loader will be retrieved from it. Otherwise the run loader and the headers will be created from the raw data. The reconstruction cannot

work if there is no galice.root file and no raw data input.

Output Data

The clusters (rec. points) are considered intermediate output and are stored in ROOT trees handled by the loaders. The final output of the reconstruction is a tree with objects of type **AliESD** stored in the file AliESDs.root. This Event Summary Data (ESD) contains lists of reconstructed tracks/particles and global event properties. The detailed description of the ESD can be found in section ESD.

Local Reconstruction (Clusterization)

The first step of the reconstruction is the so-called “local reconstruction”. It is executed for each detector separately and without exchanging information with other detectors. Usually the clusterization is done in this step.

The local reconstruction is invoked via the method Reconstruct of the reconstructor object. Each detector reconstructor runs the local reconstruction for all events. The local reconstruction method is only called if the method HasLocalReconstruction of the reconstructor returns kTRUE.

Instead of running the local reconstruction directly on raw data, it is possible to first convert the raw data digits into a digits tree and then to call the Reconstruct method with a tree as input parameter. This conversion is done by the method ConvertDigits. The reconstructor has to announce that it can convert the raw data digits by returning kTRUE in the method HasDigitConversion.

Vertexing

The current reconstruction of the primary-vertex position in ALICE is done using the information provided by the silicon pixel detectors, which constitute the two innermost layers of the ITS.

The algorithm starts by looking at the distribution of the z-coordinates of the reconstructed space points in the first pixel layers. At a vertex having z-coordinate z_{true}=0, the distribution is symmetric and its centroid (z_{cen}) is very close to the nominal vertex position. When the primary vertex is moved along the z-axis, an increasing fraction of hits will be lost and the centroid of the distribution no longer gives the primary vertex position. However, for primary vertex locations not too far from z_{true}=0 (up to about 12 cm), the centroid of the distribution is still correlated to the true vertex position. The saturation effect at large z_{true} values of the vertex position z_{true}=12–15 cm) is, however, not critical, since this procedure is only meant to find a rough vertex position, in order to introduce some cut along z.

To find the final vertex position, the correlation between the points z_1, z_2 in the two layers was considered. More details and performance studies are available in [40].

A vertexer object derived from **AliVertexer** reconstructs the primary vertex. After the local reconstruction has been done for all detectors, the vertexer method FindVertexForCurrentEvent is called for each event. It returns a pointer to a vertex object of type **AliESDVertex**.

The vertexer object is created by the method CreateVertexer of the reconstructor. So

far, only the ITS is taken into account to determine the primary vertex (**AliITSVertexerZ** class).

The precision of the primary vertex reconstruction in the bending plane required for the reconstruction of D and B mesons in p–p events can be achieved only after the tracking is done. The method is implemented in **AliITSVertexerTracks**. It is called as a second estimation of the primary vertex. The details of the algorithm can be found in Appendix VertexerTracks.

Combined Track Reconstruction

The combined track reconstruction tries to accumulate the information from different detectors in order to optimize the track reconstruction performance. The result of this is stored in the combined track objects. The **AliESDTrack** class also provides the possibility to exchange information between detectors without introducing dependencies between the reconstruction modules. This is achieved by using just integer indexes pointing to the specific track objects, which allows the retrieval of the information as needed. The list of combined tracks can be kept in memory and passed from one reconstruction module to another. The storage of the combined tracks should be done in the standard way.

The classes responsible for the reconstruction of tracks are derived from **AliTracker**. They are created by the method CreateTracker of the reconstructors. The reconstructed position of the primary vertex is made available to them via the method SetVertex. Before the track reconstruction in a detector starts, clusters are loaded from the clusters tree by means of the method LoadClusters. After the track reconstruction, the clusters are unloaded by the method UnloadClusters.

The track reconstruction (in the barrel part) is done in three passes. The first pass consists of a track finding and fitting in inward direction in TPC and then in the ITS. The virtual method Clusters2Tracks (belonging to the class **AliTracker**) provides the interface to this pass. The method for the next pass is PropagateBack. It does the track reconstruction in outward direction and is invoked for all detectors starting with the ITS. The last pass is the track refit in inward direction in order to get the track parameters at the vertex. The corresponding method RefitInward is called for TRD, TPC and ITS. All three track-reconstruction methods have an **AliESD** object as argument that is used to exchange track information between detectors without introducing code dependencies between the detector trackers.

Depending on the way the information is used, the tracking methods can be divided into two large groups: global methods and local methods. Each group has advantages and disadvantages.

With the global methods, all track measurements are treated simultaneously and the decision to include or exclude a measurement is taken when all the information about the track is known. Typical algorithms belonging to this class are combinatorial methods, Hough transform, templates, and conformal mappings. The advantages are the stability with respect to noise and mis-measurements, and the possibility to operate directly on the raw data. On the other hand, these methods require a precise global track model. Sometimes, such a track model is unknown or does not even exist

because of stochastic processes (energy losses, multiple scattering), non-uniformity of the magnetic field etc. In ALICE, global tracking methods are being extensively used in the High-Level Trigger (HLT) software. There, we are mostly interested in the reconstruction of the high-momentum tracks, the required precision is not crucial, but the speed of the calculations is of great importance.

Local methods do not require the knowledge of the global track model. The track parameters are always estimated “locally” at a given point in space. The decision to accept or reject a measurement is made using either the local information or the information coming from the previous `history' of this track. With these methods, all the local track peculiarities (stochastic physics processes, magnetic fields, detector geometry) can naturally be accounted for. Unfortunately, local methods rely on sophisticated space point reconstruction algorithms (including unfolding of overlapped clusters). They are sensitive to noise, wrong or displaced measurements and the precision of space point error parameterization. The most advanced kind of local track-finding methods is Kalman filtering which was introduced by P. Billoir in 1983 [41].

When applied to the track reconstruction problem, the Kalman-filter approach shows many attractive properties:

- It is a method for simultaneous track recognition and fitting.
- There is a possibility to reject incorrect space points “on the fly” during a single tracking pass. These incorrect points can appear as a consequence of the imperfection of the cluster finder or they may appear due to noise or they may be points from other tracks accidentally captured in the list of points to be associated with the track under consideration. In the other tracking methods, one usually needs an additional fitting pass to get rid of incorrectly assigned points.
- In case of substantial multiple scattering, track measurements are correlated and therefore large matrices (of the size of the number of measured points) need to be inverted during a global fit. In the Kalman-filter procedure we only have to manipulate up to 5 x 5 matrices (although as many times as we have measured space points), which is much faster.
- One can handle multiple scattering and energy losses in a simpler way than in the case of global methods. At each step, the material budget can be calculated and the mean correction computed accordingly.
- It is a natural way to find the extrapolation of a track from one detector to another (for example from the TPC to the ITS or to the TRD).

In ALICE we require good track-finding efficiency and reconstruction precision for track down to p_t=100 MeV/c. Some of the ALICE tracking detectors (ITS, TRD) have a significant material budget. Under such conditions, one can not neglect the energy losses or the multiple scattering in the reconstruction. There are also rather big dead zones between the tracking detectors, which complicate finding the continuation of the same track. For all these reasons, it is the Kalman-filtering approach that has been our choice for the offline reconstruction since 1994.

3.6.1.1General tracking strategy

All parts of the reconstruction software for the ALICE central tracking detectors (the ITS, TPC and the TRD) follow the same convention for the coordinate system used. All clusters and tracks are always expressed in some local coordinate system related to a given sub-detector (TPC sector, ITS module etc). This local coordinate system is defined as follows:

- It is a right handed-Cartesian coordinate system;
- Its origin and the z-axis coincide with that of the global ALICE coordinate system;
- The x-axis is perpendicular to the sub-detector's “sensitive plane” (TPC pad row, ITS ladder etc).

Such a choice reflects the symmetry of the ALICE set-up and therefore simplifies the reconstruction equations. It also enables the fastest possible transformations from a local coordinate system to the global one and back again, since these transformations become single rotations around the z-axis.

The reconstruction begins with cluster finding in all of the ALICE central detectors (ITS, TPC, TRD, TOF, HMPID and PHOS). Using the clusters reconstructed at the two pixel layers of the ITS, the position of the primary vertex is estimated and the track finding starts. As described later, cluster-finding, as well as track-finding procedures performed in the detectors have some different detector-specific features. Moreover, within a given detector, on account of high occupancy and a big number of overlapping clusters, cluster finding and track finding are not completely independent: the number and positions of the clusters are completely determined only at the track-finding step.

The general tracking strategy is the following: We start from our best tracker device, i.e. the TPC, and there from the outer radius where the track density is minimal. First, the track candidates (“seeds”) are found. Because of the small number of clusters assigned to a seed, the precision of its parameters is not sufficient to safely extrapolate it outwards to the other detectors. Instead, the tracking stays within the TPC and proceeds towards the smaller TPC radii. Whenever possible, new clusters are associated with a track candidate at each step of the Kalman filter, if they are within a given distance from the track prolongation, and the track parameters are more and more refined. When all of the seeds are extrapolated to the inner limit of the TPC, we proceed with the ITS. The ITS tracker tries to prolong the TPC tracks as close as possible to the primary vertex. On the way to the primary vertex, the tracks are assigned additional, precisely reconstructed ITS clusters, which also improves the estimation of the track parameters.

After all the track candidates from the TPC have been assigned their clusters in the ITS, a special ITS stand-alone tracking procedure is applied to the rest of the ITS clusters. This procedure tries to recover those tracks that were not found in the TPC because of the pt cut-off, dead zones between the TPC sectors, or decays.

At this point, the tracking is restarted from the vertex back to the outer layer of the ITS and then continued towards the outer wall of the TPC. For the track that was labelled by

the ITS tracker as potentially primary, several particle-mass-dependent, time-of-flight hypotheses are calculated. These hypotheses are then used for the particle identification (PID) within the TOF detector. Once the outer radius of the TPC is reached, the precision of the estimated track parameters is sufficient to extrapolate the tracks to the TRD, TOF, HMPID and PHOS detectors. Tracking in the TRD is done in a similar way to that in the TPC. Tracks are followed till the outer wall of the TRD and the assigned clusters improve the momentum resolution further. Next, the tracks are extrapolated to the TOF, HMPID and PHOS, where they acquire the PID information. Finally, all the tracks are refitted with the Kalman filter backwards to the primary vertex (or to the innermost possible radius, in the case of the secondary tracks). This gives the most precise information about the track parameters at the point where the track appeared.

The tracks that passed the final refit towards the primary vertex are used for the secondary vertex (V^0, cascade, kink) reconstruction. There is also an option to reconstruct the secondary vertexes “on the fly” during the tracking itself. The potential advantage of such a possibility is that the tracks coming from a secondary vertex candidate are not extrapolated beyond the vertex, thus minimizing the risk of picking up a wrong track prolongation. This option is currently under investigation.

The reconstructed tracks (together with the PID information), kink, V^0 and cascade particle decays are then stored in the Event Summary Data (ESD).

More details about the reconstruction algorithms can be found in Chapter 5 of the ALICE Physics Performance Report [198].

Filling of ESD

After the tracks have been reconstructed and stored in the **AliESD** object, further information is added to the ESD. For each detector the method FillESD of the reconstructor is called. Inside this method e.g. V^0s are reconstructed or particles are identified (PID). For the PID, a Bayesian approach is used (see Appendix 8.2. The constants and some functions that are used for the PID are defined in the class **AliPID**.

Monitoring of Performance

For the monitoring of the track reconstruction performance, the classes **AliTrackReference** are used. Objects of the second type of class are created during the reconstruction at the same locations as the **AliTrackReference** objects. So the reconstructed tracks can be easily compared with the simulated particles. This allows studying and monitoring the performance of the track reconstruction in detail. The creation of the objects used for the comparison should not interfere with the reconstruction algorithm and can be switched on or off.

Several “comparison” macros permit to monitor the efficiency and the resolution of the tracking. Here is a typical usage (the simulation and the reconstruction have been done in advance):

```
aliroot
 root [0] gSystem->SetIncludePath("-I$ROOTSYS/include \
 -I$ALICE_ROOT/include \
```

```
-I$ALICE_ROOT/TPC \
-I$ALICE_ROOT/ITS \
-I$ALICE_ROOT/TOF")
root [1] .L $ALICE_ROOT/TPC/AliTPCComparison.C++
root [2] .L $ALICE_ROOT/ITS/AliITSComparisonV2.C++
root [3] .L $ALICE_ROOT/TOF/AliTOFComparison.C++
root [4] AliTPCComparison()
root [5] AliITSComparisonV2()
root [6] AliTOFComparison()
```

Another macro can be used to provide a preliminary estimate of the combined acceptance: STEER/CheckESD.C.

Classes

The following classes are used in the reconstruction:

- **AliTrackReference** is used to store the position and the momentum of a simulated particle at given locations of interest (e.g. when the particle enters or exits a detector or it decays). It is used mainly for debugging and tuning of the tracking.
- **AliExternalTrackParams** describes the status of a track at a given point. It contains the track parameters and its covariance matrix. This parameterization is used to exchange tracks between the detectors. A set of functions returning the position and the momentum of tracks in the global coordinate system as well as the track impact parameters are implemented. There is the option to propagate the track to a given radius PropagateTo and Propagate.
- **AliKalmanTrack** (and derived classes) are used to find and fit tracks with the Kalman approach. The **AliKalmanTrack** defines the interfaces and implements some common functionality. The derived classes know about the clusters assigned to the track. They also update the information in an AliESDtrack. An **AliExternalTrackParameters** object can represent the current status of the track during the track reconstruction. The history of the track during reconstruction can be stored in a list of **AliExternalTrackParameters** objects. The **AliKalmanTrack** defines the methods:
  - **Double_t GetDCA(...)** Returns the distance of closest approach between this track and the track passed as the argument.
  - **Double_t MeanMaterialBudget(...)** Calculate the mean material budget and material properties between two points.
- **AliTracker** and subclasses: The **AliTracker** is the base class for all the trackers in the different detectors. It defines the interface required to find and propagate tracks. The actual implementation is done in the derived classes.
- **AliESDTrack** combines the information about a track from different detectors. It contains the current status of the track (**AliExternalTrackParameters**) and it has (non-persistent) pointers to the individual **AliKalmanTrack** objects from each detector that contribute to the track. It contains as well detector specific

quantities like the number or bit pattern of assigned clusters, dE/dx, χ^2, etc.. and it can calculate a conditional probability for a given mixture of particle species following the Bayesian approach. It also defines a track label pointing to the corresponding simulated particle in case of Monte Carlo. The combined track objects are the basis for a physics analysis.

Example

The example below shows reconstruction with non-uniform magnetic field (the simulation is also done with non-uniform magnetic field by adding the following line in the Config.C: field→SetL3ConstField(1)). Only

the barrel detectors are reconstructed, a specific TOF reconstruction

has been requested, and the RAW data have been used:

```
void rec() {
 AliReconstruction reco;

 reco.SetRunReconstruction("ITS TPC TRD TOF");
 reco.SetUniformFieldTracking(0);
 reco.SetInput("raw.root");

 reco.Run();
 }
```

3.6.2 Event summary data

The classes that are needed to process and analyze the ESD are packaged in a standalone library (libESD.so) which can be used independently from the AliRoot framework. Inside each ESD object, the data is stored in polymorphic containers filled with reconstructed tracks, neutral particles, etc. The main class is **AliESD**, which contains all the information needed during the physics analysis:

- fields to identify the event, such as event number, run number, time stamp, type of event, trigger type (mask), trigger cluster (mask), version of reconstruction, etc.;
- reconstructed ZDC energies and number of participants;
- primary vertex information: vertex z-position estimated by the T0, primary vertex estimated by the SPD, primary vertex estimated using ESD tracks;
- tracklet multiplicity;
- interaction time estimated by the T0 together with additional time and amplitude information from T0;
- array of ESD tracks;
- arrays of HLT tracks, both from the conformal mapping and from the Hough transform reconstruction;

- array of MUON tracks;
- array of PMD tracks;
- array of TRD ESD tracks (triggered);
- arrays of reconstructed V^0 vertexes, cascade decays and kinks;
- array of calorimeter clusters for PHOS/EMCAL;
- indexes of the information from PHOS and EMCAL detectors in the array above.

3.7Analysis

3.7.1 Introduction

The analysis of experimental data is the final stage of event processing, and it is usually repeated many times. Analysis is a very diverse activity, where the goals of each particular analysis pass may differ significantly.

The ALICE detector [197] is optimized for the reconstruction and analysis of heavy-ion collisions. In addition, ALICE has a broad physics programme devoted to p–p and p–A interactions.

The Physics Board coordinates data analysis via the Physics Working Groups (PWGs). At present, the following PWG have started their activity:

- PWG0 *first physics*;
- PWG1 *detector performance*;
- PWG2 *global event characteristics*: particle multiplicity, centrality, energy density, nuclear stopping;
  soft physics: chemical composition (particle and resonance production, particle ratios and spectra, strangeness enhancement), reaction dynamics (transverse and elliptic flow, HBT correlations, event-by-event dynamical fluctuations);
- PWG3 *heavy flavors*: quarkonia, open charm and beauty production.
- PWG4 *hard probes*: jets, direct photons;

Each PWG has corresponding module in AliRoot (PWG0 – PWG4). CVS administrators manage the code.

The p–p and p–A programme will provide, on the one hand, reference points for comparison with heavy ions. On the other hand, ALICE will also pursue genuine and detailed p–p studies. Some quantities, in particular the global characteristics of interactions, will be measured during the first days of running, exploiting the low-momentum measurement and particle identification capabilities of ALICE.

The ALICE computing framework is described in details in the Computing Technical Design Report [197]. This section is based on Chapter 6 of the document.

The analysis activity

We distinguish two main types of analysis: scheduled analysis and chaotic analysis. They differ in their data access pattern, in the storage and registration of the results, and in the frequency of changes in the analysis code (more details are available below).

In the ALICE computing model, the analysis starts from the Event Summary Data (ESD). These are produced during the reconstruction step and contain all the information for analysis. The size of the ESD is about one order of magnitude lower than the corresponding raw data. The analysis tasks produce Analysis Object Data (AOD), specific to a given set of physics objectives. Further passes for the specific analysis activity can be performed on the AODs, until the selection parameter or algorithms are changed.

A typical data analysis task usually requires processing of selected sets of events. The selection is based on the event topology and characteristics, and is done by querying the tag database. The tags represent physics quantities which characterize each run and event, and permit fast selection. They are created after the reconstruction and contain also the unique identifier of the ESD file. A typical query, when translated into natural language, could look like “Give me all the events with impact parameter in <range> containing jet candidates with energy larger than <threshold>”. This results in a list of events and file identifiers to be used in the consecutive event loop.

The next step of a typical analysis consists of a loop over all the events in the list and calculation of the physics quantities of interest. Usually, for each event, there is a set of embedded loops over the reconstructed entities such as tracks, V^0 candidates, neutral clusters, etc., the main goal of which is to select the signal candidates. Inside each loop, a number of criteria (cuts) are applied to reject the background combinations and to select the signal ones. The cuts can be based on geometrical quantities such as impact parameters of the tracks with respect to the primary vertex, distance between the cluster and the closest track, distance-of-closest approach between the tracks, angle between the momentum vector of the particle combination and the line connecting the production and decay vertexes. They can also be based on kinematics quantities, such as momentum ratios, minimal and maximal transverse momentum, angles in the rest frame of the particle combination. Particle identification criteria are also among the most common selection criteria.

The optimization of the selection criteria is one of the most important parts of the analysis. The goal is to maximize the signal-to-background ratio in case of search tasks, or another ratio (typically $Signal/\sqrt{Signal + Background}$) in case of measurement of a given property. Usually, this optimization is performed using simulated events where the information from the particle generator is available.

After optimization of the selection criteria, one has to take into account the combined acceptance of the detector. This is a complex, analysis-specific quantity which depends on geometrical acceptance, trigger efficiency, decays of particles, reconstruction efficiency, efficiency of the particle identification and of the selection cuts. The components of the combined acceptance are usually parameterized and their product is used to unfold the experimental distributions, or during the simulation of model parameters.

The last part of the analysis usually involves quite complex mathematical treatment, and sophisticated statistical tools. At this point, one may include the correction for systematic effects, the estimation of statistical and systematic errors, etc.

Scheduled analysis

The scheduled analysis typically uses all the available data from a given period and stores and registers the results using Grid middleware. The tag database is updated accordingly. The AOD files generated during the scheduled analysis can be used by several subsequent analyses, or by a class of related physics tasks. The procedure of scheduled analysis is centralized and can be understood as data filtering. The requirements come from the PWGs and are prioritized by the Physics Board, taking into account the available computing and storage resources. The analysis code will be tested in advance and released before the beginning of the data processing.

Each PWG will require some sets of AOD per event, which are specific for one or several analysis tasks. The creation of those AOD sets is managed centrally. The event list of each AOD set will be registered and the access to the AOD files will be granted to all ALICE collaborators. AOD files will be generated at different computing centres and will be stored on the corresponding storage elements. The processing of each file set will thus be done in a distributed way on the Grid. Some of the AOD sets may be so small that they would fit on a single storage element or even on one computer. In this case, the corresponding tools for file replication, available in the ALICE Grid infrastructure, will be used.

Chaotic analysis

The chaotic analysis is focused on a single physics task and is typically based on the filtered data from the scheduled analysis. Each physicist also may access directly large parts of the ESD in order to search for rare events or processes. Usually the user develops the code using a small subsample of data, and changes the algorithms and criteria frequently. The analysis macros and software are tested many times on relatively small data volumes, both experimental and Monte Carlo. In many cases, the output is only a set of histograms. Such a tuning of the analysis code can be done on a local data set or on distributed data using Grid tools. The final version of the analysis will eventually be submitted to the Grid and will access large portions or even the totality of the ESDs. The results may be registered in the Grid file catalogue and used at later stages of the analysis. This activity may or may not be coordinated inside the PWGs, via the definition of priorities. The chaotic analysis is carried out within the computing resources of the physics groups.

3.7.2 Infrastructure tools for distributed analysis

3.7.2.1gShell

The main infrastructure tools for distributed analysis have been described in Chapter 3 of the Computing TDR [197]. The actual middleware is hidden by an interface to the Grid, gShell [[42]], which provides a single working shell. The gShell package contains all the commands a user may need for file catalogue queries, creation of sub-directories in

the user space, registration and removal of files, job submission and process monitoring. The actual Grid middleware is completely transparent to the user.

The gShell overcomes the scalability problem of direct client connections to databases. All clients connect to the gLite [43] API services. This service is implemented as a pool of pre-forked server daemons, which serve single-client requests. The client-server protocol implements a client state, which is represented by a current working directory, a client session ID and time-dependent symmetric cipher on both ends to guarantee privacy and security. The daemons execute client calls with the identity of the connected client.

3.7.2.2PROOF – the Parallel ROOT Facility

The Parallel ROOT Facility (PROOF [44]) has been specially designed and developed to allow the analysis and mining of very large data sets, minimizing response time. It makes use of the inherent parallelism in event data and implements an architecture that optimizes I/O and CPU utilization in heterogeneous clusters with distributed storage. The system provides transparent and interactive access to terabyte-scale data sets. Being part of the ROOT framework, PROOF inherits the benefits of a performing object storage system and a wealth of statistical and visualization tools. The most important design features of PROOF are:

- transparency – no difference between a local ROOT and a remote parallel PROOF session;
- scalability – no implicit limitations on number of computers used in parallel;
- adaptability – the system is able to adapt to variations in the remote environment.

PROOF is based on a multi-tier architecture: the ROOT client session, the PROOF master server, optionally a number of PROOF sub-master servers, and the PROOF worker servers. The user connects from the ROOT session to a master server on a remote cluster, and the master server creates sub-masters and worker servers on all the nodes in the cluster. All workers process queries in parallel and the results are presented to the user as coming from a single server.

PROOF can be run either in a purely interactive way, with the user remaining connected to the master and worker servers and the analysis results being returned to the user's ROOT session for further analysis, or in an `interactive batch' way where the user disconnects from the master and workers (see Figure 8). By reconnecting later to the master server the user can retrieve the analysis results for that particular query. The latter mode is useful for relatively long running queries (several hours) or for submitting many queries at the same time. Both modes will be important for the analysis of ALICE data.

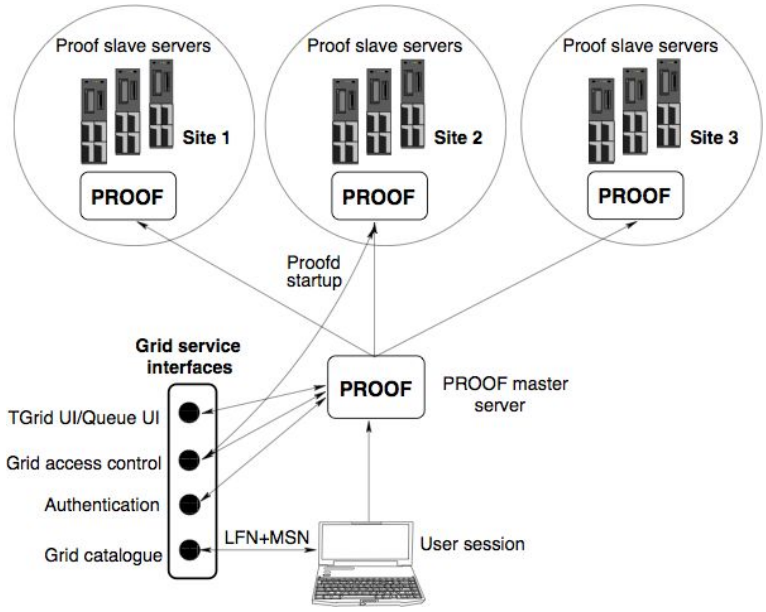


Figure 8: Setup and interaction with the Grid middleware of a user PROOF session distributed over many computing centres.

3.7.3 Analysis tools

This section is devoted to the existing analysis tools in ROOT and AliRoot. As discussed in the introduction, some very broad analysis tasks include the search for rare events (in this case, the physicist tries to maximize the signal-to-background ratio), or measurements where it is important to maximize the signal significance. The tools that provide possibility to apply certain selection criteria, and to find the interesting combinations within a given event are described below. Some of them are very general and used in many different places, for example the statistical tools. Others are specific to a given analysis.

3.7.3.1Statistical tools

Several commonly used statistical tools are available in ROOT [197]. ROOT provides classes for efficient data storage and access, such as trees (**TTree**) and ntuples (**TNtuple**). The ESD information is organized in a tree, where each event is a separate entry. This allows a chain of ESD files to be made and the elaborated selector mechanisms to be used in order to exploit the PROOF services. The tree classes permit easy navigation, selection, browsing, and visualization of the data in the branches.

ROOT also provides histogramming and fitting classes, which are used for the representation of all the one- and multi-dimensional distributions, and for extraction of their fitted parameters. ROOT provides an interface to powerful and robust minimization packages, which can be used directly during special parts of the analysis. A special fitting class allows one to decompose an experimental histogram as a superposition of source histograms.

ROOT also provides a set of sophisticated statistical analysis tools such as principal

component analysis, robust estimator and neural networks. The calculation of confidence levels is provided as well.

Additional statistical functions are included in **TMath**.

3.7.3.2Calculations of kinematics variables

The main ROOT physics classes include 3-vectors, Lorentz vectors and operations such as translation, rotation and boost. The calculations of kinematics variables such as transverse and longitudinal momentum, rapidity, pseudo-rapidity, effective mass, and many others are provided as well.

3.7.3.3Geometrical calculations

There are several classes which can be used for measurement of the primary vertex: **AliITSVertexerZ**, **AliITSVertexerIons**, **AliITSVertexerTracks**, etc. A fast estimation of the z-position can be done by **AliITSVertexerZ**, which works for both lead–lead and proton–proton collisions. A universal tool is provided by **AliITSVertexerTracks**, which calculates the position and covariance matrix of the primary vertex based on a set of tracks, and estimates the χ^2 contribution of each track as well. An iterative procedure can be used to remove the secondary tracks and improve the precision.

Track propagation towards the primary vertex (inward) is provided by **AliESDtrack**.

The secondary vertex reconstruction in case of V^0 is provided by **AliV0vertexer**, and, in case of cascade hyperons, by **AliCascadeVertexer**. **AliITSVertexerTracks** can be used to find secondary vertexes close to the primary one, for example decays of open charm like $D^0 \to K^-\pi^+$ or $D^+ \to K^-\pi^+\pi^+$. All the vertex reconstruction classes also calculate distance-of-closest approach (DCA) between track and vertex.

Calculation of impact parameters with respect to the primary vertex is done during reconstruction, and this information is available in **AliESDtrack**. It is then possible to recalculate the impact parameter during ESD analysis, after an improved determination of the primary vertex position using reconstructed ESD tracks.

3.7.3.4Global event characteristics

The impact parameter of the interaction and the number of participants are estimated from the energy measurements in the ZDC. In addition, the information from the FMD, PMD, and T0 detectors is available. It gives a valuable estimate of the event multiplicity at high rapidities and permits global event characterization. Together with the ZDC information, it improves the determination of the impact parameter, the number of participants, and the number of binary collisions.

The event plane orientation is calculated by the **AliFlowAnalysis** class.

3.7.3.5Comparison between reconstructed and simulated parameters

The comparison between the reconstructed and simulated parameters is an important

part of the analysis. It is the only way to estimate the precision of the reconstruction. Several example macros exist in AliRoot and can be used for this purpose: AliTPCComparison.C, AliITSComparisonV2.C, etc. As a first step in each of these macros, the list of so-called “good tracks” is built. The definition of a good track is explained in detail in the ITS [45] and TPC [46] Technical Design Reports. The essential point is that the track goes through the detector and can be reconstructed. Using the “good tracks”, one then estimates the efficiency of the reconstruction and the resolution.

Another example is specific to the MUON arm: the MUONRecoCheck.C macro compares the reconstructed muon tracks with the simulated ones.

There is also the possibility to calculate directly the resolutions without additional requirements on the initial track. One can use the so-called track label and retrieve the corresponding simulated particle directly from the particle stack (**AliStack**).

3.7.3.6Event mixing

One particular analysis approach in heavy-ion physics is the estimation of the combinatorial background using event mixing. Part of the information (for example the positive tracks) is taken from one event, another part (for example the negative tracks) is taken from a different, but “similar”, event. The event “similarity” is very important, because only in this case, the combinations produced from different events represent the combinatorial background. Under “similar” in the example above we understand events with the same multiplicity of negative tracks. In addition, one may require similar impact parameters of the interactions, rotation of the tracks of the second event to adjust the event plane, etc. The possibility for event mixing is provided in AliRoot by the fact that the ESD is stored in trees, and one can chain and access simultaneously many ESD objects. Then, the first pass would be to order events according to the desired criterion of “similarity” and to use the obtained index for accessing the “similar” events in the embedded analysis loops. An example of event mixing is shown in Figure 9. The background distribution has been obtained using “mixed events”. The signal distribution has been taken directly from the Monte Carlo simulation. The “experimental distribution” has been produced by the analysis macro and decomposed as a superposition of the signal and background histograms.

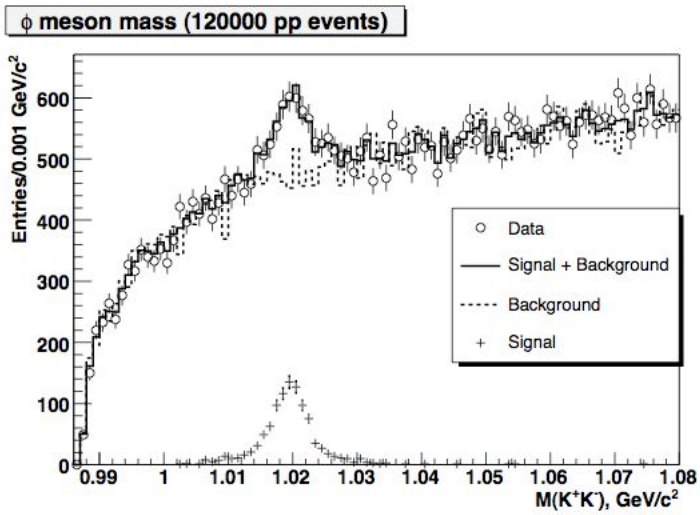


Figure 9: Mass spectrum of the ϕ meson candidates produced inclusively in the proton–proton interactions.

3.7.3.7Analysis of the High-Level Trigger (HLT) data

This is a specific analysis that is needed either in order to adjust the cuts in the HLT code, or to estimate the HLT efficiency and resolution. AliRoot provides a transparent way of doing such an analysis, since the HLT information is stored in the form of ESD objects in a parallel tree. This also helps in the monitoring and visualization of the results of the HLT algorithms.

3.7.3.8EVE – Event Visualization Environment

EVE is composed of:

1. a small application kernel;
2. graphics classes with editors and OpenGL renderers;
3. CINT scripts that extract data, fill graphics classes and register them to the application.

Because the framework is still evolving at this point, some things might not work as expected. The usage is the following:

1. Initialize ALICE environment.
2. Run the “alieve” executable, which should be in the path, and run the alieve_init.C macro, for example:

   To load the first event from the current directory:

```
# alieve alieve_init.C
```

   To load the 5th event from directory /data/my-pp-run:

```
# alieve 'alieve_init.C("/data/my-pp-run", 5)'
```

   Interactively:

```
# alieve
```

```
root[0] .L alieve_init.C
root[1] alieve_init("/somedir")
```

3. Use the GUI or the CINT command-line to invoke further visualization macros.
4. To navigate through the events use macros 'event_next.C' and 'event_prev.C'. These are equivalent to the command-line invocations:

```
root[x] Alieve::gEvent->NextEvent()
```

or

```
root[x] Alieve::gEvent->PrevEvent()
```

The general form to access an event via its number is:

```
root[x] Alieve::gEvent->GotoEvent(<event-number>)
```

See the files in EVE/alice-macros. For specific use cases, these should be edited to suit your needs.

Directory structure

EVE is split into two modules: REVE (ROOT part, not dependent on AliROOT) and ALIEVE (ALICE specific part). For the time being, both modules are kept in AliROOT CVS under the $ALICE_ROOT directory

ALIEVE/ and REVE/	sources
macros/	macros for bootstraping and internal steering
alice-macros/	macros for ALICE visualization
alica-data/	data files used by ALICE macros
test-macros/	macros for tests of specific features; usually one needs to copy and edit them

bin/, Makefile and make_base.inc used for stand-alone build of the packages.

Note that a failed macro-execution can leave CINT in a poorly defined state that prevents further execution of macros. For example:

```
 Exception Reve::Exc_t: Event::Open failed opening ALICE ESDfriend
from
 '/alice-data/coctail_10k/AliESDfriends.root'.


 root [1] Error: Function MUON_geom() is not defined in current
scope :0:
 *** Interpreter error recovered ***
 Error: G__unloadfile() File "/tmp/MUON_geom.C" not loaded :0:
```

“gROOT->Reset()” helps in most of the cases.

3.7.4

-

-

1.

2.

3.8Data input, output and exchange subsystem of AliRoot

This section, in its original form, was published in [47].

A few tens of different data types are present within AliRoot, because hits, summable digits, digits and clusters are characteristic for each sub-detector. Writing all of the event data to a single file causes a number of limitations. Moreover, the reconstruction chain introduces rather complicated dependencies between different components of the framework, what is highly undesirable from the point of view of software design. In order to solve both problems, we have designed a set of classes that manage data manipulation, i.e. storage, retrieval and exchange within the framework.

It was decided to use the “white board” concept, which is a single exchange object where all data are stored and made publicly accessible. For that purpose, we have employed TFolder facility of ROOT. This solution solves the problem of inter-module dependencies.

There are two frequently occurring use-cases regarding data-flow within the framework:

1. data production: produce - write - unload (clean)
2. data processing: load (retrieve) - process - unload

Loaders are utility classes that encapsulate and automate those tasks. They reduce the user's interaction with the I/O routines to the necessary minimum, providing a friendly and manageable interface, which for the above use-cases, consists of only 3 methods:

- Load – retrieves the requested data to the appropriate place in the white board (folder)

- Unload – cleans the data
- Write – writes the data

Such an insulation layer has number of advantages:

- facilitate data access,
- avoid the code duplication in the framework,
- minimize the risk of bugs in I/O management. The ROOT object oriented data storage extremely simplifies the user interface, however, there are a few pitfalls that are not well known to inexperienced users.

To begin with, we need to introduce briefly basic concepts and the way AliRoot operates. The basic entity is the event, i.e. all data recorded by the detector in a certain time interval plus all the information reconstructed from these data. Ideally, the data are produced by a single collision selected by a trigger for recording. However, it may happen that the data from the previous or proceeding events are present, because the bunch-crossing rate is higher than the maximum detector frequency (pile-up), or simply more than one collision occurred within one bunch crossing.

Information describing the event and the detector state is also stored, like bunch crossing number, magnetic field, configuration, alignment, etc. In the case of Monte-Carlo simulated data, information concerning the generator simulation parameters is also kept. Altogether, this data is called the “header”.

In case of collisions that produce only a few tracks (best example are the pp collisions), it may happen that the total overhead (the size of the header and the ROOT structures supporting object oriented data storage) is not negligible compared to the data itself. To avoid such situations, the possibility of storing an arbitrary number of events together within a **run** is required. Hence, the common data can be written only once per run and several events can be written to a single file.

It was decided that data related to different detectors and different processing phases should be stored in different files. In such a case, only the required data need to be downloaded for an analysis. It also allows for practical altering of the files if required, for example, when a new version of reconstruction or simulation has to be run for a given detector. Hence, only new files are updated and all the rest remains untouched. It is especially important, because it is difficult to erase files in mass storage systems. This also provides for easy comparison with data produced by competing algorithms.

Header data, configuration and management objects are stored in a separate file, which is usually named galice.root (for simplicity we will further refer to it as galice).

3.8.1 The “White Board”

The folder structure is shown in Figure 10. It is divided into two parts:

- **event data** that have the scope of single event
- **static data** that do not change from event to event, i.e. geometry and alignment, calibration, etc.

During start-up of AliRoot the skeleton structure of the ALICE white board is created. The **AliConfig** class (singleton) provides all the functionality that is needed to construct the folder structures.

Event data are stored under a single subfolder (the event folder), named as specified by the user when opening a session (run). Many sessions can be opened at the same time, provided that each of them has an unique event folder name, so they can be distinguished. This functionality is crucial for superimposing events on the level of the summable digits, i.e. analogue detector response without the noise contribution (event merging). It is also useful when two events, or the same event either simulated or reconstructed using different algorithms, need to be compared.

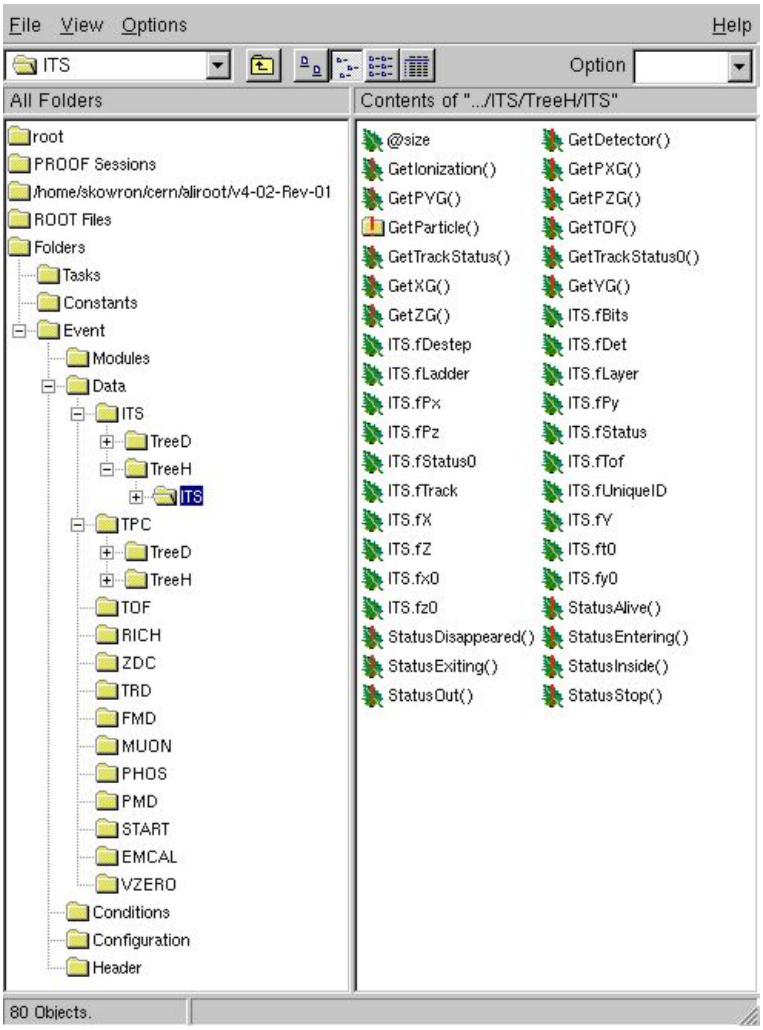


Figure 10: The folders structure. An example event is mounted under the "Event" folder.

3.8.2 Loaders

Loaders can be represented as a four layer, tree like structure (see Figure 11). It represents the logical structure of the detector and the data association.

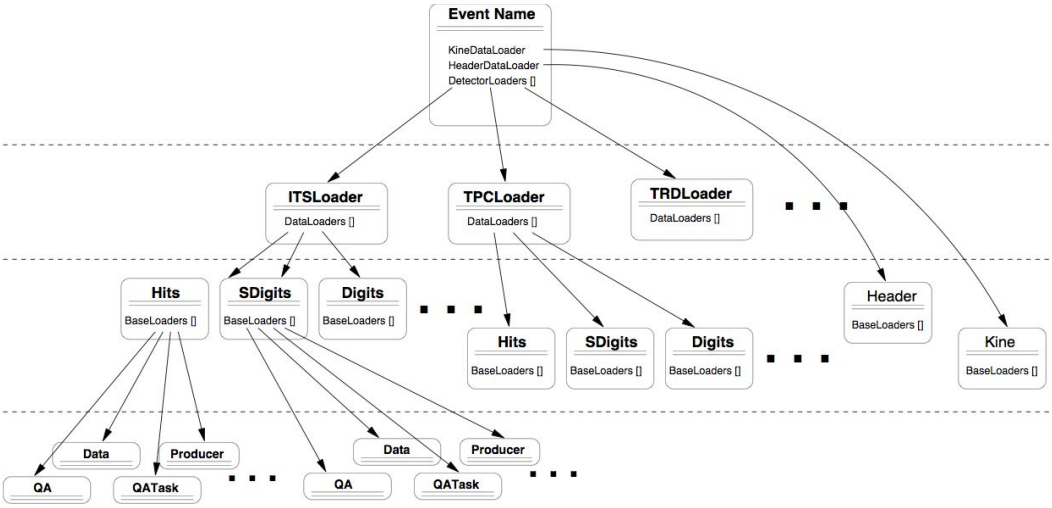


*Figure 11: Loaders diagram. Dashed lines separate layers serviced by the different types of the loaders (from top): **AliRunLoader**, **AliLoader**, **AliDataLoader**, **AliBaseLoader**.*

1. **AliBaseLoader** – One base loader is responsible for posting (finding in a file and publishing in a folder) and writing (finding in a folder and putting in a file) of a single object. **AliBaseLoader** is a pure virtual class because writing and posting depend on the type of an object. the following concrete classes are currently implemented:

   - **AliObjectLoader** – It handles **TObject**, i.e. any object within ROOT and AliRoot since an object must inherit from this class to be posted to the white board (added to **TFolder**).
   - **AliTreeLoader** – It is the base loader for **TTrees**, which requires special handling, because they must be always properly associated with a file.
   - **AliTaskLoader** – It handles **TTask**, which need to be posted to the appropriate parent **TTask** instead of **TFolder**.

   AliBaseLoader stores the name of the object it manages in its base class TNamed to be able to find it in a file or folder. The user normally does not need to use these classes directly and they are rather utility classes employed by **AliDataLoader**.

2. **AliDataLoader** manages a single data type, for example digits for a detector or kinematics tree. Since a few objects are normally associated with a given data type (data itself, quality assurance data (QA), a task that produces the data, QA task, etc.). **AliDataLoader** has an array of **AliBaseLoader**, so each of them is responsible for each object. Hence, **AliDataLoader** can be configured to meet the specific requirements of a certain data-type.

   A single file contains data corresponding to a determined processing phase and of one specific detector only. By default, the file is named according to the schema *Detector Name + Data Name + .root* but it can be changed during run-time if needed. Doing so, the data can be stored in or retrieved from an

alternative source. When needed, the user can limit the number of events stored in a single file. If the maximum number is exceeded, the current file will be closed and a new one will be created with the consecutive number added to the name of the first one (before the *.root* suffix). Of course, during the reading process, files are also automatically interchanged behind the scenes, invisible to the user.

The **AliDataLoader** class performs all the tasks related to file management e.g. opening, closing, ROOT directories management, etc. Hence, for each data type the average file size can be tuned. This is important, because it is on the one hand undesirable to store small files on the mass storage systems, on the other hand, all file systems have a maximum allowed file size.

3. **AliLoader** manages all data associated with a single detector (hits, digits, summable digits, reconstructed points, etc.). It contains an array of **AliDataLoader**, and each of them manages a single data-type.

   The **AliLoader** object is created by a class representing a detector (inheriting from **AliDetector**). Its functionality can be extended and customized to the needs of a particular detector by creating a specialized class that derives from **AliLoader**. The default configuration can be easily modified either in **AliDetector**::MakeLoader or by overriding the method **AliLoader**::InitDefaults.

4. **AliRunLoader** is the main handler for data access and manipulation in AliRoot. There is only one such an object for each run. It is always named **RunLoader** and stored on the top (ROOT) directory of a galice file.

   It keeps an array of **AliLoader**'s, one for each detector, manages the event data that are not associated with any detector – i.e. Kinematics and Header – and utilizes instances of **AliDataLoader** for this purpose.

   The user opens a session using the static method **AliRunLoader**::Open, which takes three parameters: file name, event folder name and mode. If mode is “new”, a file and a run loader are created from scratch. Otherwise, a file is opened and a run loader gets looked-up inside the file. If successful, the event folder is created under the name provided (if it does not exist yet), and the structure presented in Figure 11 is created within the folder. The run loader is put into the event folder, so the user can always find it there and use it for further data management.

   AliRunLoader provides the simple method GetEvent(n) to loop over events within a run. A call clears all currently loaded data and automatically posts the data for the newly requested event.

   In order to facilitate the way the user interacts with the loaders, **AliRunLoader** provides a set of shortcut methods. For example, if digits are required to be loaded, the user can call **AliRunLoader**::LoadDigits("ITS TPC") instead of finding the appropriate **AliDataLoader**'s responsible for digits for ITS and TPC, and then request to load the data for each of them.

3.9 Calibration and alignment

3.9.1 Calibration framework

The calibration framework is based on the following principles:

- The calibration and alignment database contains instances of ROOT **TObject** stored in ROOT files.
- Calibration and alignment objects are run-dependent objects.
- The database is read-only and provides for automatic versioning of the stored objects.
- Three different data storage structures are available:
  - a GRID folder containing ROOT files, each one containing one single ROOT object. The ROOT files are created inside a directory tree, defined by the object's name and run validity range;
  - a LOCAL folder containing ROOT files, each one containing one single ROOT object, with a structure similar to the Grid one;
  - a LOCAL ROOT file containing one or more objects (so-called “dump”). The objects are stored into ROOT TDirectories defined by the object's name and run range.
- Object storing and retrieval techniques are transparent to the user: he/she should only specify the kind of storage he wants to use (“Grid”, “local”, “dump”). Objects are stored and retrieved using **AliCDBStorage**::Put and **AliCDBStorage**::Get. Multiple objects can be retrieved using **AliCDBStorage**::GetAll.
- During object retrieval, it is possible to specify a particular version by means of one or more selection criteria.

The main features of the CDB storage classes are the following [48]:

- **AliCDBManager** is a singleton that handles the instantiation, usage and destruction of all the storage classes. It allows the instantiation of more than one storage type at a time, keeping track of the list of active storages. The instantiation of a storage element is done by means of **AliCDBManager**::GetStorage. A storage element is identified by its “URI” (a string) or by its “parameters”. The set of parameters defining each storage is contained in its specific **AliCDBParam** class (**AliCDBGridParam**, **AliCDBLocalParam**, **AliCDBDumpParam**).

- In order to avoid version clashes when objects are transferred from Grid to local and vice versa, we have introduced a new versioning schema. Two version numbers define the object: a “Grid” version and a “Local” version (sub-version). In local storage, only the local version is increased, while in Grid storage, only the Grid version is increased. When the object is transferred from local to Grid the Grid version is increased by one; when the object is transferred from Grid to Local the Grid version is kept and the sub-version is reset to zero.
- **AliCDBEntry** is the container-class of the object and its metadata, whereas the metadata of the object has been divided into two classes: **AliCDBId** contains data used to identify the object during storage or retrieval, and **AliCDBMetaData** holds other metadata which is not used during storage and retrieval.

  The **AliCDBId** object in turn contains:

  - An object describing the name (path) of the object (**AliCDBPath**). The path name must have a fixed, three-level directory structure: “level1/level2/level3”
  - An object describing the run validity range of the object (**AliCDBRunRange**)
  - The version and subversion numbers (automatically set during storage)
  - A string (fLastStorage) specifying from which storage the object was retrieved (“new”, “Grid”, “local”, “dump”)

  The **AliCDBId** object has two functions:

  - During storage it is used to specify the path and run range of the object;
  - During retrieval it is used as a “query”: it contains the path of the object, the required run and if needed the version and subversion to be retrieved (if version and/or subversion are not specified, the highest ones are looked up).

Here, we give some usage examples:

- A pointer to the single instance of the **AliCDBManager** class is obtained by invoking **AliCDBManager**::Instance().
- A storage is activated and a pointer to it is returned using the **AliCDBManager**::GetStorage(const char* URI) method. Here are some examples of how to activate a storage via an URI string. The URI's must have a well defined syntax, for example (local cases):
  - “local://DBFolder” to local storage with base directory “DBFolder” created (if not existing from the working directory)
  - “local://$ALICE_ROOT/DBFolder” to local storage with base directory “$ALICE_ROOT/DBFolder” (full path name)
  - “dump://DBFile.root” to Dump storage. The file DBFile.root is looked for

or created in the working directory if the full path is not specified

- "dump://DBFile.root;ReadOnly" to Dump storage. DBFile.root is opened in read only mode.

Concrete examples (local case):

```
AliCDBStorage *sto =
 AliCDBManager::Instance()->GetStorage("local://DBFolder"):

 AliCDBStorage *dump =
 AliCDBManager::Instance()->GetStorage
("dump:///data/DBFile.root;ReadOnly"):
```

Creation and storage of an object, how an object can be created and stored in a local database:

Let's suppose our object is an **AliZDCCalibData** object (container of arrays of pedestals constants), whose name is "ZDC/Calib/Pedestals" and is valid for run 1 to 10.

```
AliZDCCalibData *calibda = new AliZDCCalibData();
// ... filling calib data...

  // creation of the AliCDBId object (identifier of the object)
  AliCDBId id("ZDC/Calib/Pedestals",1,10);

  // creation and filling of the AliCDBMetaData
  AliCDBMetaData *md = new AliCDBMetaData();
  md->Set... // fill metadata object, see list of setters...

  // Activation of local storage
  AliCDBStorage *sto =
  AliCDBManager::Instance()->GetStorage("local://$HOME/DBFolder");

  // put object into database
  sto->Put(calibda, id, md);
```

The object is stored into local file:
$HOME/DBFolder/ZDC/Calib/Pedestals/Run1_10_v0_s0.root

Retrieval of an object:

```
// Activation of local storage
  AliCDBStorage *sto =
  AliCDBManager::Instance()->GetStorage("local://$HOME/DBFolder");

  // Get the AliCDBEntry which contains the object
"ZDC/Calib/Pedestals",
  valid for run 5, highest version
  AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5)
  // alternatively, create an AliCDBId query and use sto->Get
(query) ...

  // specifying the version: I want version 2
  AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5,2)

  // specifying version and subversion: I want version 2 and
```

```
subVersion 1
  AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5,2,1)
```

Selection criteria can be also specified using **AliCDBStorage**::AddSelection (see also the methods RemoveSelection, RemoveAllSelections and PrintSelectionList):

```
// I want version 2_1 for all "ZDC/Calib/*" objects for runs 1-100
  sto->AddSelection("ZDC/Calib/*",1,100,2,1);
  // and I want version 1_0 for "ZDC/Calib/Pedestals" objects for runs
5-10
  sto->AddSelection("ZDC/Calib/Pedestals",5,10,1,0)

  AliCDBEntry* entry = sto->Get("ZDC/Calib/Pedestals",5)
```

Retrieval of multiple objects with **AliCDBStorage**::GetAll

```
TList *list = sto->GetAll("ZDC/*",5)
```

Use of default storage and drain storages:

AliCDBManager allows to set pointers to a “default storage” and to a “drain storage”. In particular, if the drain storage is set, all the retrieved objects are automatically stored into it.

The default storage is automatically set as the first active storage.

Examples of how to use default and drain storage:

```
AliCDBManager::Instance()->SetDefaultStorage
("local://$HOME/DBFolder");
AliCDBManager::Instance()->SetDrain("dump://$HOME/DBDrain.root");
AliCDBEntry *entry =
AliCDBManager::Instance()->GetDefaultStorage()->Get
("ZDC/Calib/Pedestals",5)
// Retrieved entry is automatically stored into DBDrain.root !
```

To destroy the **AliCDBManager** instance and all the active storages:

```
AliCDBManager::Instance()->Destroy()
```

Create a local copy of all the alignment objects

```
AliCDBManager* man = AliCDBManager::Instance();
 man->SetDefaultStorage(
  "alien://folder=/alice/simulation/2006/PDC06/Residual/CDB/");

 man->SetDrain("local://$ALICE_ROOT/CDB");
 AliCDBStorage* sto = man->GetDefaultStorage();
 sto->GetAll("*",0);
```

```
// All the objects are stored in $ALICE_ROOT/CDB !
```

3.10 The Event Tag System

The event tag system [49] is designed to provide fast pre-selection of events with the characteristics desired. This task will be performed, first of all, by imposing event selection criteria within the analysis code and then by interacting with software that is designed to provide a file-transparent event access for analysis. The latter is an evolution of the procedure that has already been implemented by the STAR [50] collaboration.

In the next sections, we will first describe the analysis scheme using the event tag system. Then, we will continue by presenting in detail the existing event tag prototype. Furthermore, a separate section is dedicated to the description of the two ways to create the tag files and their integration in the whole framework [197].

3.10.1 The Analysis Scheme

ALICE collaboration intends to use a system that will reduce time and computing resources needed to perform an analysis by providing to the analysis code just the events of interest as they are defined by the users' selection criteria. Figure 12 gives a schematic view of the whole analysis architecture.

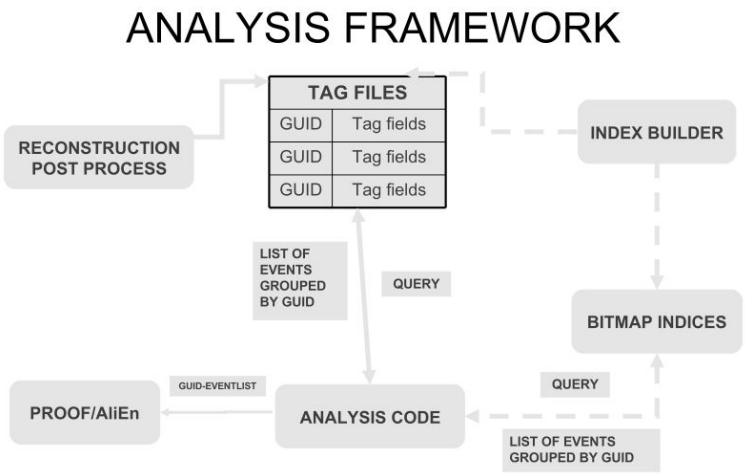


Figure 12: The selected analysis scheme using the event tag system

Before describing the architecture, let us first define a few terms that are listed in this figure:

- User/Administrator: A typical ALICE user, or even the administrator of the system, who wants to create tag files for all or a few ESDs [197] of a run.
- Index Builder: A code with Grid Collector [51, 52] functionality that allows the creation of compressed bitmap indices from the attributes listed in the tag files.

This functionality will provide an even faster pre-selection.

- Selector: The user's analysis code that derives from the TSelector class of ROOT [53].

The whole procedure can be summarized as follows: The offline framework will create the tag files, which will hold information about each ESD file (top left box of Figure 12) as a final step of the whole reconstruction chain. The creation of the tag files is also foreseen to be performed by each user in a post-process that will be described in the following sections. These tag files are ROOT files containing trees of tag objects. Then, following the procedure flow as shown in Figure 12, the indexing algorithm of the Grid Collector, the so-called Index Builder, will take the produced tag files and create the compressed bitmap indices. In parallel, the user will submit a job with some selection criteria relevant to the corresponding analysis he/she is performing. These selection criteria will be used in order to query the produced compressed indices (or as it is done at the moment, the query will be on the tags themselves) and the output of the whole procedure will be a list of **TEventList** objects grouped by GUID, which is the file's unique identifier in the file catalogue, as it is shown in the middle box of Figure 12. This output will be forwarded to the servers that will interact with the file catalogue in order to retrieve the physical file for each GUID (left part of Figure 12). The final result will be passed to a selector [198] that will process the list of the events that fulfil the imposed selection criteria and merge the output into a single object, whether this is a histogram, a tree or any ROOT object.

The whole implementation is based on the existence of an event tag system that will allow the user to create the tags for each file. This event tag system has been used inside the AliRoot framework [54] since June 2005. In the next section we will describe this system in detail.

3.10.2The Event Tag System

The event tag system has been built with the motivation to provide a summary of the most useful physics information that describe each ESD to the user. It consists of four levels of information [55] as explained in Figure 13:

- Run Level: Fields that describe the run conditions and configurations which are retrieved from the Detector Control System (DCS), the Data Acquisition system (DAQ) and the offline framework.
- LHC Level: Fields that describe the LHC condition per ALICE run which are retrieved from the DCS.
- Detector Level: Fields that describe the detector configuration per ALICE run which are retrieved from the Experiment Control system (ECS).
- Event Level: Fields that describe each event - mainly physics related information, retrieved from offline and the Grid file catalogue.

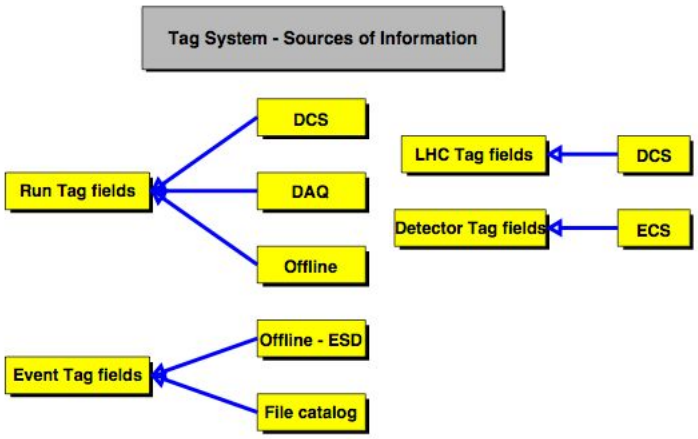


Figure 13: Sources of information for different levels of the event tag system

The corresponding classes that form this system have already been included in AliRoot's framework under the **STEER** module. The output tag files will be ROOT files having a tree structure [199].

Run tags: The class that deals with the run tag fields is called **AliRunTag**. One **AliRunTag** object is associated with each file.

LHC tags: The class that deals with the LHC tag fields is called **AliLHCTag**. One **AliLHCTag** object is associated with each file.

Detector tags: The class that deals with the detector tag fields is called **AliDetectorTag**. Information concerning the detector configuration per ALICE run will be described in the ECS database. One **AliDetectorTag** object is associated with each file.

Event tags: The class that handles the event tag fields is called **AliEventTag**. The values of these fields, as mentioned before, will be mainly retrieved from the ESDs, although there are some fields that will come from the Grid file catalogue. The number of **AliEventTag** objects that are associated to each file is equal to the number of events that are stored inside the initial ESD file.

3.10.3The Creation of the Tag-Files

The creation of the tag-files will be the first step of the whole procedure. Two different scenarios were considered:

- **On-the-fly-creation:** The creation of the tag file comes as a last step of the reconstruction procedure.
- Post-creation**:** After the ESDs have been transferred to the ALICE file catalogue [197], every user has the possibility to run the creation of tag-files as post-process and create his/her own tag-files.

3.10.3.1The on the fly creation scenario

As mentioned before, the on-the-fly creation of tag-files is implemented in such a way that the tags are filled as last step of the reconstruction chain. This process is managed inside the **AliReconstruction** class. Thus, exactly after the creation of the ESD, the file is passed as an argument to **AliReconstruction**::CreateTags. Inside this method, empty **AliRunTag** and **AliEventTag** objects are created. The next step is to loop over the events listed in the ESD file, and finally fill the run and event level information. The naming convention for the output tag file is: **Run*RunId.*Event*FirstEventId_LastEventId.*ESD.tag.root** [199].

3.10.3.2The post-creation scenario

The post-creation procedure provides the possibility to create and store the tag-files at any time [199]. The post-creation of the tag-files implies the following steps:

- The reconstruction code finishes and several ESD files are created.
- These files are then stored in the ALICE file catalogue [].
- Then, the administrator, or any user in the course of her/his private analysis, can loop over the produced ESDs and create the corresponding tag-files.
- These files can either be stored locally or in the file catalogue [].
- As a final step, a user can choose to create a single merged tag file from all the previous ones.

What a user has to do in order to create the tag-files using this procedure, depends on the location of the input AliESDs.root files. Detailed instructions on how to create tag-files for each separate case will be given in the following sections. In general, a user has to perform the following steps:

- Provide information about the location of the AliESDs.root files:
  - Result of a query to the file catalogue (TGridResult [56])
  - Grid stored ESDs
  - An upper level local directory
  - Locally stored ESDs or even a text file
  - CERN Analysis Facility (CAF) stored ESDs [57].
- Loop over the entries of the given input (**TGridResult**, local path, text file) and create the tag file for each entry.
- Either store the files locally or in the Grid's file catalogue
- Merge the tag-files into one file and store it accordingly (locally or in the file catalogue) [58].

Figure 14 provides a schematic view of these functionalities.

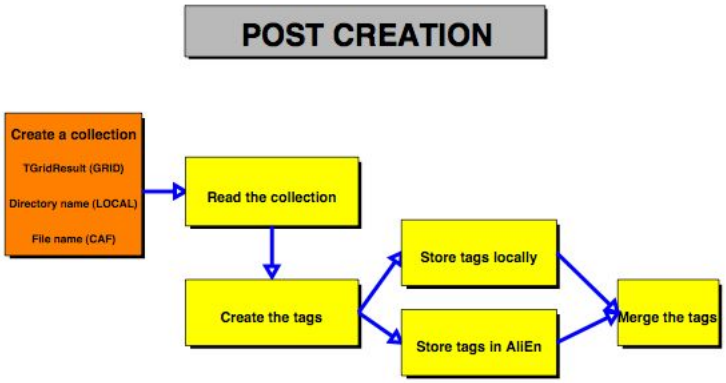


Figure 14: A schematic view of the architecture of the post-creation of tag-files

The class that addresses this procedure is **AliTagCreator**. The main methods of the class and their corresponding functionalities are described in the following lines:

- SetStorage allows the user to define the place where the tag-files will be stored.
- SetSE allowsto define the desired storage element on the Grid.
- SetGridPath allows the user to define the Grid path under which the files will be stored. Per default, the tag-files will be stored in the home directory of the user in the file catalogue.
- ReadGridCollection is used when creating tag-files from ESDs that are stored in the file catalogue. It loops over the corresponding entries and calls **AliTagCreator**::CreateTags in order to create the tag-files that will be stored accordingly.
- ReadCAFCollection is used when creating tag-files from ESDs that are stored in the CERN Analysis Facility (CAF) [199].
- ReadLocalCollection is used when creating tag-files from ESDs that are stored locally.
-
- MergeTags chains all the tags, locally or stored or in the Grid, and merges them by creating a single tag file named **Run*RunId*.Merge.ESD.tag.root**. This file is then stored either locally or in the Grid according to the value set in the SetStorage method.

3.10.3.3Usage of AliRoot classes

The following lines intend to give an example on how to use the AliTagCreator class in order to create tags. Additional information can be found in [199]. There are three different cases depending on the location where the AliESDs.root files are stored:

- Locally stored AliESDs.root files;
- CAF stored AliESDs.root files;
- Grid stored AliESDs.root files.

We will address the three different cases separately.

Locally stored AliESDs.root

We assume that for debugging or source code validation reasons, a user stored a few AliESDs.root files locally under $HOME/PDC06/pp. One level down, the directory structure can be of the form:

- xxx/AliESDs.root
- yyy/AliESDs.root
- zzz/AliESDs.root

where xxx is the directory which can can sensibly be named *Run1, Run2* etc. or even simply consists of the run number. In order to create the tag-files, we need to create an empty **AliTagCreator** object. The next step is to define whether the produced tags will be stored locally or on the Grid. If the second option is chosen, the user must define the SE and the corresponding Grid path where the tag-files will be stored. If the first option is chosen, the files will be stored locally in the working directory. Finally, the call of **AliTagCreator**::ReadLocalCollection allows the user to query the local file system and create the tag-files.

```
//create an AliTagCreator object
 AliTagCreator *t = new AliTagCreator();
 //Store the tag-files locally
 t->SetStorage(0);
 //Query the file system, create the tags and store them
 t->ReadLocalCollection("/home/<username>/PDC06/pp");
 //Merge the tags and store the merged file
 t->MergeTags();
```

AliESDs.root on the CAF

When ESD files are to be stored on the CAF, we have to provide a text file that contains information about the location of the files in the storage element of the system [199, 199]. We now assume that this input file is called ESD.txt and is located in the working directory, indicate the steps that one has to follow:

```
//create an AliTagCreator object
 AliTagCreator *t = new AliTagCreator();
 //Store the tag-files in AliEn's file catalog
 t->SetStorage(0);
 //Read the entries of the file, create the tags and store them
 t->ReadCAFCollection("ESD.txt");
 //Merge the tags and store the merged file
 t->MergeTags();
```

AliESDs.root on the GRID

When ESD files are stored in the file catalogue, the first thing a user needs to have is a ROOT version compiled with AliEn support. Detailed information on how to do this can be found in [199]. Then, we need to invoke the AliEn API services [199] and passing a query to the file catalogue (**TGridResult**). The following lines give an example of the whole procedure:

```
//connect to AliEn's API services
 TGrid::Connect("alien://pcapiserv01.cern.ch:10000","<username>");
 //create an AliTagCreator object
 AliTagCreator *t = new AliTagCreator();
 //Query the file catalogue and get a TGridResult
 TGridResult* result =
 gGrid->Query("/alice/cern.ch/user/p/pchrista/PDC06/pp/*",
 "AliESDs.root","","");
 //Store the tag-files in AliEn's file catalog
 t->SetStorage(1);
 //Define the SE where the tag-files will be stored
 t->SetSE("ALICE::CERN::se01");
 //Define the Grid's path where the tag-files will be stored
 t->SetGridPath("PDC06/Tags");
 //Read the TGridResult, create the tags and store them
 t->ReadGridCollection(result);
 //Merge the tags and store the merged file
 t->MergeTags();
```

3.11 Meta-data in ALICE

2.

•

–

–

2.

2.

2.

•

2.

4 Run and File Metadata for the ALICE File Catalogue

4.1 Introduction

In order to characterize physics data it is useful to assign metadata to different levels of data abstraction. For data produced and used in the ALICE experiment a three layered structure will be implemented:

- Run-level metadata,
- File-level metadata, and
- Event-level metadata.

This approach minimizes information duplication and takes into account the actual structure of the data in the ALICE File Catalogue.

Since the event-level metadata is fully covered by the so called 'event tags' (stored in the **`ESDtags.root`** files), it will not be discussed in this document. There is a mechanism in place to efficiently select physics data based on run- and file-level conditions and then make a sub-selection on the event level, utilizing the event level metadata. This pre-selection on run- and file-level information is not necessary, but can speed up the analysis process.

This document is organized as follows: First we will discuss the path and file name specifications. The list of files/file types to be incorporated in the file catalogue will be discussed. Finally, we will list the meta tags to be filled for a given run (or file).

4.2 Path name specification

The run and file metadata will be stored in the ALICE File Catalogue. A directory structure within this database will ensure minimization of metadata duplication. For example, all files written during one specific physics-run do not need to be tagged one by one with a 'run' condition, since all files belonging to this physics run will be tagged with the same information at the run directory level. It is advantageous to organize all these files in a directory tree, which avoids additional tagging of all files. Since this tree structure will be different to that of CASTOR, a mechanism is provided to make sure that the files once created by DAQ are later ‘tagged’ with the proper information encoded in the directory tree itself.

The CASTOR tree structure will look like

/castor/cern.ch**/**.../‹Year›/‹Month›/‹Day›/‹Hour›/.

The CASTOR file names will be of fixed width, containing information about the year (two digits: YY), the run-number (9 digits, zero padded: RRRRRRRRR), the host-

identifier of the GDC (three digits, zero padded: NNN), and a sequential file-count (S), ending up in names of the following structure:

YYRRRRRRRRRNNN.S.raw or YYRRRRRRRRRNNN.S.root.

This is different to what we discussed before where the CASTOR system and the file catalogue had the same directory structure, and getting the 'hidden' information from the CASTOR path name was easy.

The path name(s) where all ALICE files will be registered in the ALICE File Catalogue will have the following structure:

/data/<Year>/<AcceleratorPeriod>/<RunNumber>/ for real data,

/sim/<Year>/<ProductionType>/<RunNumber>/ for simulated data,

where **<Year>**, **<AcceleratorPeriod>**, and **<RunNumber>** contain values like **2007**, **LHC7a**, and **123456789** (nine digits, zero padded). **<ProductionType>** gives information about the specific simulation which was run, which includes the 'level of conditions' applied: **Ideal**, **Full**, **Residual**. Therefore one possibility example for a name for**<ProductionType>** would be **PDC06_Residual**. The subdirectory structure provides the place for different files from the specific runs and will be called

raw/ for raw data,

reco/<PassX>/cond/ for links to calibration and condition files,

reco/<PassX>/ESD/ for ESD and corresponding ESD tag-files,

reco/<PassX>/AOD/ for AOD files.

The list of these subdirectories might be extended on a later stage if necessary.

<PassX> will specify a certain reconstruction pass (encoded in a production tag) of the raw data in the same parent directory. For each raw data file the output files of several production passes might be present in the **reco/** directory, to be distinguished from each other by a different production tag **<PassX>**.

The **cond/** directory will contain a data set (in the form of an xml-file) which links back to the actual condition data base (CDB) files. The CDB files themselves will be stored in a three layered directory tree:

<Detector>/<Type>/<Subtype>,

where **<Detector>** will be something like **TPC**, **TRD**, **PHOS**, …, and **<Type>** will specify the type of condition data, like **Calib** or **Align**. **<Subtype>** can assume generic strings like **Data**, **Pedestals**, or **Map**. These conditions may be stable on a longer time scale than a single run. In order to avoid duplication or replication of these files (which can be quite large in case of certain mapping files) for each run, they will be put a few levels higher, namely in the

/data/<Year>/<AcceleratorPeriod>/CDB/

directory. The name of the actual calibration files (stored in the subdirectories **<Detector>/<Type>/<Subtype>** mentioned above) is chosen in a way to make a successful mapping between the correct calibration files for each run:

Run<XXX>_<YYY>_v<ZZ>.root,

where **<XXX>** and **<YYY>** is the first and last run number for which this file applies, and **<ZZ>** is the calibration version number.

4.3 File name specification

The file names of the stored ALICE data will be kept simple, and are considered unique for the current subdirectory, for example

<NNN>.<S>.AliESDs.root for ESD files,

where **<NNN>.<S>** is the identifier of the corresponding raw data file (**<NNN>.<S>.raw/<NNN>.<S>.root**, also called raw data-chunk). Therefore different subdirectories (for example the **reco/<PassX>/ESD/** directories for different **<RunNumber>** parent directories) may contain files with the same name but with different content. Nevertheless, the GUID scheme (and the directory structure) makes sure that these files can be distinguished.

To make local copies of files without creating the full directory structure locally (e.g. for local calibration runs over a certain set of files), a macro was developed to stage the requested data files either locally or to another ROOT supported storage area. Using **alienstagelocal.C** will copy the files and rename them to more meaningful names, which allows to distinguish them by their new filenames.

Each PbPb event was expected to have a raw data size of 12.5 Mb on average. The file size limit of 2 Gb restricts the number of events to 160 per raw data file. Since neither the ALTRO compression nor the Hough encoding will be used for now, the event size will actually be a factor of 4 (2×2) larger, which means 40 events per raw data file (or 50 Mb/event). A central event currently takes about 50 min to be processed, while a typical minimum bias event takes 10 min on average. This means that it will take 6:40 h to reconstruct all the events of one raw data file. It is therefore not necessary to group several raw data files together to produce 'larger' ESDs. As result, each **AliESDs.root** file will correspond to one exactly raw file.

For pp events the expected size of 1 Mb/event is currently exceeded by a factor of about 16 (16 Mb/event). One raw data file with a size limit of 2 Gb will therefore contain 125 events. Since it takes about 1 h to reconstruct 100 pp events, one raw data file will be processed in about 1:15 h.

A typical data-taking run with a length of 4 h and a data rate of 100 events/s will generate 360k PbPb (pp) events in 9000 (720 for pp) raw data files. The corresponding **ESD/** directory will therefore contain 9000(720)×n files. It is important to keep the number n of output files small, by storing all necessary information in as few files as possible.

4.4 Files linked to from the ALICE File Catalogue?

'All' available ALICE files will be handled (meaning registered and linked to) by the file catalogue. In particular, this includes the following files/file types:

- raw data files,
- AliESDs.root files,
- AliESDfriends.root files,
- ESDtags.root files, and
- AliAOD.root files.

For different files/file types a different set of metadata tags will be added which contains useful information of/for that specific file/file type

4.5Metadata

In the following, we will list metadata information that will be stored in the database. Note that an actual query of the file catalogue (using the AliEn command **`find <tagname>:<cond>`**) might contain more specific tags/tag names than elaborated here. For example, some of the available information will be directly stored in the directory/path structure (and only there; like the **`<RunNumber>`** and **`<Year>`**), whereas other information can be 'calculated' from the metadata provided (e.g. the **`month`** when the data was taken, by extracting this information from the **`run start/stop time`** metadata).

4.5.1 Run metadata

tag name	data format/possible values	data source
run comment	Text	log book
run type	**physics**, **laser**, **pulser**, **pedestal**, **simulation**	log book
run start time	yyyymmddhhmmss	log book
run stop time	yyyymmddhhmmss	log book
run stop reason	**normal**, **beam loss**, **detector failure**, …	log book
magnetic field setting	**FullField**, **HalfField**, **ZeroField**, **ReversedHalfField**, **ReversedFullField**, …	DCS
collision system	**PbPb**, **pp**, **pPb**, …	DCS
collision energy	text, e.g **5.5TeV**	DCS
trigger class		log book
detectors present in run	bitmap: **0**=not included, **1**=included	log book
number of events in this run		log book
run sanity	flag bit or bit mask, default **1**=OK	manually

for reconstructed data:

tag name	Data format/possible values	data source

production tag		reconstruction
production software library version		reconstruction
calibration/alignment settings	**ideal**, **residual**, **full**	reconstruction

for simulated data:

tag name	data format/possible values	data source
generator	**Hijing**, **Pythia**, …	manually
generator version		manually
generator comments	Text	manually
generator parameters		manually
transport	**Geant3**, **Fluka**, …	manually
transport version		manually
transport comments	Text	manually
transport parameters		manually
conditions/alignment settings	**ideal**, **residual**, **full**	manually
detector geometry		Manually
detector configuration		manually
simulation comments	Text	manually

All this information will be accessible through the Config.C file as well. A link to the specific Config.C file used for the production will be provided in each **/sim/<Year>/<ProductionType>/<RunNumber>/** directory.

The implementation of a 'simulation log book' is considered to be useful. It would serve as a summary of the simulation effort and the information needed for the metadata could be read directly from there.

4.5.2 File metadata

tag name	data format/possible values	data source
file sanity	flag bit: e.g. **online**/**offline** or **available**/**not available**; default **1**=**online**/**available**	manual

Additional information about files is available from the file itself, so there is no need to create special metadata.

Additional data is needed in order to store the files into the correct directories (see 'Path name specification' above). DAQ will take care of that; they will store the files in the proper location right away and – if necessary – create these new directories.

4.6 Population of the database

The metadata database has to be filled with the values obtained from different sources,

indicated by the ‘data source’ descriptor in the tables given above. At the end of each run, a process running within the shuttle program will retrieve the necessary information from the DAQ log book, and the DCS to write them into the database. This will be done in the same way the detectors retrieve their calibration data.

4.7 Data safety and backup procedures

Since the filenames are meaningless by themselves and the directory structure is completely virtual, it is very important to have a backup system or persistency scheme in place. Otherwise, a crash, corruption or loss of the database results in a complete loss of the physics data taken. Even though the logical files would be still available, their content would be completely unknown to the user.

The currently implemented backup procedure duplicates the whole ALICE File Catalogue and is considered a sufficient security measure.

5 AliEn reference

5.1 What's this section about?

This section makes you familiar with AliEn user interfaces and enables you to run production and analysis jobs as an ALICE user using the AliEn infrastructure.

The first part describes the installation procedure for the application interface client package **gapi** and the functionality of the command line interface – the AliEn shell **aliensh**.

The second part describes the AliEn GRID interface for the ROOT/AliROOT framework and introduces you to distributed analysis on the basis of examples in batch style using the AliEn infrastructure.

To be able to repeat the steps described in this HowTo, you must have a valid GRID certificate and you must be a registered user within the AliEn ALICE VO. Information about that can be found at

http://alien.cern.ch/twiki/bin/view/Alice/UserRegistration

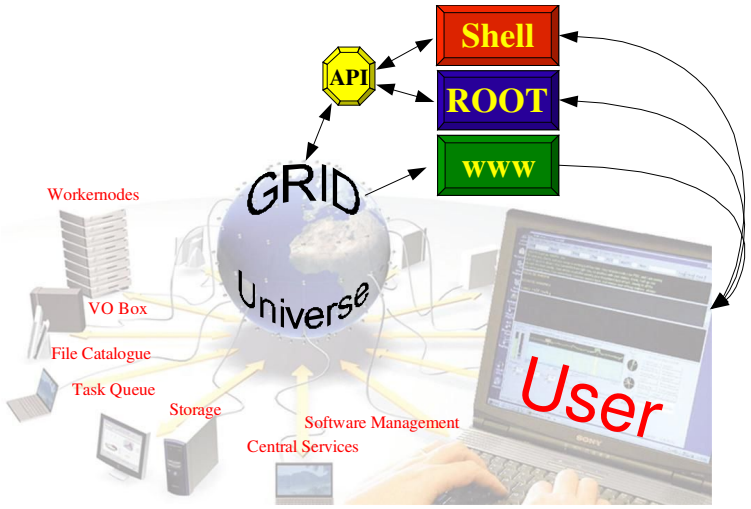


Figure 15: ALICE Grid infrastructure

5.2 The client interface API

All user interactions with AliEn are handled using a client-server architecture as shown in Figure 16. The **API** for all client applications is implemented in the library `libgapiUI`. Every client interface can communicate over a session-based connection with an **API** server. The **API** server exports all functions to the client interface, which are bridged from the AliEn PERL-UI via the AlienAS PERL→C++ interface script.

Figure 16: The AliEn interface structure

To use the shared library in a C++ program, just add to the source the line below and link with `-lgapiUI`.

```
#include <gapiUI.h>
```

5.3 Installation of the client interface library package – gapi

The standard way to install the client interface package is to use the AliEn installer. The source installation is explained for special use cases - you might skip that section.

5.3.1 Installation via the AliEn installer

- Download the AliEn installer from:

`http://alien.cern.ch/alien-installer`

  using a browser or `wget`. Set the permissions for execution of the installer:
  `chmod ugo+rx alien-installer`

- Run the installer:

`./alien-installer`

- Select in the version menu the highest version number with the 'stable' tag and follow the instructions of the installer until you are asked, which packages to install.

  Select 'gshell - Grid Shell UI' to install the client interface and ROOT to get an AliEn-enabled ROOT package and proceed.

Note: The installer asks you, where to install the packages. The default location is /opt/alien. To create this directory you need ROOT permissions on that machine. If you are not the administrator of that machine you can only install into your HOME directory or a directory, where you have write permissions.

5.3.2 Recompilation with your locally installed compiler

AliEn comes with the CERN standard compiler (currently gcc 3.2.3). If you want to compile applications with your locally installed compiler (different from AliEn's one) and link against the API library, you have to recompile the API library with your compiler.

To do so, execute the script

`/opt/alien/api/src/`**`recomplile.api [<alien dir> [modules]]`**

The script recompiles the API library and installs over the binary installation in your AliEn installation directory.

If you execute the script without arguments, your installation directory configured in the installer will be detected automatically. If you want to recompile a different installation to the one installed by the installer script, you can pass the installation directory as first argument. If you add 'modules' as the 2nd argument, also the PERL and JAVA modules

will be rebuilt. To compile the modules you need to install also the “client” package using the alien installer.

After having recompiled successfully, you should set the variable GSHELL_GCC to the path of your gcc executable (e.g. `export GSHELL_GCC=`which gcc``).

5.3.3 Source Installation using AliEnBits

Create a directory to install the build sources e.g.:

```
mkdir $HOME/alienbits/ ; cd $HOME/alienbits/
```

RSync to the development HEAD

```
rsync -acvz--exclude=alien.conf.mk
rsync://alien.cern.ch:8873/releases/HEAD/
```

or an older version e.g. v2-6:

```
rsync -acvz--exclude=alien.conf.mk
rsync://alien.cern.ch:8873/releases/v2-6/ .
```

Login in to the CVS with password 'cvs':

```
cvs -d :pserver:cvs@alisoft.cern.ch:/soft/cvsroot login
```

Update your rsynced directory with the latest CVS:

```
cvs -d :pserver:cvs@alisoft.cern.ch:/soft/cvsroot update -dPA
```

Run the configure script and define the installation location:

`./configure--prefix=/opt/alien` (or `--prefix=$HOME/alien`)

Change to the api-client source directory:

```
cd $HOME/alienbits/apps/alien/api
```

or the api-client-server source directory:

```
cd $HOME/alienbits/apps/alien/apiservice
```

Start the build:

```
make
```

AliEn Bits will download and compile every package, which is needed to build the **API**. This includes the compilation of PERL and GLOBUS and will take about 1 hour. Install the build:

```
make install
```

Now you have compiled everything that is used by the **API** package from sources on your computer.

5.3.4 The directory structure of the client interface

The location of the client interface installation is (if not changed in the Installer or in AliEnBits with the`--prefix` option) under `/opt/alien/api.`

The structure of the default installation is

- `/opt/alien/api/`
  - `bin/`

        - `aliensh`
          executable for the AliEn shell interface
          There are no modification of the PATH or LD_LIBRARY_PATH environment variables needed to run the shell.
        - `gbbox`
          busy box executable to run arbitrary commands (we will give a more detailed explanation later)
        - `alien_<cmd>`
          executables of **alien** commands outside aliensh (e.g. from a tcsh)
    - `etc/`
      shell scripts defining auto-completion functions for the **aliensh**
    - `include/`
        - `gapi_attr.h`
          C++ include file defining gapi file attributes
        - `gapi_dir_operations.h`
          C++ include file defining POSIX directory operations like opendir, closedir etc.
        - `gapi_file_operations.h`
          C++ include file defining POSIX file operations like open, close, read etc.
        - `gapi_job_operations.h`
          C++ include file defining job interface operations like submit, kill, ps etc.
        - `gapi_stat.h`
          C++ include file defining a POSIX stat interface
        - `gapiUI.h`
          C++ include file containing the interface class for command execution and authentication on a remote **apiservice** server. This can be considered as the lowest level interface and it is independent of AliEn. Other include files encapsulate more abstract functionalities, which already 'know' the AliEn functionality.
    - `lib/`
        - `java/`
          java interface to libgapiUI , if installed
        - `libgapiUI.a`
        - `libgapiUI.la`
          libtool library file
        - `libgapiUI.so`
          link to a current version of the gapiUI shared library
        - `libgapiUI.so.2`
          link to a current version of the gapiUI shared library
        - `libgapiUI.so.2.0.2`
          a current version of the gapiUI shared library (the version number might have changed in the meanwhile)

Additional files for the server

- `sbin/`
  - `gapiserver`
    server executable
- `scripts/`
  - `LocalFS.pl`
    test interface script to be used by the **gapiservice** executable
  - `AlienAS .pl`
    PERL script interfacing from the C++ server to the AliEn native PERL UI
  - `gapiserver.pl`
    startup script for the **API** server **gapiserver**

5.4 Using the Client Interface - Configuration

A minimum configuration to use the API client is recommended, although the API client works without PATH or (DY)LD_LIBRARY_PATH modifications. These modifications are for user convenience to avoid typing the full executable paths. If you want to have the executable of **aliensh** and other packaged executables in your PATH use:

```
export PATH=/opt/alien/api/bin:$GLOBUS_LOCATION/bin:$PATH
```

5.5 Using the Client Interface - Authentication

Session tokens are used for performance reasons for authentication and identification to any **API** server. These tokens are similar to GRID proxy certificates (limited lifetime). Additionally they are modified every time they have been used. Every token represents **one** well-defined role, which you specify in the initialization of the token. It is issued by the **API** server and shipped over an SSL connection to the client.

5.5.1 Token Location

The **gapi** library supports two methods for storing an authentication token:

- Store in memory

  The token can only be accessed by one application. This method is used e.g. in ROOT and will be explained later.

- Store in a token file.

  This method is used for the shell implementation **aliensh**.

In the following, we will discuss the handling of file-tokens.

5.5.2 File-Tokens

The file-token mechanism is implemented in three commands:

- `alien-token-init`
- `alien-token-info`

- `alien-token-destroy`

None of the commands touches your current user environment, i.e. they don't modify your `PATH` or `LD_LIBRARY_PATH` environment variables.

5.6 Session Token Creation

Session tokens are stored in the `/tmp` directory following the naming convention:

```
/tmp/gclient_token_${UID}
```

For security reasons, permissions on the token file are set to `"-rw-------"` for the owner of the token.

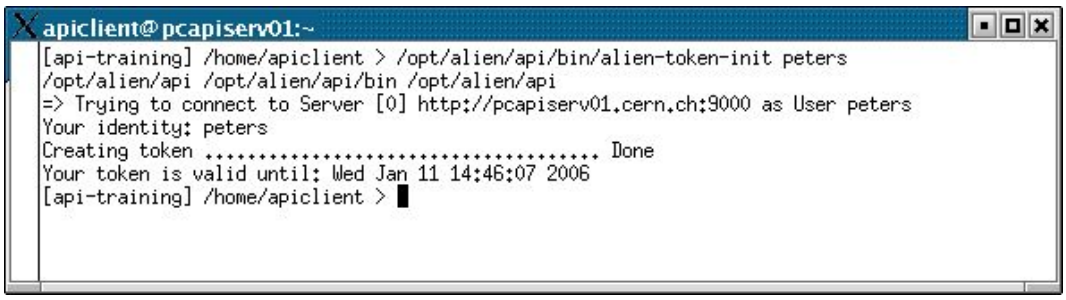
There are three authentication methods for the API service:

- using GRID proxy certificates
- using password authentication (by default disabled on ALICE servers)
- using AliEn job token

Note: It is always recommended to use a GRID proxy certificate for authentication. Password Authentication and Job Token are described for completeness.

5.6.1 Token Creation using a GRID Proxy certificate – `alien-token-init`

To obtain a session token, you need a GRID proxy certificate. For this, you have to execute `alien-token-init [role]` to contact one of the default **API** services and to obtain a session token for this service the `[role]` parameter is optional. If you don't specify a role, your local UNIX account name ($USER) is taken as the role in the session request. You can request the middle ware administrators to allow you to use other roles than your personal identity, e.g. the `aliprod` role for generic productions.



The list of default **API** services is obtained automatically. The client tries to connect in a well defined way to one of the services available, and to establish a session. The list of available services is centrally configured.

If none of the (redundant) services are available, no token will be created.

Instead of using the automatic server endpoint configuration, you can force a specific **API** server endpoint and role via the following environment variables:

- `alien_API_HOST`:

API server host name e.g. `pcapiserv01.cern.ch`

- `alien_API_PORT`:
  API server information port, default `9000`
- `alien_API_USER`:
  Role you want to request with your certificate in AliEn
- `alien_API_VO`:
  Virtual Organization. Presently this parameter is not used.

The client itself does not choose the lifetime of session tokens. It depends only on the configuration of the **API** server you are connecting. `alien-token-init` has an auto-bootstrap feature:

- The first time you use this command, it will run the bootstrap procedure, which will inform you about the creation of certain symbolic links etc.
- Ff you move or copy a client interface installation to another directory, the installation bootstraps itself (the first time, when you use the `alien-token-init` command).
- If you miss some of the required libraries, you will get a hint, how to proceed in such a case.

Return Values ($?):

- 0 Token successfully obtained
- 98 Bootstrap error – no permissions
- 99 No Token created – authentication failed

Note: if you have problems like the ones shown here, verify that you don't have the environment variable X509_CERT_DIR set. In case you have, remove it! (`unset X509_CERT_DIR` in bash)

5.6.2 Token Creation using a password – `alien-token-init`

The procedure to create a token with a password is the same as with GRID proxy certificates and is mentioned only for completeness. In a GRID environment all user authentication is done using proxy certificates.

It is possible to configure an API service to authenticate via the PAM modules using login name and password.

One has to specify the account name as the `[role]` parameter to `alien-token-init`, and you will be prompted to enter a password. The client uses an SSL connection to communicate the password, which is validated on the server side using the PAM library. Afterwards, the password is immediately purged from client memory.

5.6.3 Token Creation via AliEn Job Token – `alien-token-init`

A third authentication method using **job tokens** exists: if jobs are run via the AliEn TaskQueue, you can obtain a token using the AliEn job token known in the environment of your GRID batch job. In this case you don't need to specify any `[role]` parameter. The role is mapped automatically according to the AliEn job token found in your job sandbox and mapped on server-side using the AliEn authentication service.

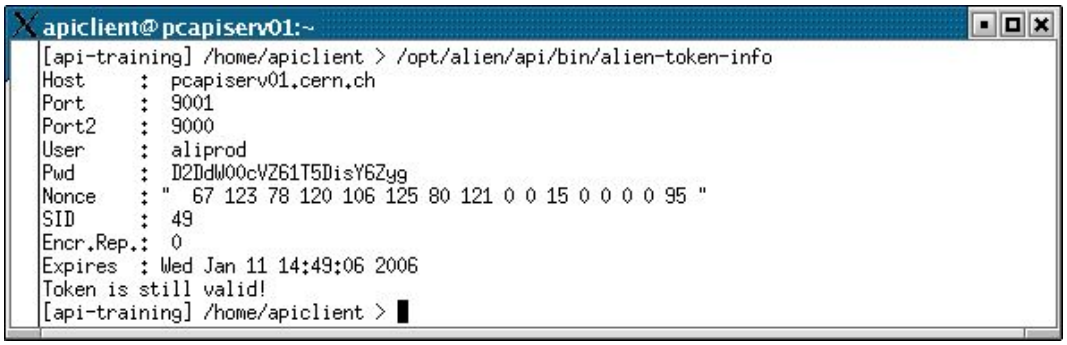
5.6.4 Manual setup of redundant API service endpoints

It is possible to overwrite the central configuration and to specify your own list of **API** service machines via the environment variable `alien_API_SERVER_LIST` e.g.:

```
export alien_API_SERVER_LIST="host1:9000|host2:9000|host3:9000"
```

5.7 Checking an API session token – `alien-token-info`

Similar to `grid-proxy-info` you can execute `alien-token-info` to get information about your session token.



- Host host name of the **API** server, where this token is to be used
- Port communication port for remote command execution
- Port2 information port, where the initial authentication methods and protocol versions are negotiated

- User role associated with this token
- Pwd random session password used in the security scheme
- Nonce- dynamic symmetric CIPHER key
- SID assigned session ID by the **API** server
- Encr. Rep. 0 specifies, that all service replies are not encrypted. 1 specifies, that service replies are fully encrypted.

  Note: the default operation mode is to encrypt all client requests, while server responses *are not encrypted*.
- Expires Local Time, when the token will expire

After the token has expired, the last line of the command will be “Token has expired!”, in case it is valid: “Token is still valid!” If no token can be found, the output will be only: “No token found!”

Return Values ($?):

- 0 Token is valid
- 1 Token is not valid
- 2 No Token found

5.8 Destroying an API session token – `alien-token-destroy`

`alien-token-destroy` is used to destroy an existing session token.

Return Values ($?):

- 0 Token has been removed
- 100 No Token to be removed

5.8.1 Session Environment Files

The `alien-token-init` command produces two other files in the `/tmp` directory.

- /tmp/gclient_env_$UID

  You can source this file, if you want to connect automatically from an application to the same **API** server without specifying host and port name in the code. This will be explained in detail later.
- /tmp/gbbox_$UID_$PID

  This file keeps all the 'client-states' which are not directly connected to authentication or session variables (which are written to the token file): for the moment only the CWD is stored. This file is bound to the PID of your shell to allow different CWDs in different shells.

The “client-state” file is configured automatically in **aliensh** via the environment variable `GBBOX_ENVFILE`.

5.9 The AliEn Shell – aliensh

The **aliensh** provides you with a set of commands to access AliEn GRID computing resources and the AliEn virtual file system. It can be run as an interactive shell as well as for single-command and script-execution.

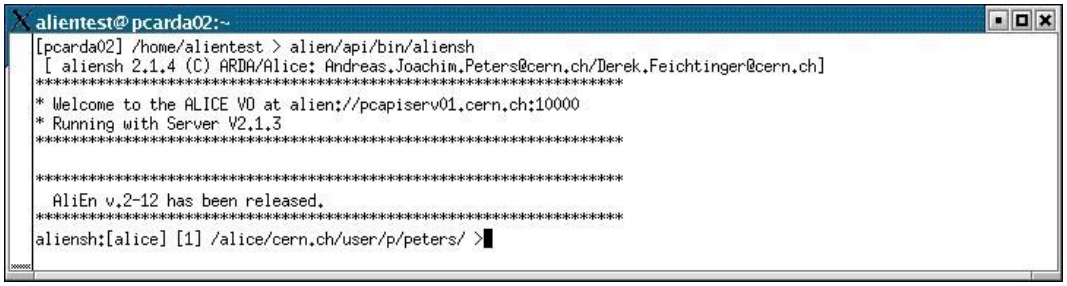
There are three command categories:

- informative + convenience commands
- Virtual File Catalogue + Data Management Commands
- TaskQueue/Job Management Commands

5.9.1 Shell Invocation

5.9.1.1 Interactive Shell Startup/Termination

The shell is started invoking the '**aliensh**' executable as seen in the following example:



The shell advertises the present protocol version used on client-side (aliensh 2.0.0 in this case) and displays the server message of the day (MOTD) after invocation.

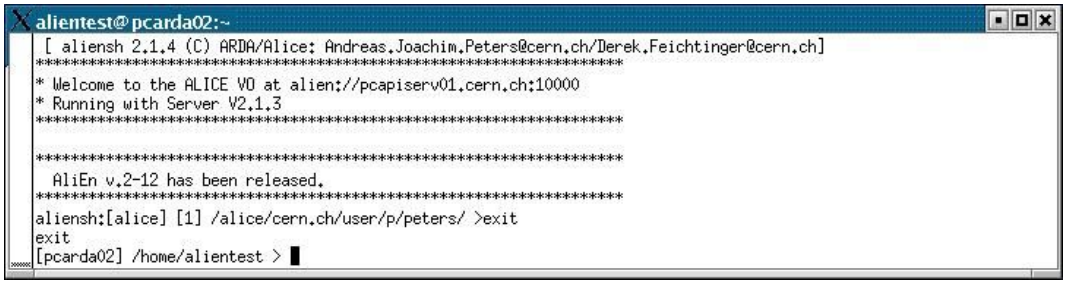
You can execute commands by typing them into the shell and invoking the execution with the ENTER key. In fact, **aliensh** is a special featured bash shell.

Shell return values are all constructed following common practice:

Return Values ($?):

- 0 Command has been executed successfully
- !=0 Command has produced “some” error

Most of the shell commands provide a help text, if executed with a “-h” flag. You can leave the shell with the “exit” command:



5.9.2 Shell Prompt

The shell prompt displays in the 1st bracket the VO name, the 2nd is a command counter. The right hand side of the prompt displays the user's CWD. If the shell is invoked the first time, the CWD is set to the user's home directory. Otherwise the CWD is taken from the previous session.

5.9.3 Shell History

The shell remembers the history of previously executed commands. These commands are stored in the history file `$HOME/.aliensh_history`.

5.9.4 Shell Environment Variables

aliensh defines the following environment variables:

- `alien_API_HOST`
- `alien_API_PORT`
- `alien_API_USER`
- `alien_API_VO`
- `MONALISA_NAME`
  Mona Lisa Site Name e.g. “CERN”
- `MONALISA_HOST`
  Responsible Host running a MonALISA Server
- `APMON_CONFIG`
  Responsible MonALISA Server to receive ApMon UDP packets
- `HOME`
  Your virtual AliEn home directory
- `GBBOX_ENVFILE`
  Environment file keeping shell state variables (e.g. `CWD`)
- `GSHELL_ROOT`
  Installation path of the **API** client package

5.9.5 Single Command Execution from a user shell

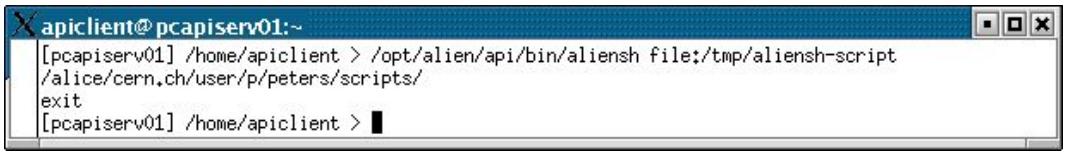
You can execute a single command by invoking the shell with the command as the first parameter:

Return Values ($?):

- 0 Command has been executed successfully
- 1 Command has been executed and returned an error
- 127 Command does not exist and could not be executed
- 255 Protocol/Connection error

5.9.6 Script File Execution from a user shell

You can execute an aliensh script by giving the script filename as the first argument to aliensh. Local script files have to be prefixed with 'file:', otherwise the script is taken from the AliEn file catalog!



The final line of the script output 'exit' is produced, when the aliensh terminates and is not part of the user script.

Return Values ($?):

- 0 Script was executed successfully and returned 0
- 1 Script was executed and returned != 0
- 127 Script does not exist and could not be executed
- 255 Protocol/Connection error

5.9.7 Script execution inside aliensh “run”

If you want to source a script which is stored in the file catalogue or on the local disk, you can use “`run <filename>`” for a file in the catalogue or “`run file:<filename>`” for a file on your local disk. The known '`source`' command sources only local files.

5.9.8 Basic aliensh Commands

whoami

Print the user's role on STDOUT.

Return Values ($?):

- 0 Successfully executed
- 255 Protocol/Connection error

clear

Clear your terminal window

pwd

Print the cwd on stdout

Return Values ($?):

- 0 Successfully executed
- 255 Protocol/Connection error

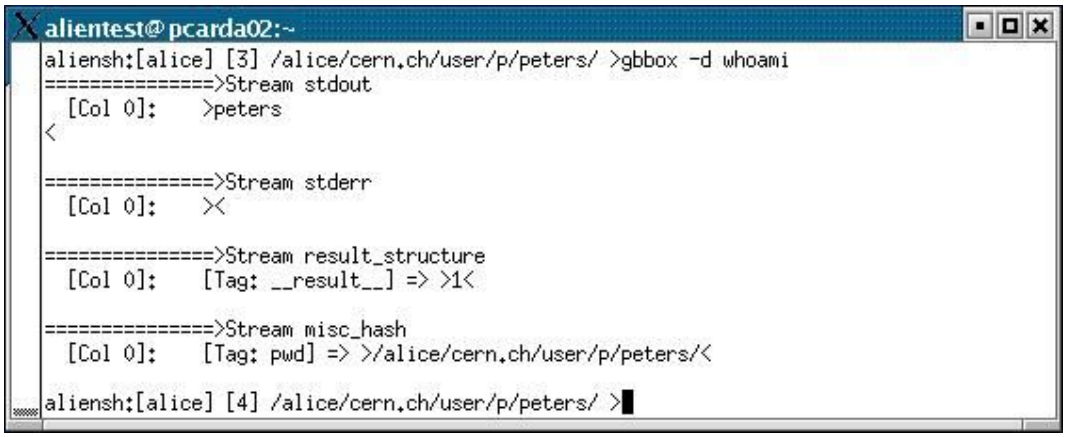
gbbox [-d] [-h] [-f <tag1>,[<tag2>]..]

Busy Box command, used to execute most of the **aliensh** commands. Example: “whoami” executes in fact “gbbox whoami”

The **API** protocol returns always four streams in the response:

- STDOUT
- STDERR
- Results Structure (Array of Key-Value pairs)
- Misc. Hash (Array of Key-Value pairs for environment variables etc.)

The shell prints only the STDOUT and STDERR stream on your terminal. With the “-d” option, you get a dump of all returned streams:



The “-o <tag>” option prints only the tag <tag> of the result stream to your terminal. For Boolean functions the return value is returned with the tag name “__result__”:

This command is very useful to make a quick check, what a busy box command returns - especially with respect to the **API** library, where you can easily access directly any tag in the result structure.

Return Values ($?):

- 0 Command has been executed successfully
- 1 Command has been executed and returned an error
- 127 Command does not exist and could not be executed
- 255 Protocol/Connection error

cd [<directory>]

change to the new working directory <directory>. If the directory is unspecified, you are dropped into your home directory. < directory> can be a relative or absolute PATH name. Like in standard shells 'cd – ' drops you in the previous visited directory – if there is no previous one, into your home directory.

Return Values ($?):

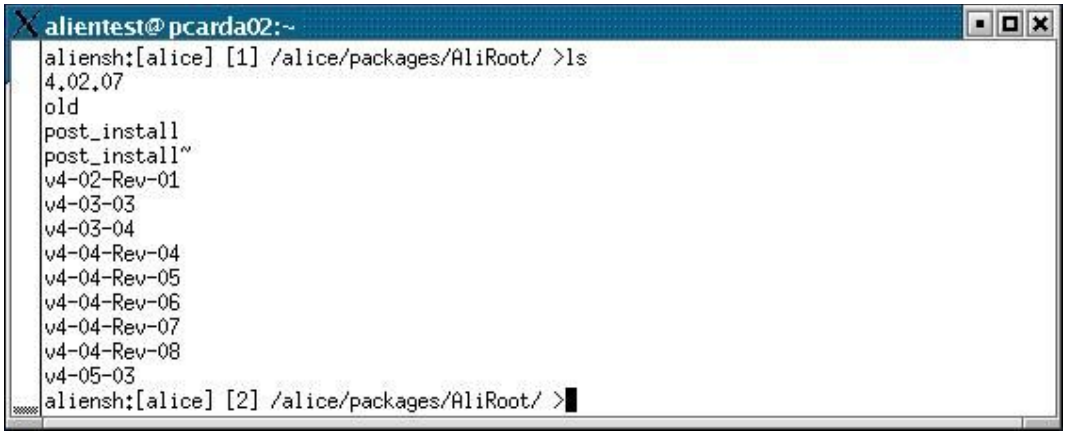
- 0 Directory has been changed
- 1 Directory has not been changed
- 255 Protocol/Connection error

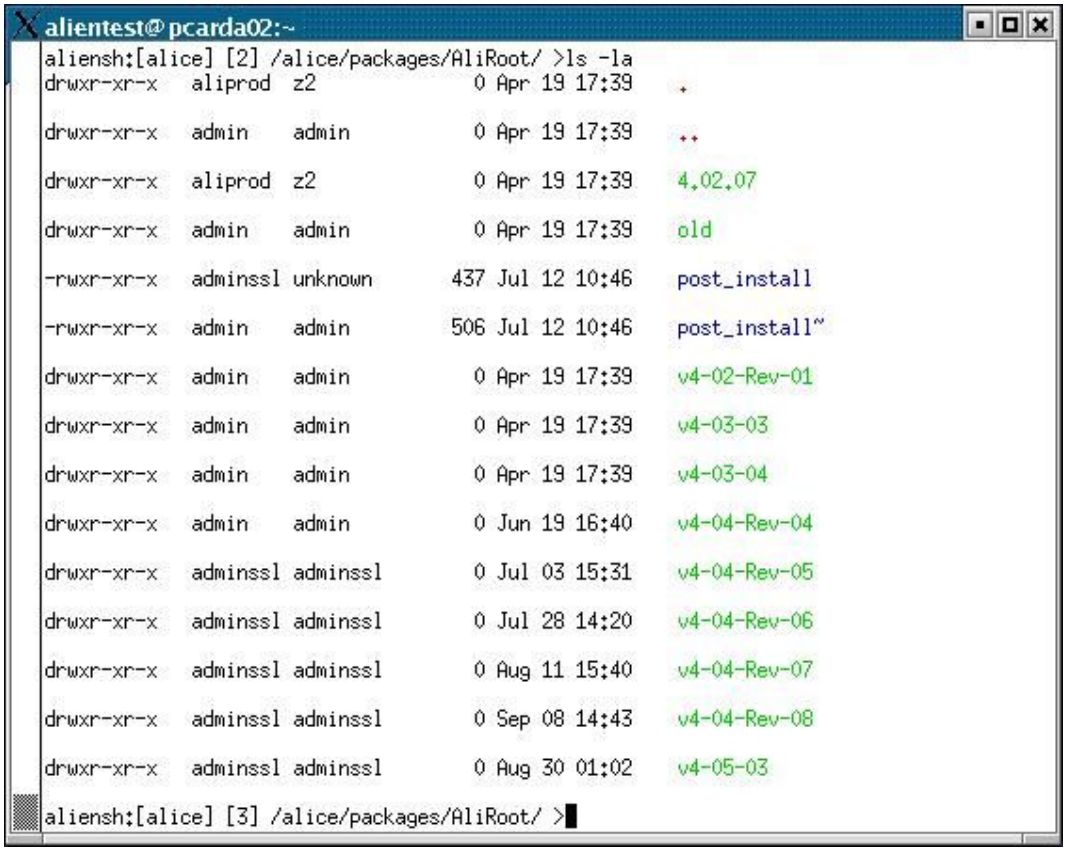
ls [-laF|b|m] [<directory>]

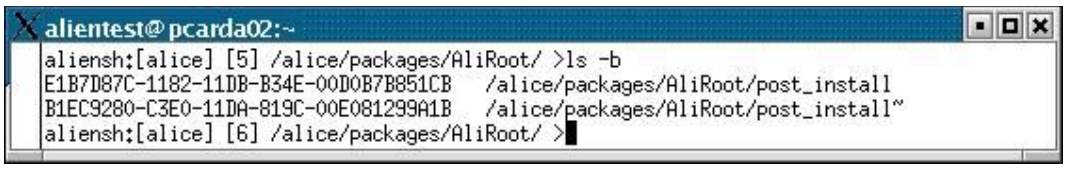
list the current directory or `<directory>`, if specified.

- "-l" list in long format
- the long format uses colour highlighting
  blue plain files
  green directories
  red JDL files
- "-a" list also hidden files starting with "."
- "-b" list in guid format (guid + name)
- "-m" list in md5 format (md5 + name)
- "-F" list directories with a trailing "/"
- "-n" switch off the colour mode for the long format

Note: if you want to use the `ls` function in scripts and parse the output, you should always add the -n flag to switch off the colour mode for the '-l' long format.







Return Values ($?):

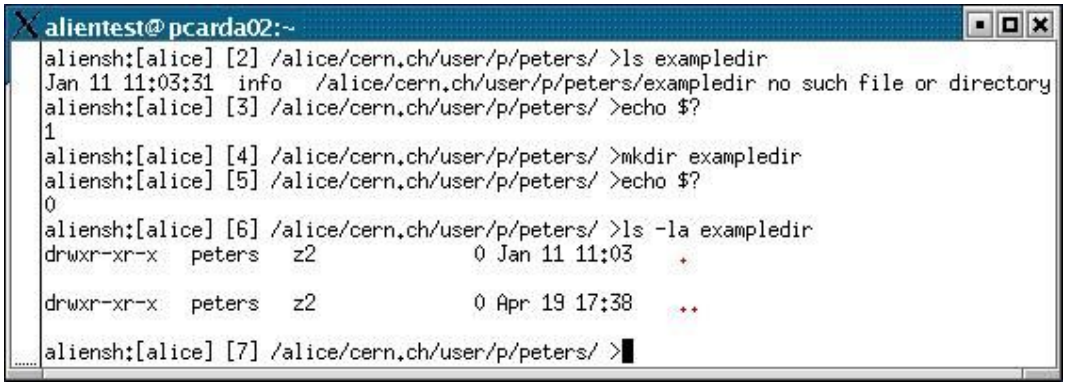
- 0 Directory has been listed

- 1 Directory has not been listed
- 255 Protocol/Connection error

mkdir [-ps] <directory>

create the directory <directory>

- "-p" recursively creates all needed subdirectories
- "-s" silences the command – no error output



Return Values ($?):

- 0 Directory has been created
- 1 Directory has not been created
- 255 Protocol/Connection error

rmdir [-r] <directory>

remove the directory <directory> or everything under the directory tree <directory>

- "-r" recursively removes files and directories

Please note that, if you use the '-r' flag, you remove only entries from the file catalogue, but not their physical files on a mass storage system. Use 'erase' to remove files physically.

5.9.8.1Data Management Commands

cp [-s][-d] [-v] [-m][-n] [-t <time>] [-f] [-i <infofile>]<src> <dst>

copy file(s) from <src> to <dst>: **This command always produces a physical copy of a file. It does not only copy an entry in the catalogue to a new name!**

<src> and <dst> have to be given with the following URL-like syntax:

`[alien:|file:] <path|dir>[@<SE>]`

`Examples:`

- `"alien:*.root"` specifies all '.root' files in the AliEn CWD
- `"alien:/alice/cern.ch/"` specifies the “/alice/cern.ch/” directory in AliEn
- `"file:/tmp/myfile.root"` specifies the local file '/tmp/myfile.root' on your computer
- `"myfile.root@ALICE::CERN::Castor2"` specifies the file “myfile.root” in the CWD in AliEn in the ALICE::CERN::Castor2 Storage Element

If a file has several replicas and you don't specify a source SE, the closest SE specified in the environment variable `alien_close_SE` is taken. If this is not set, the closest one to your client location is configured automatically..

If you don't specify a target SE, the SE specified in the environment variable `alien_close_SE` is taken

If <src> selects more than one file (e.g. `'*.root'`) <dst> must be a directory – otherwise you will get an error

If <src> is a directory (e.g. `'/tmp/mydir/'`) <dst> must also be a directory – otherwise you will get an error

Options:

- “-s” be silent, in case of no error, don’t print anything
- “-d” switch on debug output. This can be useful to find reasons for command failures
- “-v” more verbosity – print summaries about source, destination, size and transfer speed
- “-m” MD5 sum computation/verification

  local to grid file : compute the md5 sum of the local file and insert it into the catalogue. This is turned “on” by default for local to grid file copy operations

  grid to local file : verify the md5 sum of the downloaded file

  local to local file : flag ignored

  grid to grid file : recompute the md5 sum of the source file and enter it in the catalogue with the destination file

- “-n” new version – this option is only active, if the destination is a file in AliEn: if the destination file exists already, the existing destination is moved into the subversions directory (see following example) and labeled with a version number starting with v1.0. If the new version to be copied has the same md5sum, the file is not

copied at all!

Example:

- If you overwrite a file named “myfile” with the “-n” option, the original “myfile” is moved to “.myfile/v1.0” and “myfile” contains the last copied file. If you repeat this, “myfile is moved to “.myfile/v1.1”
  etc.

- “-t” <seconds> specifies the maximum waiting time for all **API** service requests (which are executed within the copy command). The default is 10 seconds.
- “-f” “force” transfer. This sets the first connection timeout to 1 week and the retry timeout (if a connection is interrupted during a transfer) to 60 seconds. The default is 15 s for the first connection timeout and 15 seconds for retries. In interactive shells however, it is more user-friendly, if a copy command times out automatically.
- “-i <file>“ writes an info file locally to <file>, containing the <src> and <dst> parameters used, e.g. you can get the information, from which SE a file has been read, if you didn’t specify it in <src>.

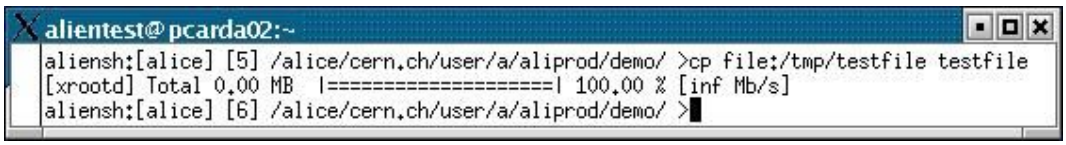
Return Values ($?):

- 0 File(s) copied successfully or you have just requested the “-h” option for the usage information
- 1 Directory has not been listed
- 20 access to the source file has been denied
- 21 access to the destination file has been denied
- 22 timeout while getting an access envelope
- 23 you tried to copy multiple files, but the destination was not a directory
- 24 could not create the target destination directory when copying multiple files
- 25 copy worked, but couldn’t write the info file given by “-i <infofile>“.
- 30 the md5 sum of the source file is not “0” and the md5sum of the downloaded file differs from the catalogue entry.
- 200 Control-C or SIG Interrupt signal received
- 250 an illegal internal operation mode has been set (can happen only in “buggy” installations/setups)
- 251 <src> has not been specified
- 252 <dst> has not been specified

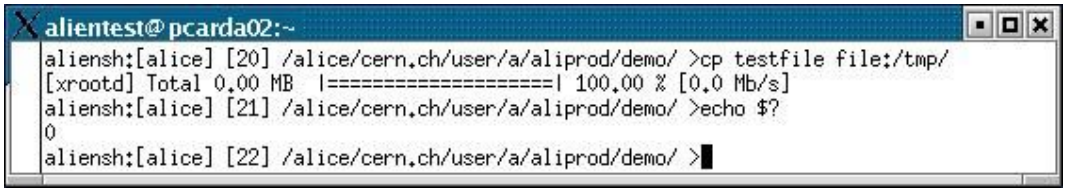
- 253 you miss the xrootd xcp or xrdcp program the first place to search for is in your environment. If it can't be found there, it tries to find it in your **API** installation directory or one level (../) higher
- 255 the copy operation itself to/from the SE failed (xrdcp failed)

Examples:

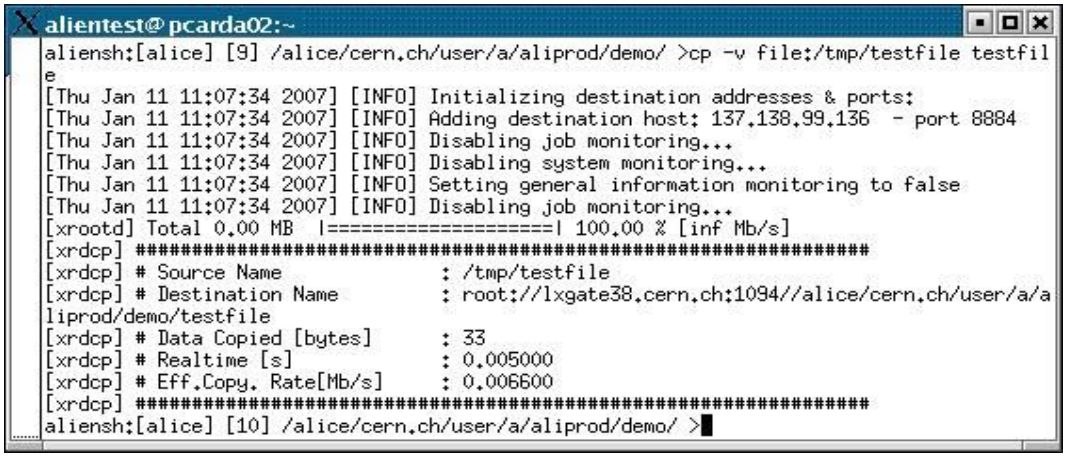
- copy local to AliEn file:



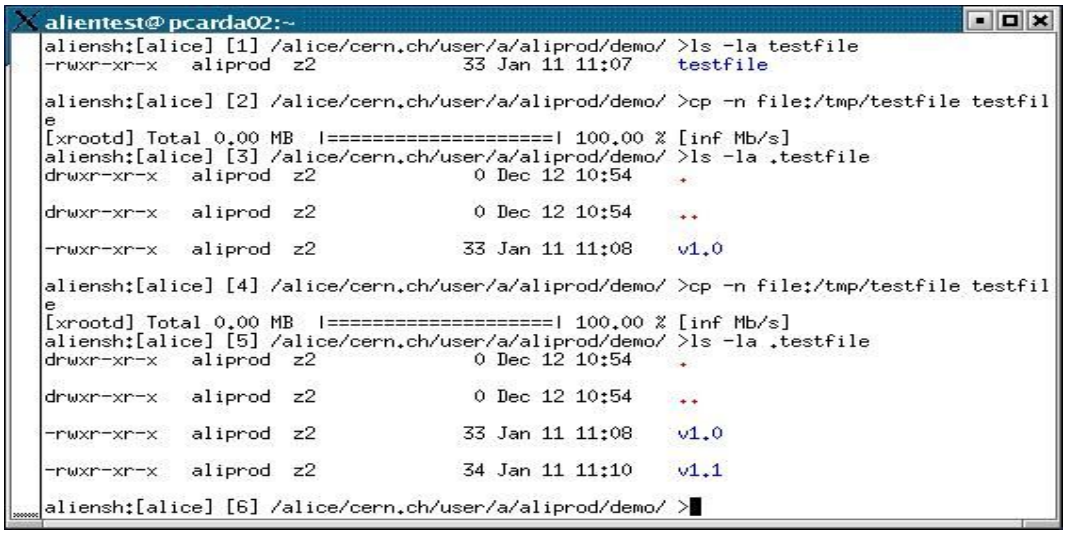
- copy AliEn to local file:



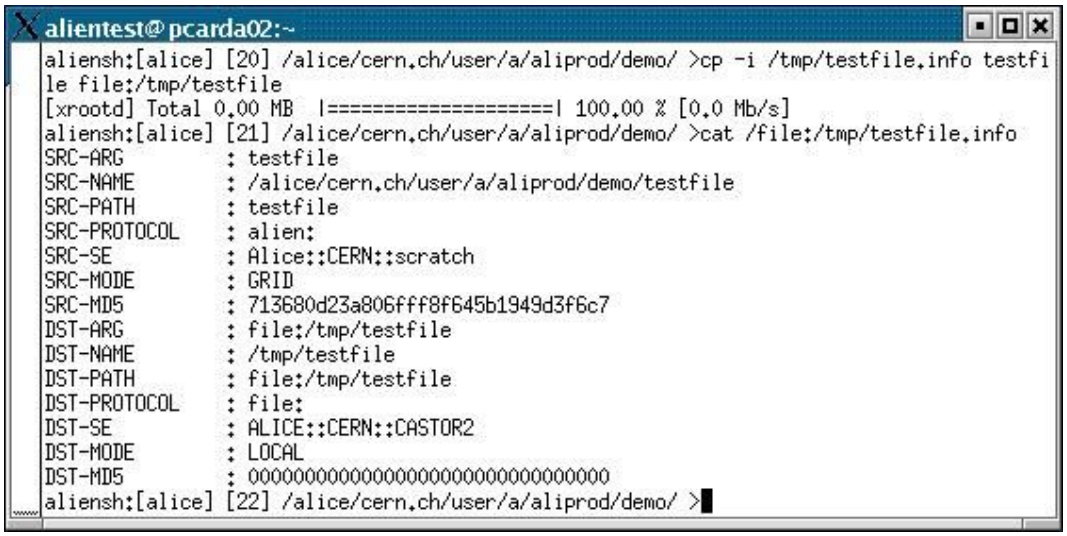
- copy local to AliEn file with verbose option:



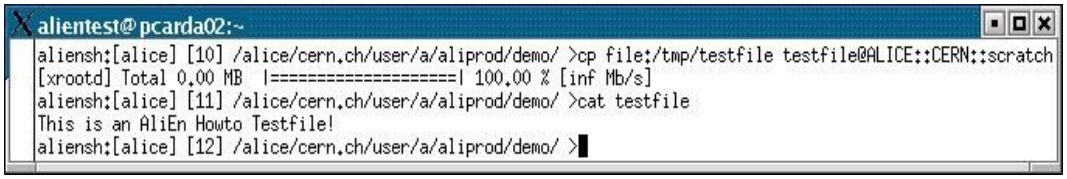
- copy local to existing AliEn file creating new version:



- copy AliEn to local file – write copy information file:



- copy to a specific AliEn storage element:



rm [-s] [-f] <file>

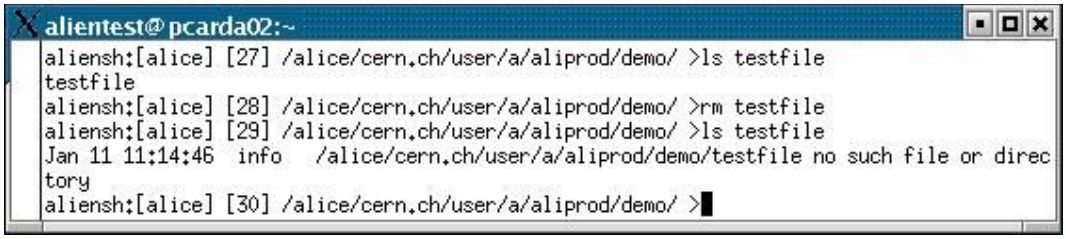
remove the entry <file> from the file catalogue

- "-s" - be silent – don't print ERROR messages!
- "-f" - succeeds even if the file did not exist!

Return Values ($?):

- 0 File has been removed

- 1 File could not be removed
- 252 <file> parameter missing
- 255 Protocol/Connection error



cat <file>

print <file> to STDOUT. <file> has an URL-like syntax and can reference an AliEn or local file:

- "`cat file:/tmp/myfile.txt`" - print the local file "/tmp/myfile.txt"
- "`cat myfile.txt`" - print the AliEn file "myfile.txt" from the CWD
- "`cat alien:/alice/myfile.txt`" - print the AliEn file "/alice/myfile.txt" (the protocol "alien:" is optional from within the shell).

Return Values ($?):

- 0 File has been printed with "cat"
- 1 File could not be printed with "cat"
- 246 the local copy of the file could not be removed
- 250 an illegal internal operation mode was set (can happen only in "buggy" installations/setups)
- 252 <file> parameter missing
- 255 Protocol/Connection error

more <file>

use "more" to print the file <file> on STDOUT. <file> has an URL-like syntax and can reference an AliEn or local file:

- "`more file:/tmp/myfile.txt`" - print the local file "/tmp/myfile.txt"
- "`more myfile.txt`" - print the AliEn file "myfile.txt" from the CWD
- "`more alien:/alice/myfile.txt`" - print the AliEn file "/alice/myfile.txt" (the protocol "alien:" is optional from within the shell).

Return Values ($?):

- 0 File has been printed with "more"

- 1 File could not be printed with “more”
- 246 the local copy of the file could not be removed
- 250 an illegal internal operation mode was set (can happen only in “buggy” installations/setups)
- 252 <file> parameter missing
- 255 Protocol/Connection error

less <file>

print the file <file> to STDOUT. <file> has an URL-like syntax and can reference an AliEn or local file:

- “`less file:/tmp/myfile.txt`” - print the local file “/tmp/myfile.txt”
- “`less myfile.txt`” - print AliEn file “myfile.txt” in the CWD
- “`less alien:/alice/myfile.txt`” - print the AliEn file “/alice/myfile.txt” (the protocol “alien:” is optional from within the shell).

Return Values ($?):

- 0 File has been printed with “less”
- 1 File could not be printed with “less”
- 246 the local copy of the file could not be removed
- 250 an illegal internal operation mode was set (can happen only in “buggy” installations/setups)
- 252 <file> parameter missing
- 255 Protocol/Connection error

edit [-c] <file>

edit local or AliEn files using your preferred editor. The file <file> is copied automatically into the /tmp directory, and you work on this local copy. If you close your editor, the file is saved back to the original location, if you have modified it. The default editor is “vi”. You can switch to another editor by setting the environment variable **`EDITOR:`**

- for vim: EDITOR="vim"
- for emacs: EDITOR="emacs" or "emacs -nw"
- for xemacs: EDITOR="xemacs" or "xemacs -nw"
- for pico: EDITOR="pico"

Note: Change this setting in the local aliensh rc-file **`$HOME/.alienshrc`** ! <file> has an URL-like syntax and can reference an AliEn or a local file:

- “`edit file:/tmp/myfile.txt`” - edit the local file

"/tmp/myfile.txt"

- "`edit myfile.txt`" - edit the AliEn file "myfile.txt" in the CWD
- "`edit alien:/alice/myfile.txt`" - edit the AliEn file "/alice/myfile.txt" (the protocol "alien:" is optional from within the shell).
- "`edit alien:/alice/myfile.txt@ALICE::CERN::Tmp`" - edit the AliEn file "/alice/myfile.txt". The file is read preferably from the SE "`ALICE::CERN::Tmp`", if this is not possible from another "closest" SE. The file will be written back into "`ALICE::CERN::Tmp`".

AliEn files are per default written back into the same storage element, unless you specify a different one by appending "@<SE-Name>" to <file>.

Every modified AliEn file is saved as a new version in the file catalog. See the "cp -v" command for information about versioning.

Of course, you can only edit files that exist. If you want to edit a new empty file use:

- "-c": create a new empty file and save it to <file>. If <file> is a local file and already exists, it is overwritten with the newly edited file. If <file> is an AliEn file and exists, it is renamed to a different version and your modified version is stored as <file>
- "-h": print the usage information for this command

Return Values ($?):

- 0 File has been edited and rewritten, or the "-h" flag
- 1 File could not be written back. You get information about your temporary file to rescue it by hand.
- 246 the local copy of the file could not be removed
- 250 an illegal internal operation mode was set (can happen only in "buggy" installations/setups)
- 252 <file> parameter missing

erase <file>

remove physically all replicas of <file> from storage elements and the catalogue entry

Return Values ($?):

- 0 File has been erased
- 1 File could not be erased
- 255 Protocol/Connection error

purge <file>|<directory>

with <file> parameter: removes all previous versions of <file> except the latest

with <directory> parameter: same as above for each file contained in <directory>

Return Values ($?):

- 0 File has been purged | directory has been purged
- 1 File or Directory could not be purged – see output message for details
- 255 Protocol/Connection error

whereis [-l] <file>

list all replicas of file <file>. It includes the GUID, the TURL and the SE name.

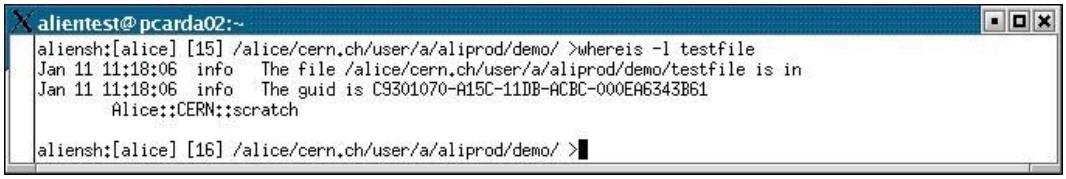
- "-l" list only the names of the SEs and the GUID
- "-h" print the usage information

Return Values ($?):

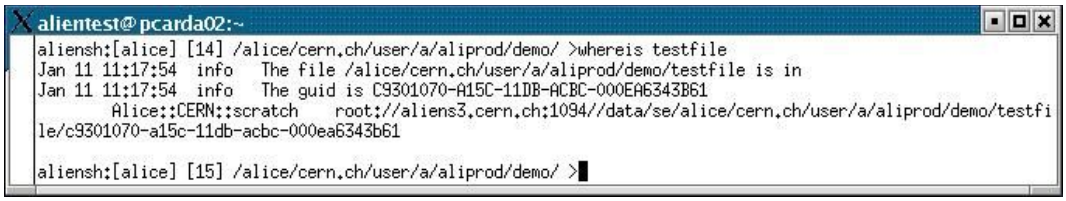
- 0 Command successful
- 1 Command failed
- 255 Protocol/Connection error

Example:

- locate a file on SEs only:



- locate a files GUIDs/TURLs and SEs:



- advanced usage: use the busy box command, to print only one output field of the command:

mirror <lfn> <se>

If you want to replicate files from one SE to another, you can use the mirror command. <lfn> is the file you want to replicate, <se> is the target storage element.

Return Values ($?):

- 0 Command successful
- 1 Command failed
- 255 Protocol/Connection error

df [<SE-name>]

Report the disk space usage of the default SE or <SE-name>

Return Values ($?):

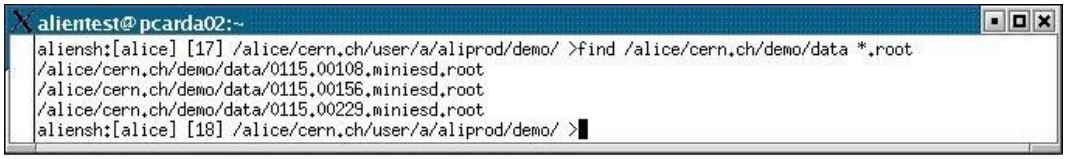
- 1 in any case

find [-<flags>] <path> <fileName|pattern> [[<tagname>:<condition>] [[and|or] [<tagname>:<condition>]]*]

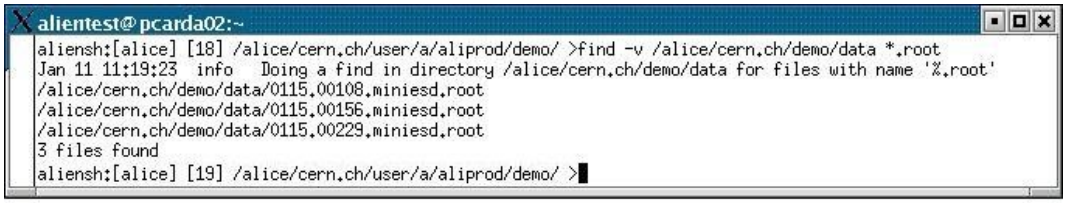
helps to list catalogue entries according to certain criteria. The search is always tree-oriented, following the hierarchical structure of the file catalogue (like the UNIX find command).

The simplest find syntax is :

find /alice/cern.ch/demo/data *.root

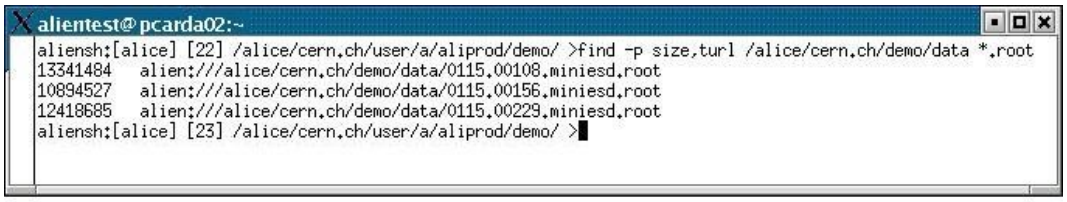


Use the ‘%’ or the ‘*’ character as a wildcard. You can switch on the summary information with the '-v' flag:



Depending on your needs you can also select, which information you want to

print per entry using the '-p <field1>[,<field2>[,<field2>...]]' option. E. g. :



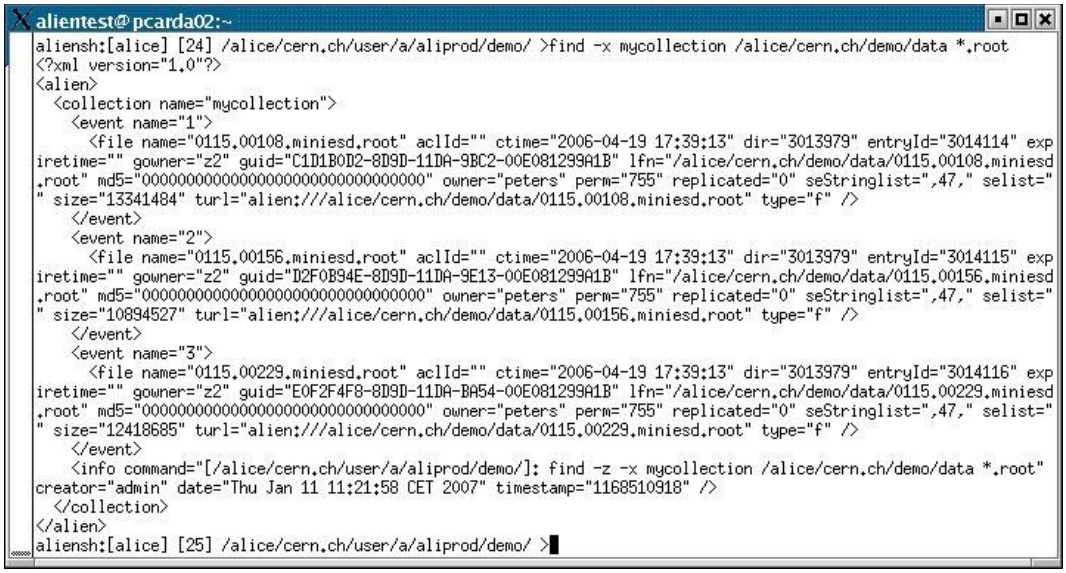
Available fields are (only the 'interesting' ones are commented):

seStringlist	
aclId	
lfn	logical AliEn file name
dir	
size	size of the file in bytes
gowner	Group
guid	the GUID of the file
owner	owner (role)
ctime	
replicated	
entryId	
expiretime	
selist	
type	
md5	the MD5 sum of that file
perm	the file permissions in text format

If you add the '-r' flag, you can print additional fields describing the location of that file:

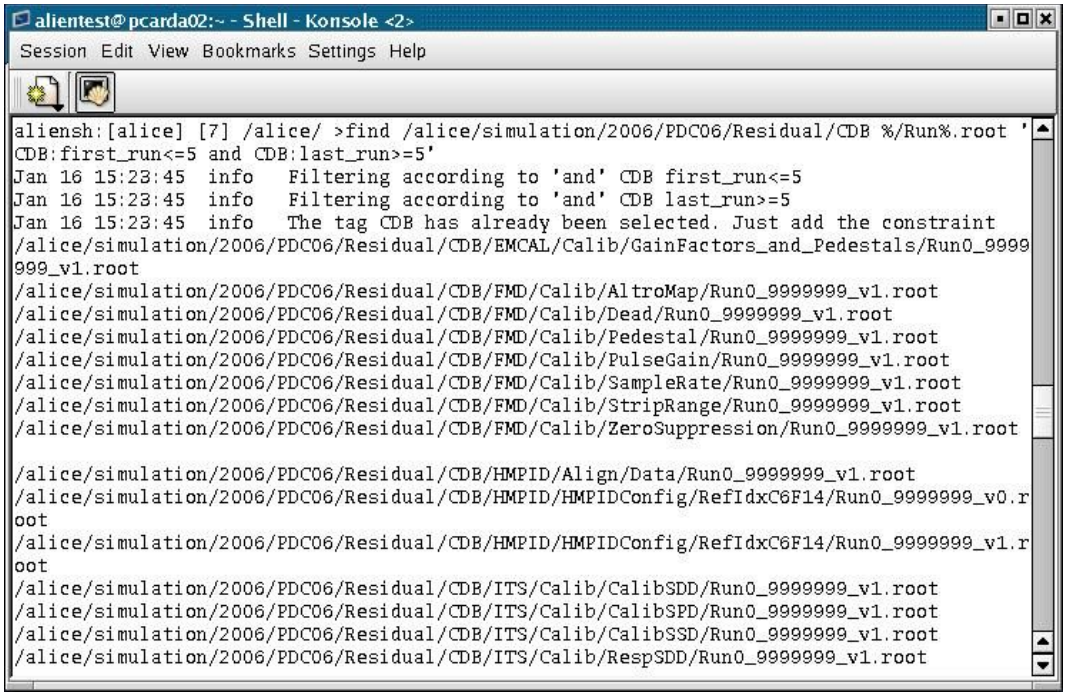
- longitude
- latitude
- location
- msd (mass storage domain)
- domain

In combination with ROOT the flag '-x <collection name>' is useful since it prints the result as XML collection, which can be read directly by ROOT.

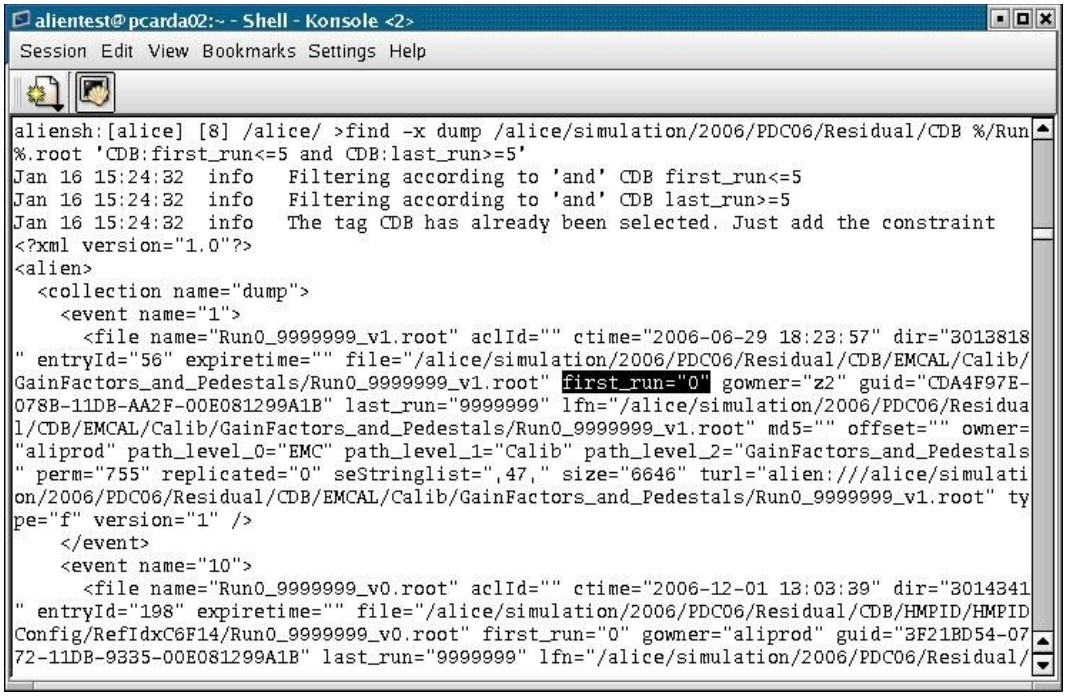


You can easily store this in a local file 'find ..... > /tmp/collection.xml'.

Additional metadata queries can be added using the tagname:condition syntax. Several metadata queries can be concatenated with the logical 'or' and 'and' syntax. You should always single-quote the metadata conditions, because < > are interpreted by the shell as pipes. The following example shows a metadata query on the tag database:



The same query as above, but this time with the XML output format. All metadata information is returned as part of the XML output file:



5.9.8.2Metadata Commands

Schema Definition

To tag metadata in AliEn, you first need to create a schema file defining the metadata tag names and variable types. You find existing tag schema files for your VO in the AliEn FC in the directory "/<VO>/tags/" or in your user home directory "~/tags".

The syntax used to describe a tag schema is:

```
<tagname 1>  <variable type 1> [,<tarname 2> <variable type 2> …]
```

<variable type> are SQL types and can be e.g.

- char(5)
- int(4)
- float
- text

If you want to create your own tag schema you will have to copy a schema file into the tags directory in your home directory. You might need to create it, if it does not exist.

To tag your files e.g. with some description text, create a tag file ~/tags/description containing:

addTag [-d] <directory> <tag name>

add the schema <tag name> to <directory>. The schema file called <tag name> must be saved in the AliEn FC under '/<VO>/tags' or in your home directory '~/tags'. All subdirectories will inherit the same schema. By default, all directories using the same schema store the metadata information in the same table. The '-d' flag creates a separate database table for the metadata of schema <tag name>.

Return Values ($?):

- 0 Command successful
- 1 Command failed
- 255 Protocol/Connection error

showTags <directory>

Shows all defined tags for <directory>.

Return Values ($?):

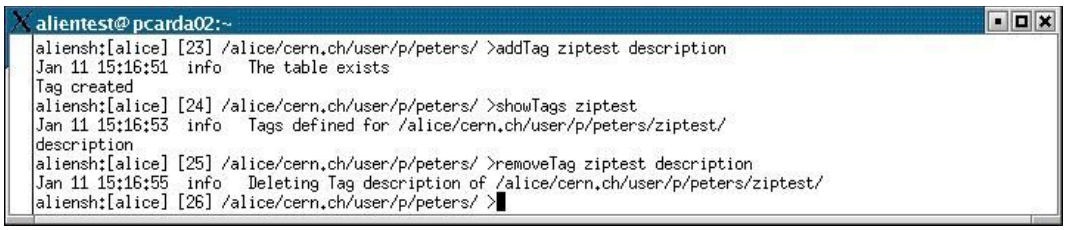
- Command successful
- Command failed
- Protocol/Connection error

removeTag <directory> <tag name>

Removes the tag <tag name> from <directory>. All defined metadata will be dropped.

Return Values ($?):

- 0 Command successful
- 1 Command failed
- 255 Protocol/Connection error



addTagValue <file> <tag name> <variable>=<value> [<variable>=<value> …]

Sets the metadata <variable> to <value> in schema <tag name> for <file>. <tag name> must be an existing metadata schema defined for a parent directory of <file>, otherwise the command will return an error.

Return Values ($?):

- 0 Command successful
- 1 Command failed
- 255 Protocol/Connection error

showTagValue <file> <tag name>

Shows the tag values for <file> from metadata schema <tag name>.

Return Values ($?):

- 0 Command successful
- 1 Command failed
- 255 Protocol/Connection error

removeTagValue <file> <tag name>

Removes a tag value :

File Catalogue Trigger

AliEn allows you to trigger actions on **insert**, **delete** or **update** events for a certain directory tree. A trigger action is a script registered in the file catalogue under `/<VO>/triggers` or `~/triggers`, which receives the full logical file name of the modified entry as a first argument when invoked.

addTrigger <directory> <trigger file name> [<action>]

Adds a trigger <trigger file name> to <directory>. <action> can be **insert**, **delete** or **update** (default is insert).

showTrigger <directory>

Shows the defined trigger for <directory>.

removeTrigger <directory> [<action>]

Removes the trigger with <action> from <directory>.

5.9.8.3Job Management Commands

top [-status <status>] [-user <user>] [-host <exechost>] [-command <commandName>] [-id <queueId>] [-split <origJobId>] [-all] [-all_status]

print job status information from the AliEn task queue

-status <status>	print only jobs with <status>
-user <user>	print only jobs from <user>
-host <exechost>	print only hosts on <exechost>
-command <command>	print only tasks executing <command>
-id <queueId>	print only task <queueId>
-split <masterJobId>	print only tasks belonging to <masterJobId>
-all	print jobs of all users
-all_status	print jobs in any state

Return Values ($?):

- anything not 255 Command has been executed
- 255 Protocol/Connection error

ps [.....]

similar functionality to 'top': report process states

If the environment variable `alien_NOCOLOUR_TERMINAL` is defined, all output will be black&white. This is useful, if you want to pipe or parse the output directly in the shell.

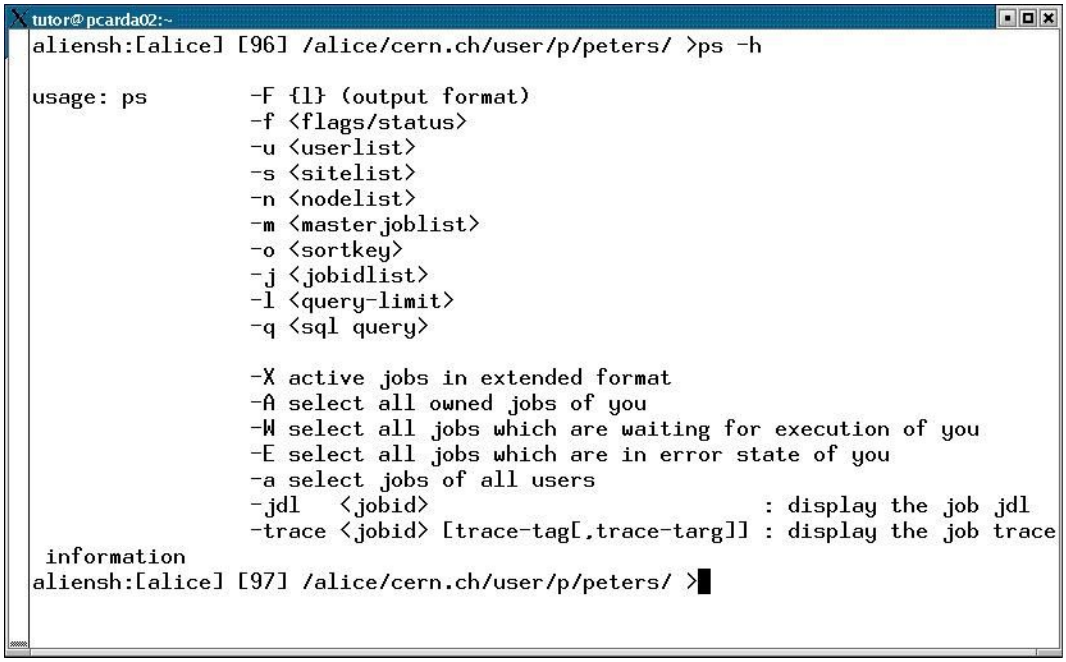
The following options are defined (parameters like <list> are comma separated names or even just a single name):

-F {l}	l = long (output format)
-f <flags/status>	e.g. -f DONE lists all jobs in status 'done'
-u <userlist>	list jobs of the users from <userlist>. -u % selects jobs from ALL users!
-s <sitelist>	lists jobs which are or were running in <sitelist>
-n <nodelist>	lists jobs which are or were running in <nodelist>
-m <masterjoblist>	list all sub-jobs which belong to one of the master jobs in <masterjoblist>
-o <sortkey>	execHost, queueId, maxrsize, cputime, ncpu, executable, user, sent, split, cost, cpufamily, cpu, rsize, name, spyurl, commandArg, runtime, mem, finished, masterjob, status, splitting, cpuspeed, node, error, current, received, validate, command, splitmode, merging, submitHost, vsize, runtimes, path, jdl, procinfotime, maxvsize, site, started, expires
-j <jobidlist>	list all jobs with from <jobidlist>.
-l <query-limit>	set the maximum number of jobs to query. For a non-privileged user the maximum is 2000 by default

-q <sql query>	if you are familiar with the table structure of the AliEn task queue, you can specify your own SQL database query using the keys mentioned above. If you need a specific query, ask one of the developers for help.
-X	active jobs in extended format
-A	select all your owned jobs in any state
-W	select all YOUR jobs which are waiting for execution
-E	select all YOUR jobs which are in error state
-a	select jobs of ALL users
-jdl <jobid>	display the job jdl of <jobid>.
-trace <jobid> [trace-tag[,trace-tag]]	display the job trace information. If tags are specified, the trace is only displayed for these tags per default, all proc tags are disabled. to see the output with all available trace tags, use ps -trace <jobid> all. the available tags are: proc resource information state job state changes error Error statements trace job actions (downloads etc.) all output with all previous tags

Return Values ($?):

- 0 Command has been executed
- 255 Wrong command parameters specified

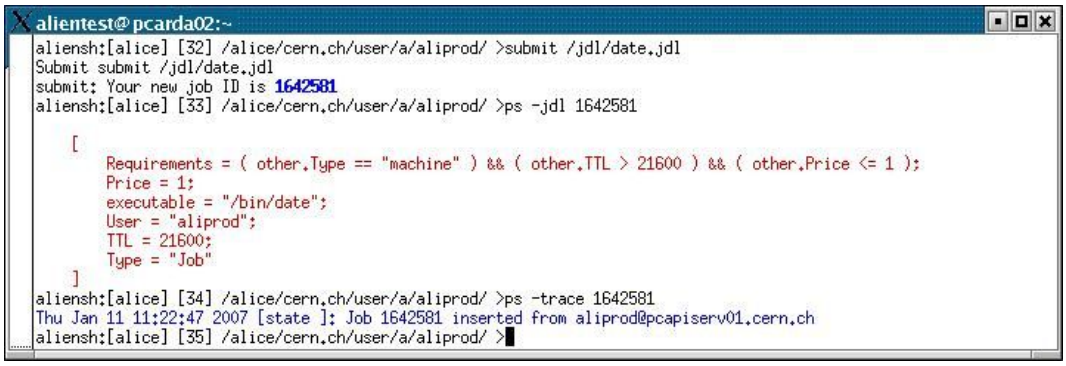


submit [-h] <jdl-file>

submits the jdl file <jdl-file> to the AliEn task queue.

local jdl files are referenced using "`file:<local-file-name>`"

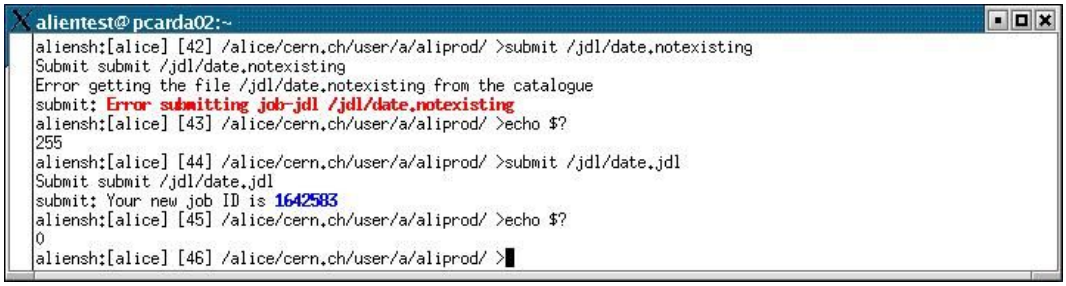
AliEn files are referenced by "`<alien-file-name>`" or "`alien:<alien-file-name>`".



Return Values ($?):

- 0 Submission successful
- 255 Submission failed

Warning: local JDL files can only be submitted, if they don't exceed a certain size. In case you reference thousands of input data files, it is safer to register this JDL file in the file catalogue and submit it from there. The proper way to access many input data files is to use the **InputDataCollection** tag and to register an XML input data collection in AliEn as explained later on.



kill <job-id>

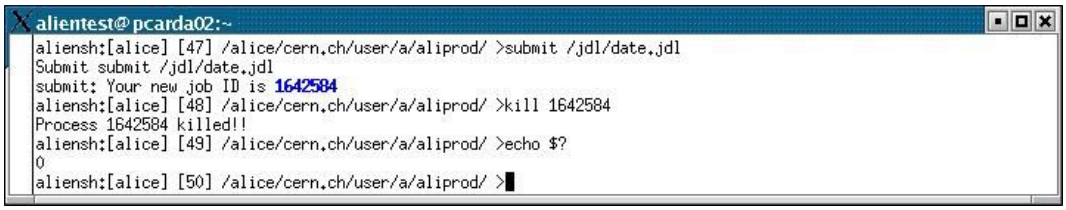
kills a job from the AliEn task queue

<job-id> can be a 'single' or a 'master' job. Killing a 'master' automatically kills all it's sub jobs.

to be able to kill a job, you must have the same user role like the submitter of that job or be a privileged user.

Return Values ($?):

- 0 Job successfully killed
- 255 Job couldn't be killed



queue [info, list, priority .....]

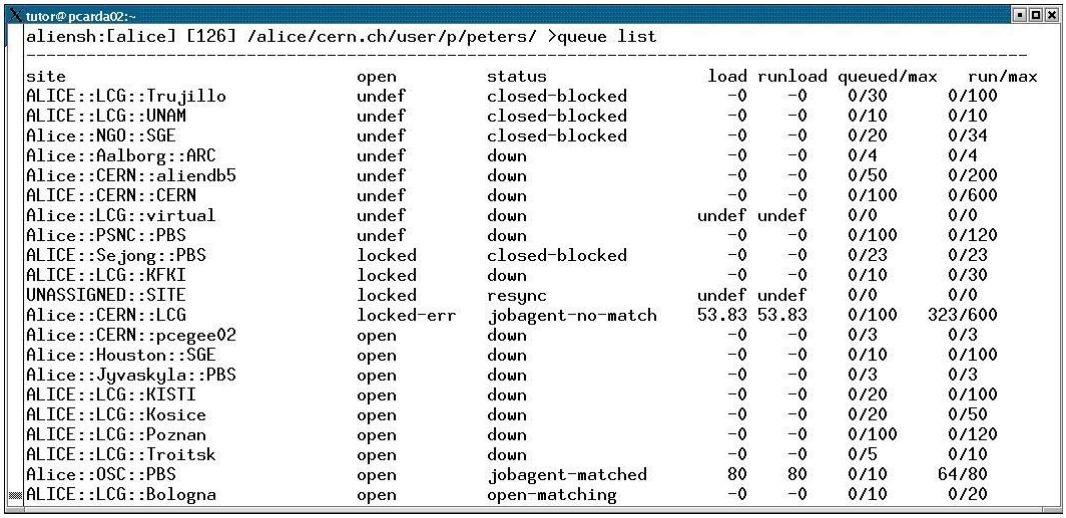
provides TaskQueue state information and other parameters to the user. It accepts three subcommands:

`queue **info** [<Queue Name>]`

prints for one <Queue Name> or all sites the status information and counters of individual job states. Here you can see e.g. if the site, where you want to run your GRID job is currently blocked, closed or in any error state.

`queue **list** [<Queue Name>]`

prints for <Queue Name> or all sites status information, configuration and load parameters.

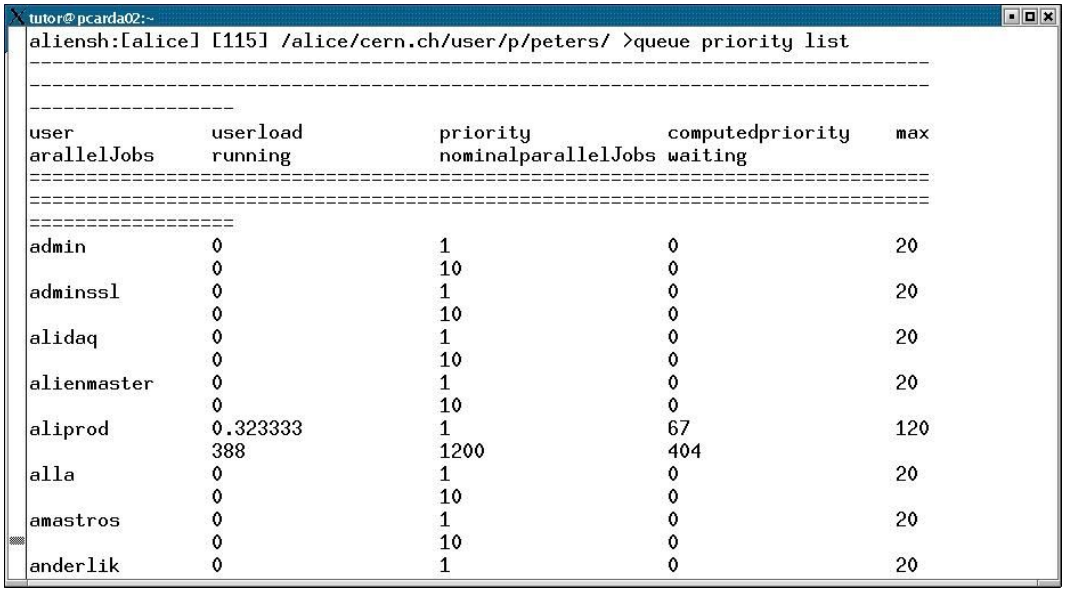


```
tutor@pcarda02:~
aliensh:[alice] [126] /alice/cern.ch/user/p/peters/ >queue list
------------------------------------------------------------------------------------------------------
site                              open        status               load runload queued/max   run/max
ALICE::LCG::Trujillo              undef       closed-blocked         -0    -0    0/30       0/100
ALICE::LCG::UNAM                  undef       closed-blocked         -0    -0    0/10       0/10
Alice::NGO::SGE                   undef       closed-blocked         -0    -0    0/20       0/34
Alice::Aalborg::ARC               undef       down                   -0    -0    0/4        0/4
Alice::CERN::aliendb5             undef       down                   -0    -0    0/50       0/200
ALICE::CERN::CERN                 undef       down                   -0    -0    0/100      0/600
Alice::LCG::virtual               undef       down                undef undef    0/0        0/0
Alice::PSNC::PBS                  undef       down                   -0    -0    0/100      0/120
ALICE::Sejong::PBS                locked      closed-blocked         -0    -0    0/23       0/23
ALICE::LCG::KFKI                  locked      down                   -0    -0    0/10       0/30
UNASSIGNED::SITE                  locked      resync              undef undef    0/0        0/0
Alice::CERN::LCG                  locked-err  jobagent-no-match   53.83 53.83    0/100    323/600
Alice::CERN::pcegee02             open        down                   -0    -0    0/3        0/3
Alice::Houston::SGE               open        down                   -0    -0    0/10       0/100
Alice::Jyvaskyla::PBS             open        down                   -0    -0    0/3        0/3
ALICE::LCG::KISTI                 open        down                   -0    -0    0/20       0/100
ALICE::LCG::Kosice                open        down                   -0    -0    0/20       0/50
ALICE::LCG::Poznan                open        down                   -0    -0    0/100      0/120
ALICE::LCG::Troitsk               open        down                   -0    -0    0/5        0/10
Alice::OSC::PBS                   open        jobagent-matched       80    80    0/10      64/80
ALICE::LCG::Bologna               open        open-matching          -0    -0    0/10       0/20
```

`queue` **`priority`**

`queue priority` **`list`** `[<user-name>]`

prints for one (<user-name>) or all users information about their priorities. Priorities are specified by a nominal and maximal value of parallel jobs. Currently, the maximum is not enforced. Initially, every new user has both parameters set to 20. If the system is not under load, you will be able to run more than 20 jobs in parallel. If the system is loaded in such a way, that all nominal jobs fill up exactly the capacity, you can run exactly 20 jobs in parallel. If the system is running above the nominal load, you will run less than 20 jobs in parallel, according to a fair share algorithm. If you need a higher job quota, contact the ALICE VO administrators.



```
tutor@pcarda02:~
aliensh:[alice] [115] /alice/cern.ch/user/p/peters/ >queue priority list
------------------------------------------------------------------------------
------------------------------------------------------------------------------
------------------
user            userload            priority            computedpriority    max
arallelJobs     running             nominalparallelJobs waiting
==============================================================================
==============================================================================
==================
admin           0                   1                   0                   20
                0                   10                  0
adminssl        0                   1                   0                   20
                0                   10                  0
alidaq          0                   1                   0                   20
                0                   10                  0
alienmaster     0                   1                   0                   20
                0                   10                  0
aliprod         0.323333            1                   67                  120
                388                 1200                404
alla            0                   1                   0                   20
                0                   10                  0
amastros        0                   1                   0                   20
                0                   10                  0
anderlik        0                   1                   0                   20
```

`queue priority` **`jobs`** `[<user-name>|%] [<max jobs>]`

prints the job ranking for all jobs or <user-name>. <max jobs> parameter limits the list, if there are too many jobs in the queue. If you want to see the ranking of the ten jobs which are to be executed next, if picked up by an agent, do

```
queue priority jobs % 10
```

spy <job id> workdir|nodeinfo|<sandbox file name>

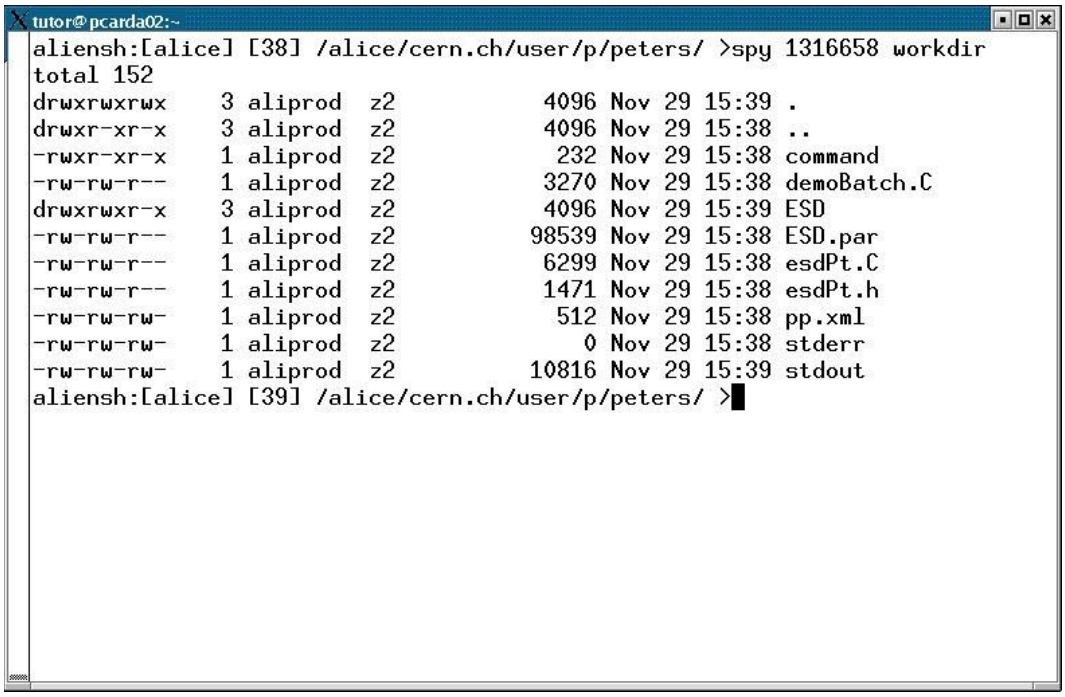
allows to inspect running jobs. The first argument is the job ID you want to inspect. The second argument can be:

`Workdir` lists the contents of the working directory

registerOutput <job id>

Failing jobs do not automatically register their output in the file catalogue. If you want to see (log) files of failed jobs you can use this command.

Output files are registered in your current working directory. It is convenient, to create an empty directory and change that before executing this command.

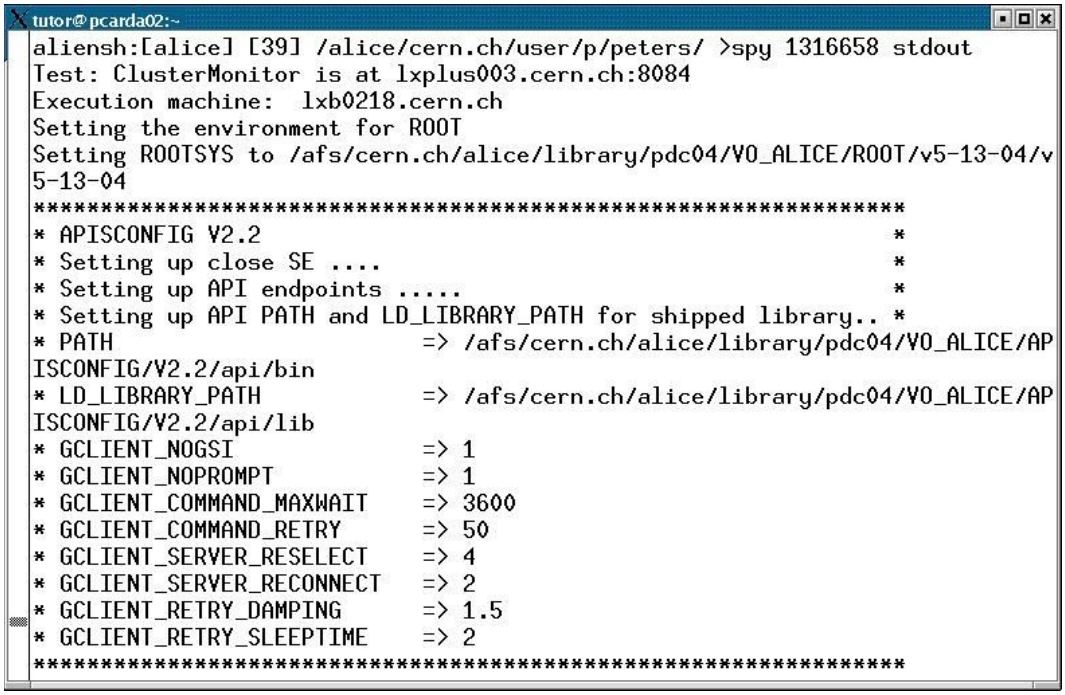


`nodeinfo`

display information about the worker node where the job <job ID> is running

<sandbox file name>

can be e.g. **stdout**, **stderr** or any other file name existing in the sandbox. Be careful not to spy big files and be aware that the contents is printed as text onto your terminal.



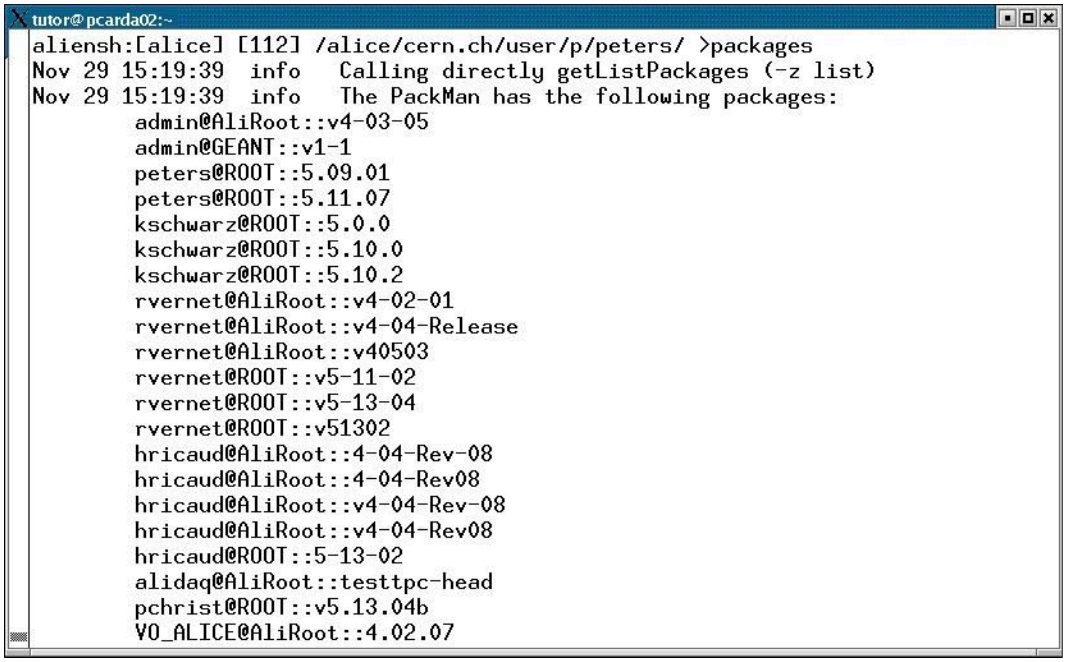
5.9.8.4Package Management

packages

This command lists the available packages defined in the package management system.

Return Values ($?):

- 0 Command has been executed
- 255 Protocol/Connection error



5.9.8.5Structure and definition of Packages

You select a certain package by specifying in your job description:

```
Packages={ <package name 1> [, <package name 2> ...] };
```

Packages are divided into user and VO packages. The VO packages can be found in the file catalogue under `/alice/packages` , while user packages are stored in the home directory of users under `$HOME/packages`.

5.9.8.6Create an AliEn package from a standard ROOT CVS source

If you want to publish your self-compiled ROOT version:

- cd root/
- make dist
- cd ../
- create a file ".alienEnvironment"
- unzip the dist file created by ROOT: e.g. `unzip root*.tgz`
- add the .alienEnvironment file: `tar rvf root*.tar .alienEnvironemnt`
- zip the ROOT archive file: `gzip root*.tar`
- publish the new package in AliEn as your private version 5.10.0 :
- `mkdir -p $HOME/packages/ROOT/5.10.0/`
- `cp file:<root-tgz> $HOME/packages/ROOT/5.10.0/`uname`-`uname -p``

```
##################### PackMan Setup File for ROOT ####################
echo "*** PackMan Setup Start  ***"
```

```
export ROOTSYS=$1/root
echo
"*****************************************************************"
echo "ROOTSYS set to $ROOTSYS"
export PATH=$ROOTSYS/bin:$PATH
echo
"*****************************************************************
"
echo "PATH set to $PATH"
export LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH
echo
"*****************************************************************"
echo "LD_LIBRARY_PATH set to $LD_LIBRARY_PATH"
echo
"*****************************************************************"
echo "*** PackMan Setup End ***"
# The following two lines MUST be there!
shift
$*
```

5.9.8.7Define Dependencies

To define dependencies, you first need to add the metadata schema `PackageDef` to your software directory `~/packages`. Then, you add another software package as a dependency by adding the tag variable 'dependencies' to your package version directory. Several packages can be referenced by a comma separated list.

Example:

```
addTag PackageDef ~/packages/ROOT
addTagValue 5.11.07 PackageDef depencies="VO_ALICE@APISCONFIG::V2.2"
```

5.9.8.8Pre- and Post-Installation scripts

Pre- and Post-Installation scripts are defined as dependencies via metadata tags on the software version directory. The schema is again PackageDef, the tags are:

- pre_install
- post_install

The assigned tag value references the pre-/post-installation script with its logical AliEn file name.

5.10The ROOT AliEn Interface

5.10.1Installation of ROOT with AliEn support

This document proposes three different ways to install ROOT. If you develop within the ROOT framework and you need to modify ROOT itself, it is recommended to follow 5.10.1.1. If you don't need to develop parts of ROOT, you can use 5.10.1.2 (which recompiles ROOT on your machine) or 5.10.1.3 (which installs a precompiled binary).

5.10.1.1Manual Source Installation from CVS

- Login to the ROOT CVS server with 'cvs' as password:

```
cvs -d :pserver:cvs@root.cern.ch:/user/cvs login
```

- Checkout the ROOT source code, either the CVS Head:

```
cvs -d :pserver:cvs@root.cern.ch:/user/cvs co root
```

  Or a tagged version (≥ v5-10-00):

```
cvs -d :pserver:cvs@root.cern.ch:/user/cvs -r v5-10-00 co root
```

- The AliEn module in ROOT needs GLOBUS to be enabled. You need to set the environment variable GLOBUS_LOCATION, e.g. the version installed by the AliEn installer is referenced by:

```
export GLOBUS_LOCATION=/opt/alien/globus (or setenv)
```

- Run the configure script enabling AliEn:

```
./configure --enable-alien
```

- the script will look for the **API** package in the default location `'/opt/alien/api'`. If the **API** is installed in another location, you can specify this using `'--with-alien-incdir=<>'` and `'--with-alien-libdir=<>'` options. E.g. if you have the **API** installed under $HOME/api, execute:

```
./configure --enable-alien
      --with-alien-incdir=$HOME/include
      --with-alien-libdir=$HOME/lib
```

- Compile ROOT

```
make
```

- For all questions concerning the ROOT installation in general, consult the ROOT web page http://root.cern.ch

5.10.1.2Source installation using AliEnBits

Follow the instructions in 5.3.3 which explain how to install the AliEnBits framework until (including) it comes to the '`configure`' statement. Change into the ROOT application directory and start the compilation

```
cd $HOME/alienbits/apps/alien/root
make
```

If you previously installed the **API** via AliEnBits, the ROOT configuration and compilation will start immediately. If not, the AliEnBits system will start to download all dependent packages and compile them beforehand.

Note: the AliEnBits system will install the ROOT version defined in the build system for the AliEn release you are using. It is defined as the symbol '`GARVERSION`' in `$HOME/alienbits/apps/alien/root/root/Makefile`. You cannot easily switch to another CVS tag following this procedure.

5.10.1.3Binary Installation using the AliEn installer

Follow the steps in 5.3.1 but select 'ROOT' as the package to be installed. If you selected the default localtion '`/opt/alien/`' in the installer, you will find ROOT installed under '`/opt/alien/root`'.

5.10.2ROOT Startup with AliEn support - a quick test

To validate your installation, do the following test:

Use the `alien-proxy-init` command to retrieve a shell token from an **API** service (see chapter 5.6).

It is convenient to write a small script for the ROOT start-up:

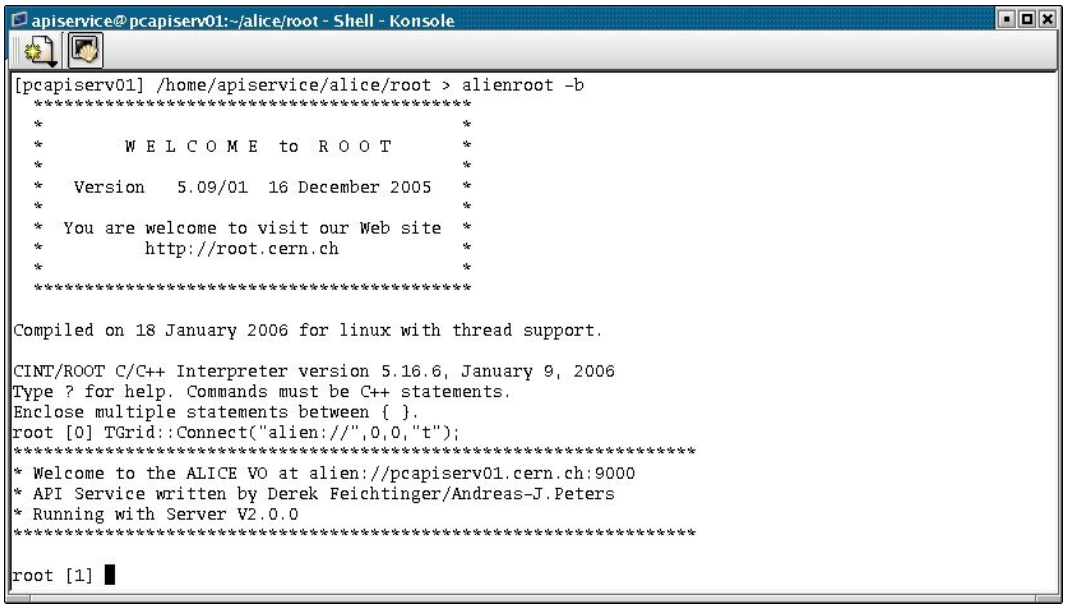
```
#!/bin/bash
test -z $ROOTSYS && export ROOTSYS=/opt/alien/root
export PATH=$ROOTSYS/bin:$PATH
export
LD_LIBRARY_PATH=$ROOTSYS/lib:$LD_LIBRARY_PATH:/opt/alien/api/lib
if [ -e /tmp/gclient_env_$UID ]; then
             source /tmp/gclient_env_$UID;
      root $*
fi
```

If you got the ROOT prompt, execute

```
TGrid::Connect("alien:",0,0,"t");
```

This uses your present session token and prints the 'message of the day (MOTD)' onto your screen. This method is described more in detail in the following subsections.

If you got here, everything is properly configured.



5.10.3The ROOT TGrid/TAlien module

ROOT provides a virtual base class for GRID interfaces encapsulated in the TGrid

class. The ROOT interface to AliEn is implemented in the classes **TAlien** and **TAlienFile**. **TAlien** is a plug-in implementation for the **TGrid** base class and is loaded, if you specify 'alien:' as the protocol to load in the static factory function of **TGrid**. **TAlienFile** is a plug-in implementation for the **TFile** base class.

E.g. the factory function for **TGrid** is the 'Connect' method:

```
TGrid* alien = TGrid::Connect("alien://");
```

This triggers the loading of the AliEn plug-in module and returns an instance of TAlien

TFile is the base class for all file protocols. A **TAlienFile** is created in the same manner by the static factory function 'Open' of **TFile**:

```
TFile* alienfile = TFile::Open("alien://....");
```

The following sections highlight the most important aspects of the interface and are not meant to be exhaustive. For more details see the ROOT documentation included in the source code, which is located in the 'root/alien' directory of the source.

Note: Examples in this chapter are self contained, which means that they always start with a `Connect` statement.

5.10.3.1 TGrid::Connect - Authentication and Session Creation

The first thing to do (see the quick test in 5.10.2), in order to get access to AliEn data and job management functionalities, is to authenticate with an **API** service, or to use an already existing session token.

As described in 5.5.1, we can store a session token within the ROOT application, or we access a session token that was established outside the application to be used by **aliensh**.

```
//--- Load desired plug-in and setup conection to GRID
static TGrid *Connect(const char *grid, const char *uid = 0,
const char *pw = 0, const char *options = 0);
Syntax 1: TGrid::Connect
```

Consider these example statements to initiate connections:

Connect creating a memory token to a default **API** service :

```
TGrid::Connect("alien://");
```

Connect creating a memory token to a user-specified **API** service:

```
TGrid::Connect("alien://myhost.cern.ch:9000");
```

Connect creating a memory token with a certain role:

```
TGrid::Connect("alien://myhost.cern.ch:9000","aliprod");
```

Connect using an existing file token (created by alien-token-init):

```
TGrid::Connect("alien://",0,0,"t");
```

Note: the first method mentioned should apply to 99% of all use cases. If you use programs that fork, you should always use the file based token mechanism or call in every forked process the `Connect` method again. If you are using threads, you must lock the `Command` statements later on with a mutex lock.

Return Values :

- (TGrid*) 0 Connect failed
- (TGrid*) != 0 Connection established

ROOT sets the global variable `gGrid` automatically with the result of **TGrid**`::Connect`. If you deal only with one GRID connection, you can just use that one to call any of the **TAlien** methods e.g. **`gGrid->Ls()`** , but not `TAlien::Ls()` !

TGrid`::Connect("alien://")` is equivalent to the call new `TAlien(...)`, which bypasses the plug-in mechanism of ROOT!

5.10.3.2TAlien::Command - arbitrary command execution

You can execute any **aliensh** command with the `Command` method (except the `cp` command).

```
TGridResult *Command(const char *command, bool interactive = kFALSE,
UInt_t stream = kOUTPUT);
Syntax 2: TAlien::Command
```

As you have already seen, the **API** protocol returns four streams. The stream to be stored in a **TGridResult** is, by default, the result structure of each command (**TAlien**`::kOUTPUT`). Another stream can be selected using the `stream` parameter:

STDOUT	stream = TAlien::kSTDOUT
STDERR	stream = TAlien::kSTDERR
result structure	stream = TAlien::kOUTPUT
misc. hash	stream = TAlien::kENVIR

Note: you need to add `#include <TAlien.h>` to your ROOT code, in order to have the stream definitions available!

If you set `interactive=kTRUE`, the STDOUT+STDERR stream of the command appear on the terminal, otherwise it is silent.

The result structure and examples for using the `Command` method are explained in the next section.

Continue reading until section and then try the example given.

5.10.3.3TAlienResult - the result structure of TAlien::Command

TAlienResult is the plug-in implementation of a **TGridResult** returned by **TAlien**`::Command`. A **TAlienResult** is based on a **TList** which contains a **TMap** as list entries. It roughly is a list of key-value pairs. If you want to find out names of returned key names, just use the **TAlienResult**`::Print("all")` method or use the 'gbbox `-d'` command.

5.10.3.4TAlien::Ls - Example of Directory Listing in the File Catalogue

Consider this example, which executes the '`ls -la`' function, then dumps the complete result and finally loops over the result and prints all file names in the `/alice` directory:

```
TGrid::Connect("alien://");
TGridResult* result =gGrid->Command("ls -la/alice",0,TAlien::kOUTPUT);
result->Print("all");
while (result->GetKey(i,"name"))
      printf("Listing file name: %s\n",result->GetKey(i++,"name") ;
```

The keys defined for an 'ls' operation are currently:

`ls-la`	name, md5, size, group, path, permissions, user,date
`ls -m`	md5,path (here path is the full path name)
`ls -b`	guid,path (here path is the full path name)
`ls`	name,path

Since this is a very common use case, there are convenience functions defined to simplify the syntax of the listing example:

```
TGrid::Connect("alien://");
TGridResult* result =gGrid->Ls("/alice",);
while (result->GetFileName(i))
   printf("Listing file name: %s\n",result->GetFileName(i++) ;
```

5.10.3.5TAlien::Cd + TAlien::Pwd - Example how to navigate the CWD

The CWD allows you to reference files without absolute path names. The CWD is by default (if you don't use a file token, where the CWD is shared between all sessions using the same token and stored on disk) your home directory after connecting.

To see the current working directory use the Pwd command:

```
TGrid::Connect("alien://");
printf("Working Directory is %s\n",gGrid->Pwd());
```

It returns a `const char*` to your current workding directory name.

To navige the CWD use the Cd command:

```
TGrid::Connect("alien://");
Bool result = gGrid->Cd("/alice");
```

Return Values :

- kTRUE Directory created
- kFALSE Directory creation failed

5.10.3.6TAlien::Mkdir - Example how to create a directory

```
Bool_t TAlien::Mkdir(const char* ldn, Option_t* options, Bool_t
verbose)
```

ldn	specifies the logical directory name you want to create e.g. "/alice/cern.ch/mydirectory"
options	flags for the command
verbose=kTRUE	controls verbosity

```
TGrid::Connect("alien://");
Bool result = gGrid->Mkdir("mydirectory");
```

Return Values :

- kTRUE Directory created
- kFALSE Directory creation failed

5.10.3.7TAlien::Rmdir - Example how to remove a directory

```
Bool_t TAlien::Rmdir(const char* ldn, Option_t* options, Bool_t
verbose)
```

ldn	specifies the logical directory name you want to remove e.g. "/alice/cern.ch/mydirectory"
options	are flags for the command
verbose=kTRUE	controls on verbosity of the command

```
TGrid::Connect("alien://");
Bool result = gGrid->Rmdir("mydirectory");
```

Return Values :

- kTRUE Directory removed
- kFALSE Directory deletion failed

5.10.3.8TAlien::Rm - Example how to remove a file entry

```
Bool_t TAlien::Rm(const char* lfn, Option_t* options, Bool_t verbose)
```

lfn	specifies the logical file name you want to remove e.g. "/alice/cern.ch/myfile"
options	are flags for the command (see 5.9.8).
verbose=kTRUE	switches on verbosity of the command

```
TGrid::Connect("alien://");
Bool result = gGrid->Rm("myfile");
```

Return Values :

- kTRUE File entry removed
- kFALSE File entry deletion failed

Note: as said previously - this function removes only file catalogue entries, no physical files

5.10.3.9TAlien::Query - Querying files in the Catalogue

The query function is a very convenient way to produce a list of files, which can be converted later into a ROOT **TChain** (e.g. to run a selector on your local machine or on

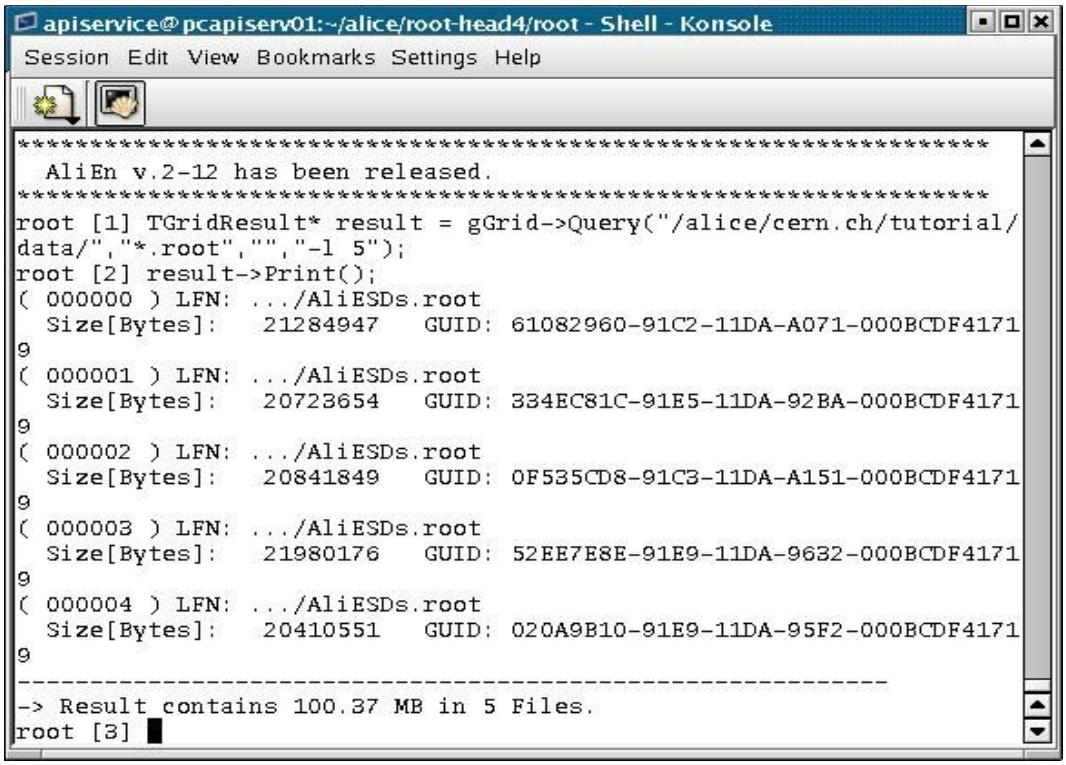
a PROOF cluster).

```
virtual TGridResult *Query(const char *path, const char *pattern,
                           const char *conditions = "", const char
*options = "");
```

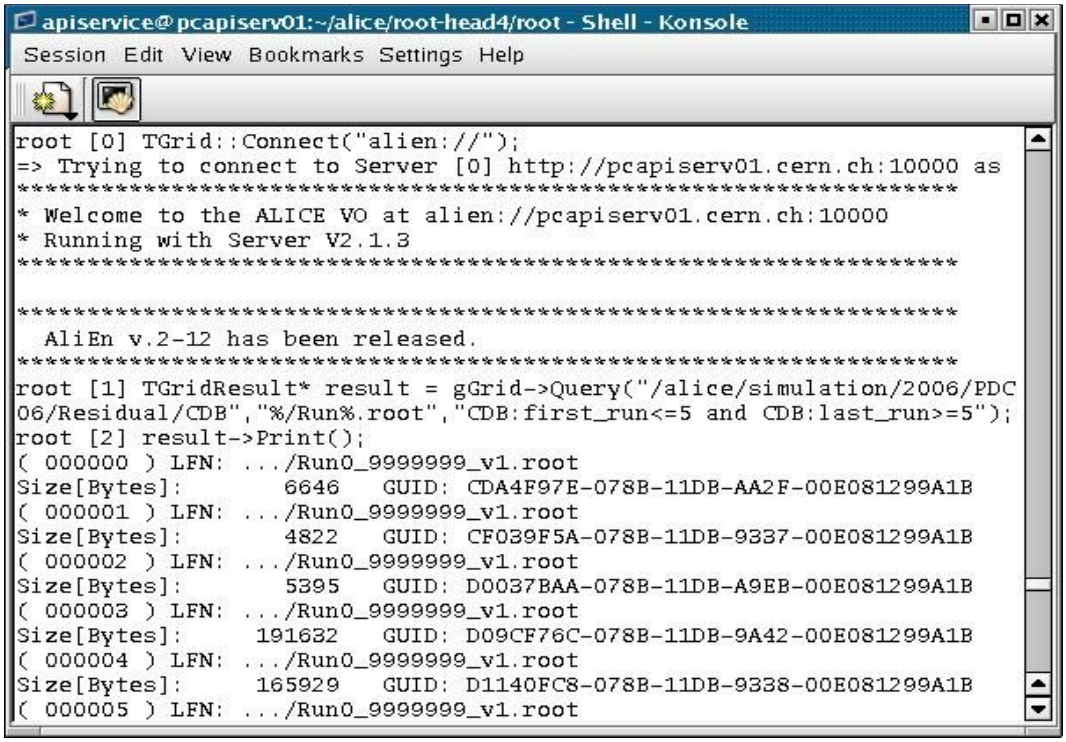
The syntax is straightforward:

path	specifies the node (directory) where to start searching in the directory tree	
pattern	specifies a pattern to be matched in the full filename e.g.	
	*.root	matches all files with 'root' suffix
	*	matches all files
	galice.root	matches exact file names
conditions	conditions on metadata for the queried files	
options	options to the query command (see find in the aliensh section)	

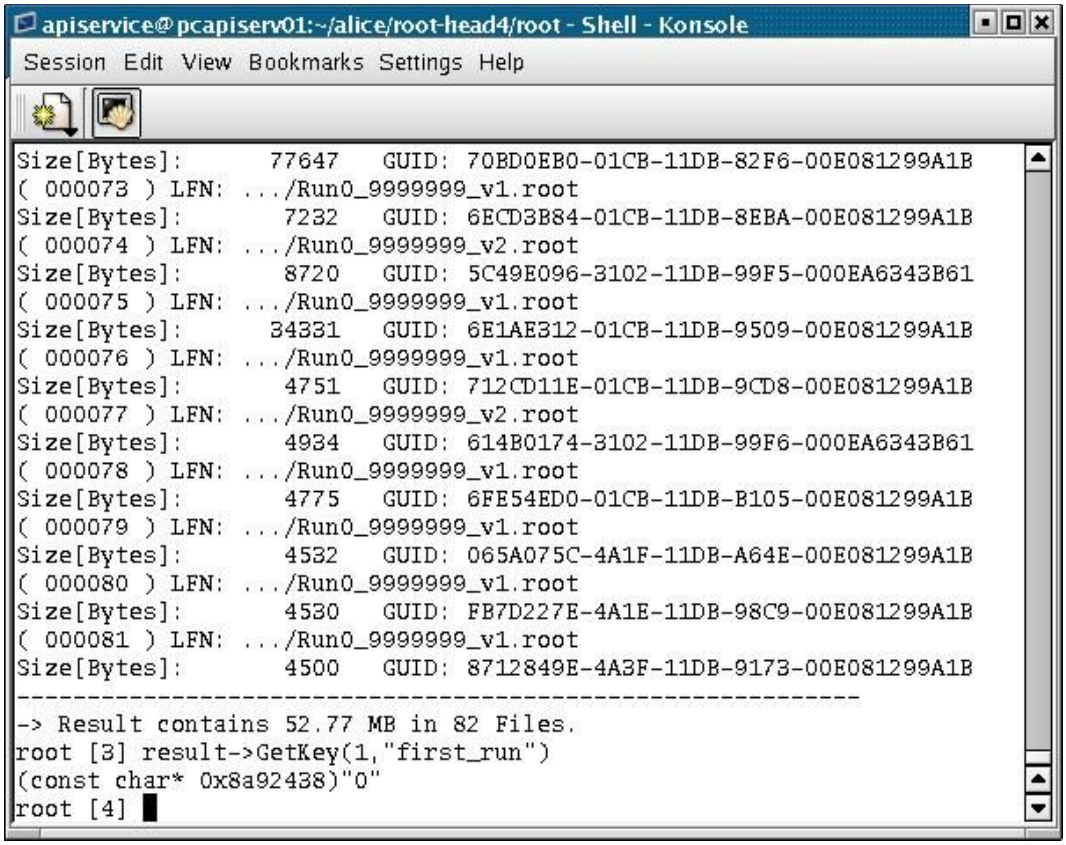
This is a simple example querying all “*.root” files under a certain directory tree. The option “-l 5” sets a limit to return max. five files.



A more advanced example using metadata is shown here:



The query returns also all metadata fields in the TGridResult object. You can use **TGridResult**::GetKey to retrieve certain metadata values:



All returned metadata values are in text (char*) format, and you have to apply the appropriate conversion function.

5.10.3.10File Access using TFile - TAlienFile

ROOT has a plug-in mechanism for various file access protocols. These plug-ins are called via the static **TFile**::Open function. The protocol specified in the URL refers to the appropriate plug-in. **TAlienFile** is the implemented plug-in class for file registered in AliEn. Transfers are done using the **TXNetFile** class (xrootd) internally. To reference a logical file in AliEn, use the 'alien://' protocol or add '/alien' as prefix to the logical file name space:

```
TFile::Open
("alien:///alice/cern.ch/demo/data/0115.00108/miniesd.root")
```

opens an AliEn file in READ mode.

```
TFile::Open("/alien/alice/cern.ch/demo/data/0115.00108/miniesd.root");
```

is equivalent to the first statement.

```
TFile::Open
("alien:///alice/cern.ch/user/t/test/myfile.root@ALICE::CERN::se01","R
ECREATE")
```

opens an AliEn file in WRITE mode using the file versioning system at the storage element **'ALICE::CERN::se01'**.

5.10.3.11File Copy operations in ROOT

The class **TFileMerger** implements besides File Merging functionality a copy function to copy from 'arbitrary' source to destionation URLs. Instead of using **aliensh** commands, you can copy a file within ROOT code following this example:

```
TFileMerger m;
m.Cp
("alien:///alice/cern.ch/demo/data/0115.00108/miniesd.root","file:/tmp
/miniesd.root");
```

This works also when copying local to local files or AliEn to AliEn files.

5.11Appendix JDL Syntax

Every JDL tag follows this syntax for single values:

```
<tag-name> = "<value>";
```

or

```
<tag-name> = {"<value>"};
```

for a value list:

```
<tag-name> = { "<val1>" , "<val2>", "<val3>" [ ... "<valN>"};
```

5.11.1JDL Tags

Executable

This is the only compulsory field in the jdl. It states the name of the lfn that will be executed. The file must be located either in /bin or /<VO>/bin or /<HOME>/bin

Arguments

These will be passed to the executable

Packages

This constrains the execution of the job to be done at a site where the package is installed. You can also request a specific version of a package. For example *Packages="AliRoot"* will require the current version of AliRoot, or *Packages="AliRoot::3.07.02"* will require Version 3.07.02.

InputFile

A list of files that will be transported to the node where the job will be executed. Lfn's have to be specified like "LF:/alice/cern.ch/mymacro.C".

InputData

InputData is similar to InputFile, but the physical location of InputData adds automatically a requirement to the location of execution to the JDL.

It is required to execute the job in a site close to the files specified here. You can specify pattern like *"LF:/alice/simulation/2003-02/V3.09.06/00143/*/tpc.tracks.root"*,and then all the LFN that satisfy this pattern will be included.

If you don't want the files to be staged to the sandbox (typical for Analysis) as it is done also for input files, you can specify "LF:/alice/...../file.root,nodownload".

We recommend to use this tag only for a small number of files (<100) – otherwise the JDL becomes very large and slows down processing. If you have to specify many files, use InputDataCollection instead.

InputDataCollection

An input data collection is used to specify long lists of input data files and allows to group corresponding files together. Use the find command to create an input data collection.

The input data collection file contains the InputData in XML list format that can be produced using the find command e.g.

```
find -x example1 /alice/cern.ch/data/ *.root > /tmp/example1.xml
```

This file is afterwards copied into AliEn using the cp command. You should use this mechanism, if you have many input files the submission is much fasterit is better for the job optimizer services you don't need to specify the InputData field

InputDataList

This is the name of the file were the Job Agent saves the InputData list. The format of this file is specified in `InputDataListFormat.`

InputDataListFormat

This is the list format of the InputData list. Possible formats are:

- "xml-single"
- "xml-group"

'xml-single' implies, that every file is equivalent to one event. If you specify 'xml-group', a new event starts every time the first base filename appears again, e.g.

```
"LF: ..../01/galice.root",            ←1st event
"LF: ..../01/Kinematics.root",
"LF: ..../02/galice.root",            ← 2nd event
"LF: ..../02/Kinematics.root",
......
```

OutputFile

The files that will be registered in the catalogue after the job has finished. You can specify the storage element by adding “@<SE-Name” to the file name.

Example:

```
OutputFile="histogram.root@ALICE::CERN::se01"
```

OutputArchive

Here you can define, which output files are archived in ZIP archives. Per default, AliEn puts all OutputFiles together in ONE archive. Example:

This writes two archives: one with all the log files + STDOUT + STDERR stored in the SE ALICE::CERN::se01, another archive containing ROOT files, which are stored in SE ALICE::CERN::Castor2.

```
OutputArchive =
           {
              "root_archive.zip:*.root@Alice::CERN::Castor2",
              "log_archive:*.log,stdout,stderr@Alice::CERN::se01"
           };
```

Validationcommand:

This specifies the script to be executed as validation. If the return value of that script is ! =0, the job will terminate with status ERROR_V, otherwise SAVED→DONE.

Email

If you want to receive an email when the job has finished, you can specify your email address here. This does not yet work for master jobs.

TTL

Here you specify the maximum run-time for your job. The system then selectes a worker node which provides the requested run time for you job.

Split:

If you want to split your job in several sub jobs, you can define a method to split the job according to the input data (collection) or some production variables. Valid are:

file	There will be one sub job per file in the InputData section.
directory	All the files of one directory will be analyzed in the same sub-job.

se	This should be used for most analysis. The jobs are split according to the list of storage elements where data are located. Job <1> reads all data at <SE1>, job <2> all data at <SE2>. You can, however, force to split on the second level the job <1> ,<2> … into several jobs using the two tags SplitMaxInputFileNumber and SplitMaxInputFileSize.
event	All files with the same name of the last subdirectory will be analyzed in the same sub job.
userdefined	Check the field SplitDefinitions
Production (<#start>-<#end>))	This kind of split does not require any InputData. It will submit the same JDL several times (from #start to #end). You can reference this counter in SplitArguments using "#alien_counter#"

SplitArguments

You can define the argument field for each sub-job. If you want e.g. to give the sub jobs counter produced by the Split="production:1-10" tag, you can write e.g. something like

```
SplitArguments = "simrun.C --run 1 --event #alien_counter#";
```

If you define more than one value, each sub-job will be submitted as many times as there are items in this array, and the sub-jobs will have the element in the array as arguments.

SplitMaxInputFileNumber

Defines the maximum number of files that are in each of the sub-jobs. For instance, if you split per SE putting 'SplitMaxInputFileNumber=10', you can make sure that no sub-job will have more than ten input data files.

SplitMaxInputFileSize

Similar to the previous, but puts a limit on the size of the file. The size has to be given in bytes.

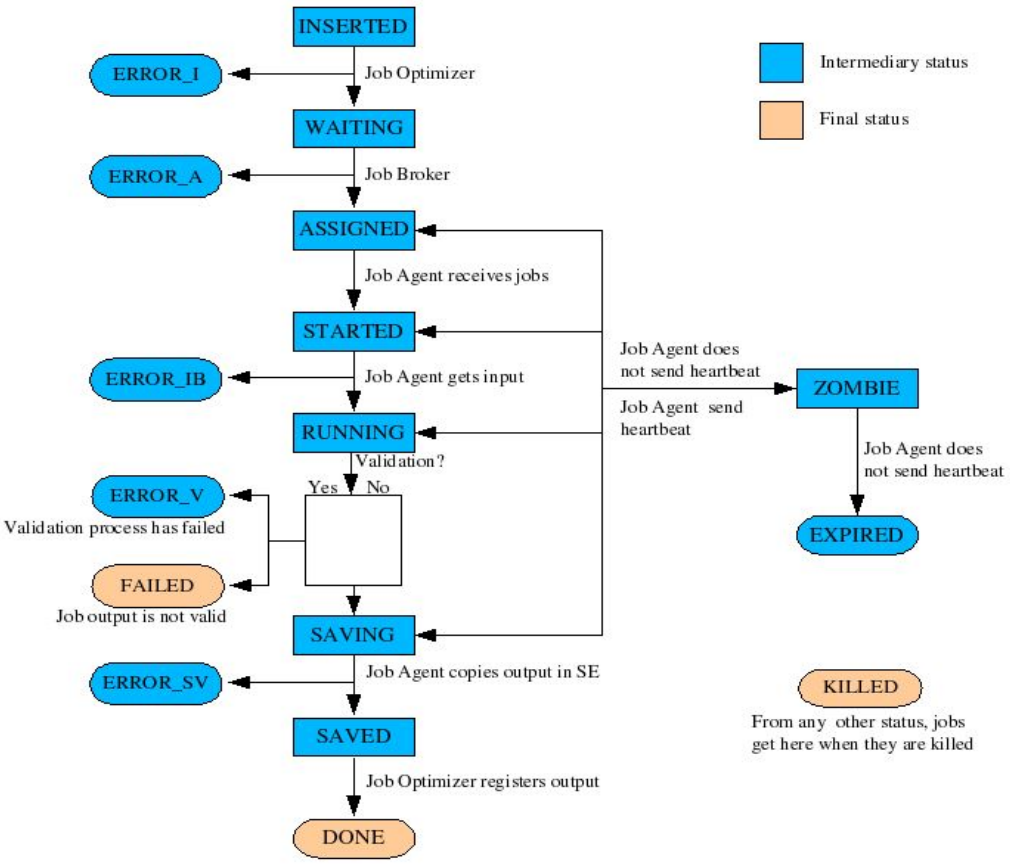
SplitDefinitions

This is a list of JDLs. If defined by the user, AliEn will take those jdls as the sub-jobs, and all of them will behave as if they were sub-jobs of the original job (for instance, if the original jobs gets killed, all of them will get killed, and once all of the sub-jobs finish, their output will be copied to the master job).

5.12 Appendix Job Status

The following flow chart shows the job status transitions after you have successfully submitted a job. It will help you to understand the meaning of the various error conditions.

5.12.1Status Flow Diagram



5.12.2Non-Error Status Explanation

In the following we describe the non-error status. The abbreviation in brackets is what the ps command shows.

INSERTING (I)

The job is waiting to be processed by the Job Optimizer. If this is a split job, the Optimizer will produce many sub jobs out of your master job. If this is a plain job, the Optimizer will prepare the input sandbox for this job.

WAITING (W)

The job is waiting to be picked up by any Job Agent, that can fulfil the job requirements.

ASSIGNED (A)

A Job Agent has matched your job and is about to pick it up.

STARTED (ST)

A Job Agent is now preparing the input sandbox downloading the specified input files.

RUNNING (R)

Your executable has finally been started and is running.

SAVING (SV)

Your executable has successfully terminated, and the agent saves your output to the specified storage elements.

SAVED (SVD)

The agent has successfully saved all the output files which are not yet visible in the catalogue.

DONE (D)

A central Job Optimizer has registered the output of your job in the catalog.

5.12.3Error Status Explanation

ERROR_I (EI) - ERROR_A (EA)

These errors are normally not based on a 'bad' user job, but arise due to service failures.

ERROR_IB (EIB)

This is a common error which occurs during the download phase of the required input files into the sandbox. Usually its cause is, that a certain input file does not exist in the assumed storage element, or the storage element is not reachable from the job worker node.

ERROR_V (EV)

The validation procedure failed, i.e. your validation script (which you have specified in the JDL) exited with a value !=0.

ERROR_SV (ESV)

At least one output file could not be saved as requested in the JDL. Probably, one of the storage elements required was not available.

ZOMBIE/EXPIRED (Z/EXP)

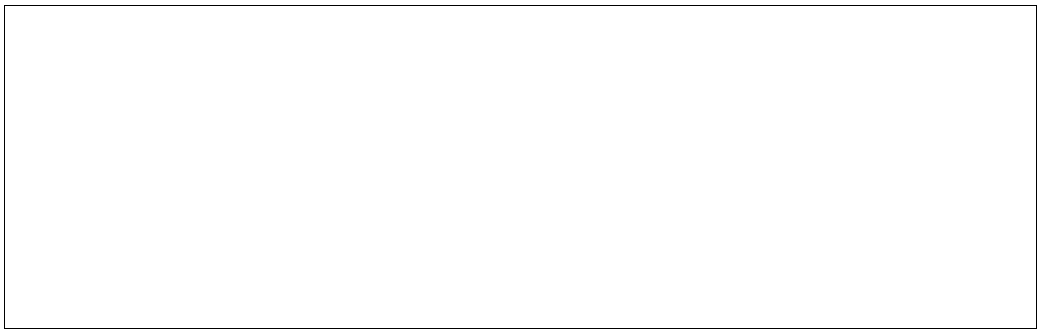
Your job got lost on a worker node. This can happen due to a node failure or a network interruption. The only solution is to re-submit the job.

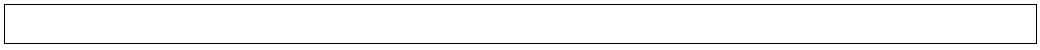
6 197197199

199

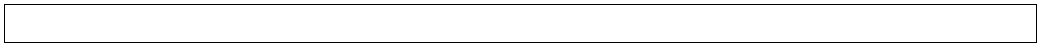
-
-
-
-
-
-

197





-
-
-
-



-

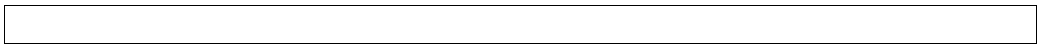
-
-

-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

6.1

6.2

6.2.1



6.2.2

6.3

199

7 Distributed analysis

7.1 Abstract

In order to perform physics analysis in ALICE, a physicist has to use the GRID infrastructure, since the data will be distributed to many sites. The machinery, though complex due to the nature of the GRID, is totally transparent to the user who is shielded by a friendly user interface. In order to provide some guidelines to successfully utilize the functionalities provided, it was decided that an up-to-date manual was needed. Here, we try to explain the analysis framework as of today taking into account all recent developments.

After a short introduction, the general flow of a GRID based analysis will be outlined. The different steps to prepare one's analysis and execute it on the GRID using the newly developed analysis framework will be covered in the subsequent sections.

7.2 Introduction

Based on the official ALICE documents [197, 197], the computing model of the experiment can be described as follows:

- Tier 0 provides permanent storage of the raw data, distributes them to Tier 1 and performs the calibration and alignment task as well as the first reconstruction pass. The calibration procedure will also be addressed by PROOF clusters such as the CERN Analysis Facility (CAF) [199]
- Tier 1s outside CERN collectively provide permanent storage of a copy of the raw data. All Tier 1s perform the subsequent reconstruction passes and the scheduled analysis tasks.
- Tier 2s generate and reconstruct the simulated Monte Carlo data and perform the chaotic analysis submitted by the physicists.

The experience of past experiments shows that the typical data analysis (chaotic analysis) will consume a large fraction of the total amount of resources. The time needed to analyze and reconstruct events depends mainly on the analysis and reconstruction algorithm. In particular, the GRID user data analysis has been developed and tested with two approaches: the asynchronous (batch approach) and the synchronous (interactive) analysis.

Before going into detail on the different analysis tasks, we would like to address the general steps a user needs to take before submitting an analysis job:

- Code validation: In order to validate the code, a user should copy a few AliESDs.root or AliAODs.root files locally and try to analyze them by following the instructions listed in section .

- Interactive analysis: After the user is satisfied from both the code sanity and the corresponding results, the next step is to increase the statistics by submitting an interactive job that will analyze ESDs/AODs stored on the GRID. This task is done in such a way to simulate the behaviour of a GRID worker node. If this step is successful then we have a high probability that our batch job will be executed properly. Detailed instructions on how to perform this task are listed in section .
- Finally, if the user is satisfied with the results from the previous step, a batch job can be launched that will take advantage of the whole GRID infrastructure in order to analyze files stored in different storage elements. This step is covered in detail in section .

It should be pointed out that what we describe in this note involves the usage of the whole metadata machinery of the ALICE experiment: that is both the file/run level metadata [59] as well as the Event Tag System [60]. The latter is used extensively, because apart from the fact that it provides an event filtering mechanism to the users and thus reducing the overall analysis time significantly, it also provides a transparent way to retrieve the desired input data collection in the proper format (= a chain of ESD/AOD files) which can be directly analyzed. On the other hand, if the Event Tag System is not used, then, apart from the fact that the user cannot utilize the event filtering, he/she also has to create the input data collection (= a chain of ESD/AOD files) manually.

7.3 Flow of the analysis procedure

shows a schematic view of the flow of the analysis procedure. The first thing a typical user needs to do in order to analyze events stored on the GRID, is to interact with the file catalogue to define the collection of files that he/she needs. This action implies the usage of the metadata fields assigned at the level of a run. Detailed description about the structure of the file catalogue as well as the metadata fields on this level can be found in [, 61]. As an example, we have a user who wants to create a collection of tag-files that fulfil the following criteria:

- The production year should be 2008.
- The period of the LHC machine should be the first of 2008 (LHC08a).
- The data should come from the third reconstruction pass.
- The collision system should be p+p.
- The start and stop time of the run should be beyond the 19th of March 2008 and no later than 10:20 of the 20th of March 2008 respectively.

Then, what needs to be written in the AliEn shell (as a one line command) is:

```
[aliensh] find -x pp /alice/data/2008/LHC08a/*/reco/Pass3/*
*Merged*tag.root
  Run:collision_system=''pp'' and Run:stop<''2008-03-20 10:20:33''
  and Run:start>''2008-03-19'' > pp.xml
```

The previous lines make use of the `find` command of the alien shell [62]. The first

argument of this command is the name of the collection which will be written inside the file (header of the xml collection). If the `-x pp` option is not used, we will not get back all the information about the file but instead we will just retrieve the list of logical file names (lfn). Then, the path of the file catalogue where the files we want to analyze are stored is given, followed by the name of the file. In this example, the path implies that the user requests a collection of real data (`/alice/data`) coming from the first period of the LHC machine in year 2008 (`/2008/LHC08a`), containing all the run numbers of the third pass of the reconstructed sample (`/*/reco/Pass3`). The last argument is the list of metadata fields: a variety of such fields can be combined using logical statements. The reader should notice that the output of this command is redirected to an xml file, which consists of all the necessary information (the file's unique identifier - guid, the logical file name etc) about the files that fulfil the imposed selection criteria. This xml collection, which is stored in the local working directory, plays a very important role in the overall distributed analysis framework, as we will see in the following sections. From the above example it is obvious that wild cards can be used.

Going back to the description of and following the flow of the arrows, we assume that the user creates a tag xml collection. Then, in parallel inside a macro he/she imposes some selection criteria at the event level. Having those two components as an input, the Event Tag System is queried and from this action, the system can either provide directly the input data collection, which is a chain of ESD/AOD files along with the associated event list (which describes the events that fulfil the imposed selection criteria), or a new ESD/AOD xml (different to the tag xml collection discussed earlier) collection (mainly used for batch analysis – see section for details). Even in the latter case, we end up having the chain along with the associated event list. This chain can then be processed by an analysis manager [199] either locally, in AliEn [197], or using PROOF [198].

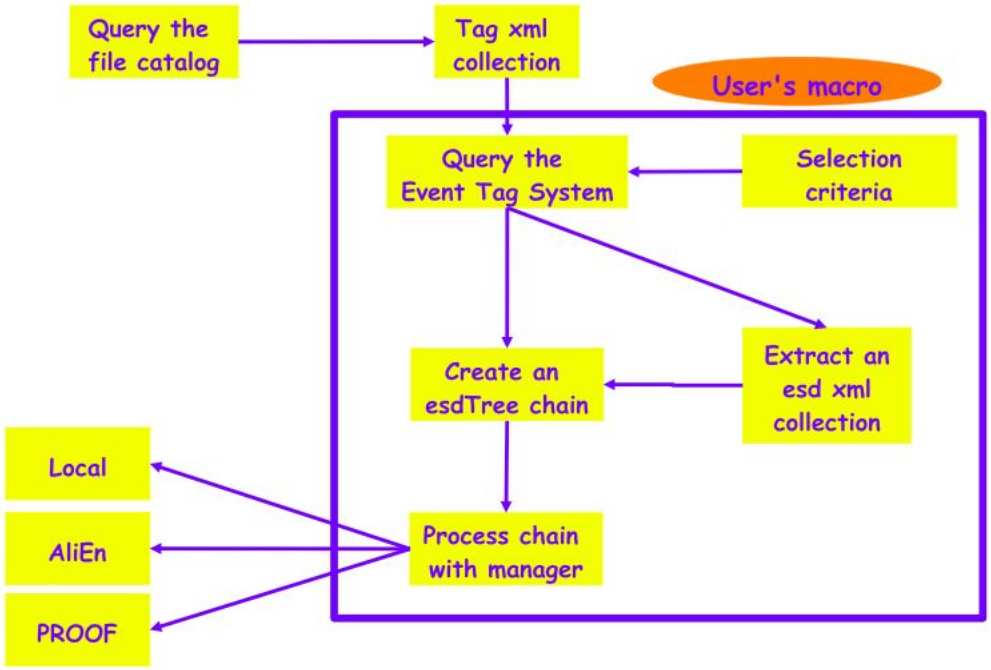


Figure 17: The flow of the analysis procedure: Having as an input the tag xml collection that we create by querying the file catalogue and the selection criteria, we interact with the Event Tag System and we either get a chain along with the associated event list (events that fulfil the imposed selection criteria) or we create an ESD xml collection (for batch sessions) from which we create the ESD chain. This chain is processed with an analysis manager locally, in AliEn or even in PROOF.

We would also like to point out that we have two options on how to use the framework:

- We can work with AliRoot and try all the examples that will be described in the next sections by loading all the corresponding libraries.
- We can try to be as flexible as possible, by running ROOT along with the corresponding AliRoot libraries (e.g. in the case of the analysis of AliESDs.root or/and AliAODs.root we need the libSTEERBase.so, libESD.so, libAOD.so along with the libANALYSIS.so - the latter is needed for the new analysis framework which will be described in the next section). These libraries are created from the compilation of the relevant AliRoot code which can be included in the so-called `par file`. This par file is nothing more than a tarball containing the .h and .cxx aliroot code along with the Makefile and Makefile.arch (needed to compile the AliRoot code in different platforms).

The user has these two possibilities, although for the following examples, we will concentrate on the case where we use the `par files`. The lines listed below show how we can setup, compile and load the libESD.so from the *ESD.par*.

```
const char* pararchivename = "ESD";
  // Setup PAR File
  if (pararchivename) {
```

```
    char processline[1024];
    sprintf(processline,".! tar xvzf %s.par",pararchivename);
    gROOT->ProcessLine(processline);
    const char* ocwd = gSystem->WorkingDirectory();
    gSystem->ChangeDirectory(pararchivename);

    // check for BUILD.sh and execute
    if (!gSystem->AccessPathName("PROOF-INF/BUILD.sh")) {
      printf("*******************************\n");
      printf("*** Building PAR archive    ***\n");
      printf("*******************************\n");

      if (gSystem->Exec("PROOF-INF/BUILD.sh")) {
        Error("runProcess","Cannot Build the PAR Archive! - Abort!");
        return -1;
      }
    }
    // check for SETUP.C and execute
    if (!gSystem->AccessPathName("PROOF-INF/SETUP.C")) {
      printf("*******************************\n");
      printf("*** Setup PAR archive       ***\n");
      printf("*******************************\n");
      gROOT->Macro("PROOF-INF/SETUP.C");
    }

    gSystem->ChangeDirectory("../");
  }
  gSystem->Load("libVMC.so");
  gSystem->Load("libESD.so");
```

7.4 Analysis framework

Recently, a new attempt has started that led to the development of a new analysis framework [199]. By the time this note was written, the framework has been validated in processes that concern this document (GRID analysis). We will review the basic classes and functionalities of this system in this paragraph (for further details the reader should look in []).

The basic classes that constitute this development are the following (we will give a full example of an overall implementation later):

- **AliAnalysisDataContainer**: The class that allows the user to define the basic input and output containers.
- **AliAnalysisTask**: This is the class the lines of which should be implemented by the user. In the source file we can write the analysis code.
- **AliAnalysisManager**: Inside such a manager the user defines the analysis containers, the relevant tasks as well as the connection between them.

A practical example of a simple usage of this framework is given below where we extract a p_t spectrum of all charged tracks (histogram) from an ESD chain (the

examples can be found under the `AnalysisMacros/Local/` directory of the PWG2 module of AliRoot).

AliAnalysisTaskPt.h

```
#ifndef AliAnalysisTaskPt_cxx
#define AliAnalysisTaskPt_cxx

// example of an analysis task creating a p_t spectrum
// Authors: Panos Cristakoglou, Jan Fiete Grosse-Oetringhaus,
Christian Klein-Boesing

class TH1F;
class AliESDEvent;

#include "AliAnalysisTask.h"

class AliAnalysisTaskPt : public AliAnalysisTask {
 public:
  AliAnalysisTaskPt(const char *name = "AliAnalysisTaskPt");
  virtual ~AliAnalysisTaskPt() {}

  virtual void   ConnectInputData(Option_t *);
  virtual void   CreateOutputObjects();
  virtual void   Exec(Option_t *option);
  virtual void   Terminate(Option_t *);

 private:
  AliESDEvent *fESD;    //ESD object
  TH1F        *fHistPt; //Pt spectrum

  AliAnalysisTaskPt(const AliAnalysisTaskPt&); // not implemented
  AliAnalysisTaskPt& operator=(const AliAnalysisTaskPt&); // not
implemented
  ClassDef(AliAnalysisTaskPt, 1); // example of analysis
};

#endif
```

AliAnalysisTaskPt.cxx

```
#include "TChain.h"
#include "TTree.h"
#include "TH1F.h"
#include "TCanvas.h"

#include "AliAnalysisTask.h"
#include "AliAnalysisManager.h"

#include "AliESDEvent.h"
#include "AliESDInputHandler.h"

#include "AliAnalysisTaskPt.h"
```

```
// example of an analysis task creating a p_t spectrum
// Authors: Panos Cristakoglou, Jan Fiete Grosse-Oetringhaus, Christian
Klein-Boesing

ClassImp(AliAnalysisTaskPt)

//_____________________________________________________________________________
___
AliAnalysisTaskPt::AliAnalysisTaskPt(const char *name)
  : AliAnalysisTask(name, ""), fESD(0), fHistPt(0) {
  // Constructor

  // Define input and output slots here
  // Input slot #0 works with a TChain
  DefineInput(0, TChain::Class());
  // Output slot #0 writes into a TH1 container
  DefineOutput(0, TH1F::Class());
}

//_____________________________________________________________________________
___
void AliAnalysisTaskPt::ConnectInputData(Option_t *) {
  // Connect ESD or AOD here
  // Called once

  TTree* tree = dynamic_cast<TTree*> (GetInputData(0));
  if (!tree) {
    Printf("ERROR: Could not read chain from input slot 0");
  } else {
    // Disable all branches and enable only the needed ones
    // The next two lines are different when data produced as
AliESDEvent is read
    tree->SetBranchStatus("*", kFALSE);
    tree->SetBranchStatus("fTracks.*", kTRUE);

    AliESDInputHandler *esdH = dynamic_cast<AliESDInputHandler*>
(AliAnalysisManager::GetAnalysisManager()->GetInputEventHandler());

    if (!esdH) {
      Printf("ERROR: Could not get ESDInputHandler");
    } else
      fESD = esdH->GetEvent();
  }
}

//_____________________________________________________________________________
___
void AliAnalysisTaskPt::CreateOutputObjects() {
  // Create histograms
  // Called once

  fHistPt = new TH1F("fHistPt", "P_{T} distribution", 15, 0.1, 3.1);
```

```
  fHistPt->GetXaxis()->SetTitle("P_{T} (GeV/c)");
  fHistPt->GetYaxis()->SetTitle("dN/dP_{T} (c/GeV)");
  fHistPt->SetMarkerStyle(kFullCircle);
}

//_____________________________________________________________________________
___
void AliAnalysisTaskPt::Exec(Option_t *) {
  // Main loop
  // Called for each event

  if (!fESD) {
    Printf("ERROR: fESD not available");
    return;
  }

  Printf("There are %d tracks in this event", fESD->GetNumberOfTracks
());

  // Track loop to fill a pT spectrum
  for (Int_t iTracks = 0; iTracks < fESD->GetNumberOfTracks();
iTracks++) {
    AliESDtrack* track = fESD->GetTrack(iTracks);
    if (!track) {
      Printf("ERROR: Could not receive track %d", iTracks);
      continue;
    }

    fHistPt->Fill(track->Pt());
  } //track loop

  // Post output data.
  PostData(0, fHistPt);
}

//_____________________________________________________________________________
___
void AliAnalysisTaskPt::Terminate(Option_t *) {
  // Draw result to the screen
  // Called once at the end of the query

  fHistPt = dynamic_cast<TH1F*> (GetOutputData(0));
  if (!fHistPt) {
    Printf("ERROR: fHistPt not available");
    return;
  }

  TCanvas *c1 = new TCanvas("AliAnalysisTaskPt","Pt",10,10,510,510);
  c1->cd(1)->SetLogy();
  fHistPt->DrawCopy("E");
}
```

The **AliAnalysisTaskPt** is a sample task that inherits from the **AliAnalysisTask** class. The main functions that need to be implemented are the following:

- Basic constructor: Inside the constructor we need to define the type of the input (if any input is to be used) as well as of the output of the task (in our example the input is of the form of a ROOT's `TChain` while the output is a histogram – `TH1F`).
- `ConnectInputData` Inside this function we need to initialize the input objects (get the TTree and the AliESDEvent object).
- `CreateOutputObjects` Create the output objects that will be written in the file (in our example we create the output histogram).
- `Exec`: The place where the analysis code should be implemented.
- `Terminate`: The function inside of which we can draw histograms (as in our example) or perform any kind of actions (like merging of output).

Macro that creates an **AliAnalysisManager**

```
  //_____________________________________________//
  // Make the analysis manager
  AliAnalysisManager *mgr = new AliAnalysisManager("TestManager");
  AliVEventHandler* esdH = new AliESDInputHandler;
  mgr->SetInputEventHandler(esdH);
  //_____________________________________________//
  // 1st Pt task
  AliAnalysisTaskPt *task1 = new AliAnalysisTaskPt("TaskPt");
  mgr->AddTask(task1);
  // Create containers for input/output
  AliAnalysisDataContainer *cinput1 = mgr->CreateContainer
("cchain1",TChain::Class(),AliAnalysisManager::kInputContainer);
  AliAnalysisDataContainer *coutput1 = mgr->CreateContainer("chist1",
TH1::Class(),AliAnalysisManager::kOutputContainer,"Pt.ESD.root");

  //_____________________________________________//
  mgr->ConnectInput(task1,0,cinput1);
  mgr->ConnectOutput(task1,0,coutput1);
  if (!mgr->InitAnalysis()) return;
  mgr->PrintStatus();
  mgr->StartAnalysis("local",chain);
```

In the previous lines, we first created a new analysis manager:

```
AliAnalysisManager *mgr = new AliAnalysisManager(“TestManager”);
```

The next step is to set the input event handler which will allow us to retrieve inside our task a valid esd/aod object (in our example we get an AliESDEvent):

```
AliVEventHandler* esdH = new AliESDInputHandler();
mgr->SetInputEventHandler(esdH);
```

We then created a new **AliAnalysisTaskPt** object giving it a name and we assigned it to the manager:

```
AliAnalysisTaskPt *task1 = new AliAnalysisTaskPt("TaskPt");
mgr->AddTask(task1);
```

Then, the input and output containers were created and their types defined:

```
AliAnalysisDataContainer *cinput1 = mgr->CreateContainer("cchain1",
TChain::Class(),AliAnalysisManager::kInputContainer);
AliAnalysisDataContainer *coutput1 = mgr->CreateContainer("chist1",
TH1::Class(),AliAnalysisManager::kOutputContainer,"Pt.ESD.root");
```

The next step was to link the task with the corresponding input and output container while setting the created chain of files (in the following section we will review how we can create this chain) to our input:

```
mgr->ConnectInput(task1,0,cinput1);
mgr->ConnectOutput(task1,0,coutput1);
```

Finally, we process the chain with the manager

```
mgr->StartAnalysis("local",chain);
```

In order to give the user an idea of how the framework could look like in a complicated example, we provide and . In , we show the flow of a typical analysis process:

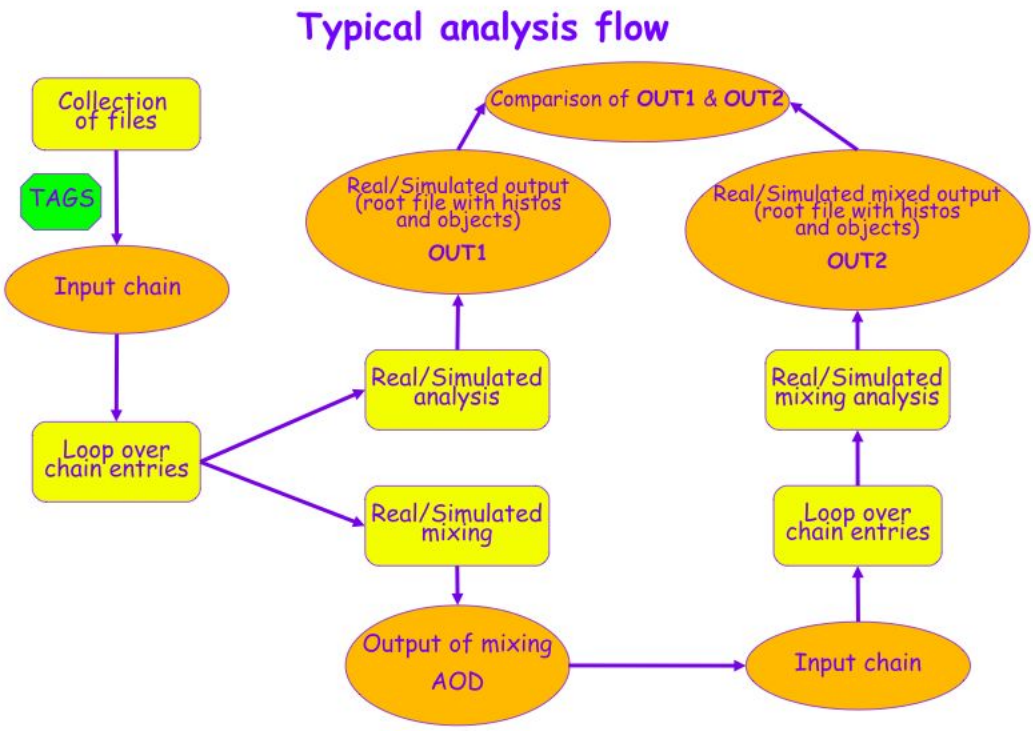


Figure 18: The flow of a typical physics analysis.

A user interacts with the file catalogue and creates a tag collection (the reader should check section for the instructions on how to perform this action). Then, the Event Tag System is queried (details on how we can do this will be provided in the next sections) and a chain of files is created. While looping through the entries of this chain, we split our analysis into two branches. On the first of, we analyze the (real or simulated) data and we provide as a last step an output ROOT file with histograms. On the second

branch, we mix the events (event mixing method) in order to express our background. This second branch creates an output of the form of an Analysis Object Data (AOD) [197]. A new chain is then created from this AOD and after analyzing the corresponding entries (analysis of mixed events), the output ROOT file with the relevant histograms is created. Finally, a last process is launched that compares the two output ROOT files and extracts the studied signal.

shows how we can integrate the previous use case to the new framework. The first steps are identical: A user interacts with the file catalogue and creates a tag collection (as described in section), which is used as an input to query the Event Tag System and create a chain of files. This chain is the first input container and is assigned to the **AliAnalysisManager**. In parallel, we define two tasks which are also assigned to the manager and are both linked to the same input container (chain): The first one analyzes the input data and creates an output container (ROOT file with histograms - signal + background plots), while the second is designed to mix the events and create a second output container (AOD). In order to analyze our mixed events, we initialize a second analysis manager that links the input container (AOD) with the new task (analysis of mixed events) and create a third output container (ROOT file with histogram - background plots). Finally, the comparison task is added to the second manager. This task is triggered by the ending of the analysis of mixed events and takes two input containers (ROOT files having the signal + background plots and the pure background plots) while creating one output (extracted signal).

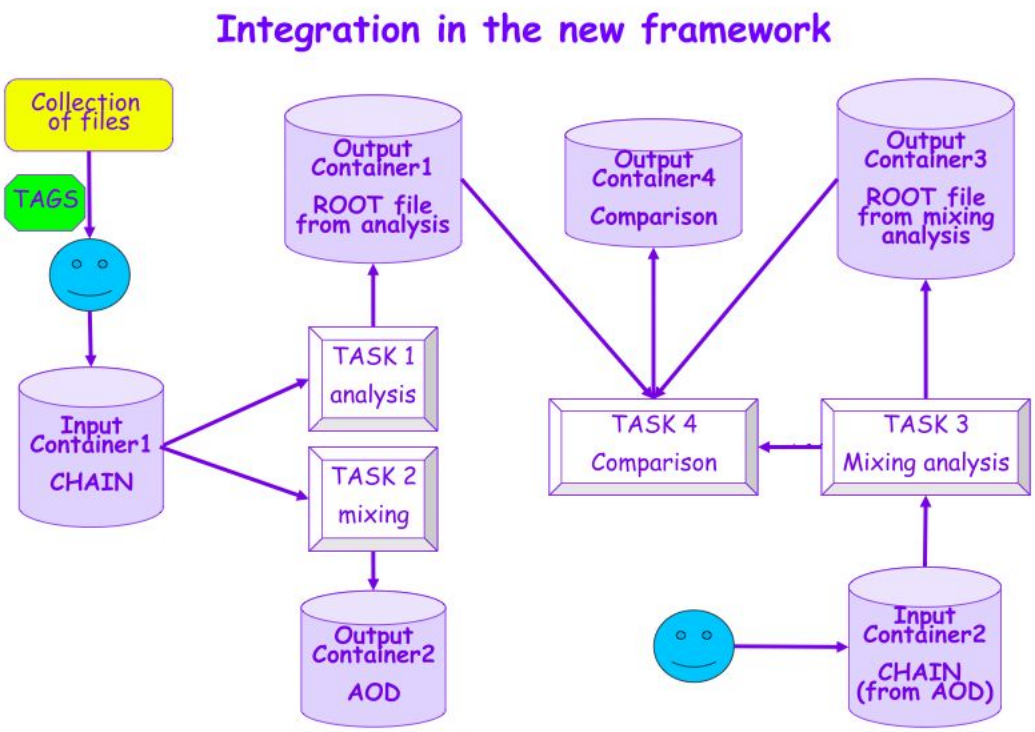


Figure 19: This figure shows how the physics analysis described in can be integrated in the new framework.

In the next paragraphs, we will be using the simplest possible case of manager and

task, which has been described, at the beginning of this section.

7.5 Interactive analysis with local ESDs

We assume that you have stored a few ESDs or AODs locally (the way to do this is described in detail in []), and that the first step regarding the creation of the tag-files, which are also stored locally (under `path`), for these ESDs/AODs has been finished [].

We setup the `par file` (see section for details) and then we invoke the following lines that summarize what we have to do in order to analyze data stored locally using the Event Tag System:

To specify cuts, we either do

```
AliRunTagCuts *runCutsObj = new AliRunTagCuts();
runCutsObj->SetRunId(340);
// add more run cuts here...

AliLHCTagCuts *lhcCutsObj = new AliLHCTagCuts();
lhcCutsObj->SetLHCStatus(“test”);

AliDetectorTagCuts *detCutsObj = new AliDetectorTagCuts();
detCutsObj->SetListOfDetectors(“TPC”);

AliEventTagCuts *evCutsObj = new AliEventTagCuts();
evCutsObj->SetMultiplicityRange(2, 100);
// add more event cuts here...
```

or we use

```
const char *runCutsStr = "fAliceRunId == 340";
const char *lhcCutsStr = "fLHCTag.fLHCState == test";
const char *detCutsStr = "fDetectorTag.fTPC == 1";
const char *evCutsStr = "fEventTag.fNumberOfTracks >= 2 &&
                        fEventTag.fNumberOfTracks <= 100";
// extend the strings to apply more cuts
```

Then, we chain the tag-files (keep in mind that the tag-files in this example are stored locally under `path`), we query the Event Tag System according to your cuts provided and we follow the example shown in section to create a manager and a task:

```
  AliTagAnalysis *tagAna = new AliTagAnalysis(“ESD”);
  tagAna->ChainLocalPaths(“path”);
  TChain *chain = tagAna->QueryTags(runCutsObj, lhcCutsObj,
detCutsObj, evCutsObj);
  //TChain *chain = tagAna->QueryTags(runCutsStr, lhcCutsStr,
detCutsStr, evCutsStr);

  //_____________________________________________//
  // Make the analysis manager
  AliAnalysisManager *mgr = new AliAnalysisManager
("TestManager");
  AliVEventHandler* esdH = new AliESDInputHandler;
  mgr->SetInputEventHandler(esdH);
```

```
  //_____________________________________________//
  // 1st Pt task
  AliAnalysisTaskPt *task1 = new AliAnalysisTaskPt("TaskPt");
  mgr->AddTask(task1);
  // Create containers for input/output
  AliAnalysisDataContainer *cinput1 = mgr->CreateContainer
("cchain1",TChain::Class(),AliAnalysisManager::kInputContainer);
  AliAnalysisDataContainer *coutput1 = mgr->CreateContainer
("chist1", TH1::Class(),
AliAnalysisManager::kOutputContainer,"Pt.ESD.root");

  //_____________________________________________//
  mgr->ConnectInput(task1,0,cinput1);
  mgr->ConnectOutput(task1,0,coutput1);
  if (!mgr->InitAnalysis()) return;
  mgr->PrintStatus();
  mgr->StartAnalysis("local",chain);
```

There are two possible ways to impose run- and event-cuts. The first is to create the objects, called **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and **AliEventTagCuts**, whose member functions will take care of your cuts. The second is to provide the strings that describe the cuts you want to apply on the run and on the event level. In the following we will describe both methods.

7.5.1 Object based cut strategy

The first step in the object based cut strategy is to create an **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and an **AliEventTagCuts** object:

```
AliRunTagCuts *runCutsObj = new AliRunTagCuts();
AliLHCTagCuts *lhcCutsObj = new AliLHCTagCuts();
AliDetectorTagCuts *detCutsObj = new AliDetectorTagCuts();
AliEventTagCuts *evCutsObj = new AliEventTagCuts();
```

These objects are used to describe the cuts imposed to your analysis, in order to reduce the number of runs and events to be analyzed to the ones effectively satisfying your criteria. There are many selections possible and they are provided as member functions of the two classes **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and **AliEventTagCuts** class. In case the member functions describe a range of an entity, the run, LHC status, detector configuration or event will pass the test if the described entity lies inclusively within the limits $\texttt{low} \leq \texttt{value} \leq \texttt{high}$. In case of only one argument, the run or event will pass the test if the entity is equal to the input flag or mask ($\texttt{value} == \texttt{flag}$, $\texttt{value} == \texttt{mask}$) or, in case of a 'Max' or 'Min' identifier, if the run or event quantity is lower or equal ($\texttt{quantity} \leq \texttt{value}$) or higher or equal ($\texttt{quantity} \geq \texttt{value}$) than the provided value. A full list of available run and event cut functions can be found in Appendix\,\ref{App:ObjectCuts}.

Let us consider only a cut on the run number, a cut on the LHC status, a cut on the detector configuration and one on the multiplicity: All events with run-numbers other than 340, with LHC status other than “test”, with the TPC not included and with less than 2 and more than 100 particles will be discarded.

```
runCutsObj->SetRunId(340);
```

```
lhcCutsObj->SetLHCStatus(“test”);
detCutsObj->SetListOfDetectors(“TPC”);
evCutsObj->SetMultiplicityRange(2, 100);
```

You can add as many other cuts as you like here.

7.5.2 String based cut strategy

Contrary to the object based cut strategy, you also have the possibility to provide your desired cut criteria as strings. You can do that by creating two separate strings, one for the run cuts and one for the event cuts. The syntax is based on C and the string is later evaluated by the **TTreeFormula** mechanism of ROOT. Therefore a wide range of operators is supported (see the ROOT manual [] for details). The variables used to describe the run and event properties are the data members of the **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and the **AliEventTagCuts** classes. Because of the enhanced number of available operators, this system provides more flexibility.

In order to create the same cuts as in the object based example above, the two strings should look like this:

```
const char *runCutsStr = "fAliceRunId == 340";
const char *lhcCutsStr = "fLHCTag.fLHCState == test";
const char *detCutsStr = "fDetectorTag.fTPC == 1";
const char *evCutsStr = "fEventTag.fNumberOfTracks >= 2 &&
fEventTag.fNumberOfTracks <= 100";
```

The full list of available data members to cut on can be found in Appendix\,\ref {App:StringCuts}. Within the quotes you can easily extend your cut statements in C style syntax.

Regardless of the way you choose to define your cuts, you create an **AliTagAnalysis** object, which is responsible to actually perform your desired analysis task.

```
AliTagAnalysis *TagAna = new AliTagAnalysis(“ESD”);
```

You have to provide this object with the locally stored tags since we assumed at the beginning of this section that these files were created and were stored locally (in the next section we will see how we can use the Grid stored tags). In order to do this you have to specify the correct path where the tag file(s) is/are located

```
tagAna->ChainLocalTags("path");
```

This function will pick up every file under the given path ending with `tag.root`. Now you ask your **AliTagAnalysis** object to return a **TChain**, imposing the event cuts as defined in the **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and **AliEventTagCuts** objects or by the two strings representing the run and event tags:

```
TChain *chain = tagAna->QueryTags(runCutsObj, lhcCutsObj, detCutsObj,
evCutsObj);
```

```
TChain *chain = tagAna->QueryTags(runCutsStr, lhcCutsStr, detCutsStr,
evCutsStr);
```

The two arguments must be of the same type: two Ali*TagCuts objects or two strings! If you don't want to impose run- or event-cuts, simply provide a NULL pointer.

Finally, you process the **TChain** by invoking your analysis manager with the following line of code:

```
  //_____________________________________________//
  // Make the analysis manager
  AliAnalysisManager *mgr = new AliAnalysisManager
("TestManager");
  AliVEventHandler* esdH = new AliESDInputHandler;
  mgr->SetInputEventHandler(esdH);
  //_____________________________________________//
  // 1st Pt task
  AliAnalysisTaskPt *task1 = new AliAnalysisTaskPt("TaskPt");
  mgr->AddTask(task1);
  // Create containers for input/output
  AliAnalysisDataContainer *cinput1 = mgr->CreateContainer
("cchain1",TChain::Class(),AliAnalysisManager::kInputContainer);
  AliAnalysisDataContainer *coutput1 = mgr->CreateContainer
("chist1", TH1::Class(),
AliAnalysisManager::kOutputContainer,"Pt.ESD.root");


  //_____________________________________________//
  mgr->ConnectInput(task1,0,cinput1);
  mgr->ConnectOutput(task1,0,coutput1);
  if (!mgr->InitAnalysis()) return;
  mgr->PrintStatus();
  mgr->StartAnalysis("local",chain);
```

One thing to mention is that even in case you do not want to imply any run- and event-cuts, it is useful to use the chain of commands described above. You would then simply pass two `NULL` pointers to the **AliTagAnalysis** class. The advantage of this procedure is that this setup takes care of chaining all the necessary files for you.

All the files needed to run this example can be found inside the PWG2 module of AliRoot under the AnalysisMacros/Local directory.

7.6 Interactive analysis with GRID ESDs

Once the first step described in Section was successful and we are satisfied from both the code and the results, we are ready to validate our code on a larger data sample. In this section, we will describe how we can analyze interactively (that is sitting in front of a terminal and getting back the results in our screen) files that are stored in the Grid. We will once again concentrate on the case where we use the Event Tag System [,].

The first thing we need to create is a collection of tag-files by querying the file catalogue (for details on this process the reader should look either in the example of section or in [,]). These tag-files, which are registered in the Grid, are the official ones created as a last step of the reconstruction code []. Once we have a valid xml collection, we launch a ROOT session, we setup the `par files` (the way to do this has been described in detail in section), we apply some selection criteria and we query the Event Tag System which returns the desired events in the proper format (a **TChain** along with the associated list of events that satisfy our cuts). The following lines give a snapshot of how a typical code should look like:

Usage of **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and **AliEventTagCuts** classes

```
// Case where the tag-files are stored in the file catalog
// tag.xml is the xml collection of tag-files that was produced
// by querying the file catalog.
TGrid::Connect("alien://");
TAlienCollection* coll = TAlienCollection::Open("tag.xml");
TGridResult* tagResult = coll->GetGridResult("",0,0);

// Create a RunTagCut object
AliRunTagCuts *runCutsObj = new AliRunTagCuts();
runCutsObj->SetRunId(340);

// Create a LHCTagCut object
AliLHCTagCuts *lhcCutsObj = new AliLHCTagCuts();
lhcCutsObj->SetLHCStatus(“test”);

// Create a DetectorTagCut object
AliDetectorTagCuts *detCutsObj = new AliDetectorTagCuts();
detCutsObj->SetListOfDetectors(“TPC”);

// Create an EventTagCut object
AliEventTagCuts *evCutsObj = new AliEventTagCuts();
evCutsObj->SetMultiplicityRange(2, 100);

// Create a new AliTagAnalysis object and chain the grid stored tags
AliTagAnalysis *tagAna = new AliTagAnalysis(“ESD”);
tagAna->ChainGridTags(tagResult);

// Cast the output of the query to a TChain
TChain *chain = tagAna->QueryTags(runCutsObj, lhcCutsObj, detCutsObj,
evCutsObj);
```

```
//_____________________________________________//
// Make the analysis manager
AliAnalysisManager *mgr = new AliAnalysisManager("TestManager");
AliVEventHandler* esdH = new AliESDInputHandler;
mgr->SetInputEventHandler(esdH);
//_____________________________________________//
// 1st Pt task
AliAnalysisTaskPt *task1 = new AliAnalysisTaskPt("TaskPt");
mgr->AddTask(task1);
// Create containers for input/output
AliAnalysisDataContainer *cinput1 = mgr->CreateContainer
("cchain1",TChain::Class(),AliAnalysisManager::kInputContainer);
AliAnalysisDataContainer *coutput1 = mgr->CreateContainer("chist1",
TH1::Class(),AliAnalysisManager::kOutputContainer,"Pt.ESD.root");

//_____________________________________________//
mgr->ConnectInput(task1,0,cinput1);
mgr->ConnectOutput(task1,0,coutput1);
if (!mgr->InitAnalysis()) return;
mgr->PrintStatus();
```

```
  mgr->StartAnalysis(“local”,chain);
```

Usage of string statements

```
  // Case where the tag-files are stored in the file catalog
  // tag.xml is the xml collection of tag-files that was produced
  // by querying the file catalog.
  TGrid::Connect("alien://");
  TAlienCollection* coll = TAlienCollection::Open("tag.xml");
  TGridResult* tagResult = coll->GetGridResult("",0,0);

  //Usage of string statements//
  const char* runCutsStr = "fAliceRunId == 340";
  const char *lhcCutsStr = "fLHCTag.fLHCState == test";
  const char *detCutsStr = "fDetectorTag.fTPC == 1";
  const char* evCutsStr = "fEventTag.fNumberOfTracks >= 2 &&
fEventTag.fNumberOfTracks <= 100";

  // Create a new AliTagAnalysis object and chain the grid stored tags
  AliTagAnalysis *tagAna = new AliTagAnalysis(“ESD”);
  tagAna->ChainGridTags(tagResult);

  // Cast the output of the query to a TChain
  TChain *chain = tagAna->QueryTags(runCutsStr, lhcCutsStr, detCutsStr,
evCutsStr);
```

```
  //_____________________________________________//
  // Make the analysis manager
  AliAnalysisManager *mgr = new AliAnalysisManager("TestManager");
  AliVEventHandler* esdH = new AliESDInputHandler;
  mgr->SetInputEventHandler(esdH);
  //_____________________________________________//
  // 1st Pt task
  AliAnalysisTaskPt *task1 = new AliAnalysisTaskPt("TaskPt");
  mgr->AddTask(task1);
  // Create containers for input/output
  AliAnalysisDataContainer *cinput1 = mgr->CreateContainer
("cchain1",TChain::Class(),AliAnalysisManager::kInputContainer);
  AliAnalysisDataContainer *coutput1 = mgr->CreateContainer("chist1",
TH1::Class(),AliAnalysisManager::kOutputContainer,"Pt.ESD.root");

  //_____________________________________________//
  mgr->ConnectInput(task1,0,cinput1);
  mgr->ConnectOutput(task1,0,coutput1);
  if (!mgr->InitAnalysis()) return;
  mgr->PrintStatus();
  mgr->StartAnalysis(“local”,chain);
```

We will now review the previous lines. Since we would like to access Grid stored files, we have to connect to the API server using the corresponding ROOT classes:

```
TGrid::Connect("alien://");
```

Then, we create a **TAlienCollection** object by opening the xml file (tag.xml) and we convert it to a **TGridResult**:

```
TAlienCollection* coll = TAlienCollection::Open("tag.xml");
TGridResult* tagResult = coll->GetGridResult("",0,0);
```

where `tag.xml` is the name of the file (which is stored in the working directory) containing the collection of tag-files.

The difference of the two cases is located in the way we apply the event tag cuts. In the first case, we create an **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and an **AliEventTagCuts** object and impose our criteria at the run- and event-level of the Event Tag System, while in the second we use the string statements to do so. The corresponding lines have already been described in the previous section.

Regardless of the way we define our cuts, we need to initialize an **AliTagAnalysis** object and chain the GRID stored tags by providing as an argument to the `ChainGridTags` function the **TGridResult** we had created before

```
AliTagAnalysis *tagAna = new AliTagAnalysis(“ESD”);
tagAna->ChainGridTags(tagResult);
```

We then query the Event Tag System, using the imposed selection criteria and we end up having the chain of ESD files along with the associated event list (list of the events that fulfil the criteria):

```
TChain *chain = tagAna->QueryTags(runCutsObj, lhcCutsObj, detCutsObj,
evCutsObj);
```

for the first case (usage of objects), or

```
TChain *chain = tagAna->QueryTags(runCutsStr, lhcCutsStr, detCutsStr,
evCutsStr);
```

for the second case (usage of sting statements).

Finally we process the **TChain** by invoking our implemented task using a manager:

```
  //_____________________________________________//
  // Make the analysis manager
  AliAnalysisManager *mgr = new AliAnalysisManager("TestManager");
  AliVEventHandler* esdH = new AliESDInputHandler;
  mgr->SetInputEventHandler(esdH);
  //_____________________________________________//
  // 1st Pt task
  AliAnalysisTaskPt *task1 = new AliAnalysisTaskPt("TaskPt");
  mgr->AddTask(task1);
  // Create containers for input/output
  AliAnalysisDataContainer *cinput1 = mgr->CreateContainer
("cchain1",TChain::Class(),AliAnalysisManager::kInputContainer);
  AliAnalysisDataContainer *coutput1 = mgr->CreateContainer("chist1",
TH1::Class(),AliAnalysisManager::kOutputContainer,"Pt.ESD.root");

  //_____________________________________________//
  mgr->ConnectInput(task1,0,cinput1);
  mgr->ConnectOutput(task1,0,coutput1);
  if (!mgr->InitAnalysis()) return;
  mgr->PrintStatus();
```

```
mgr->StartAnalysis("local",chain);
```

All the files needed to run this example can be found inside the PWG2 module of AliRoot under the AnalysisMacros/Interactive directory.

7.7Batch analysis

In this section, we will describe the batch framework. We will first describe the flow of the procedure; we dedicate a sub-section to describe in detail the integration of the Event Tag System in the batch sessions. We will then use the next paragraphs to describe in detail the files needed to submit a batch job as well as the jdl syntax. Finally, we will provide a snapshot of a jdl and we will mention how we can submit a job on the grid and how we can see at any time its status.

7.7.1 Overview of the framework

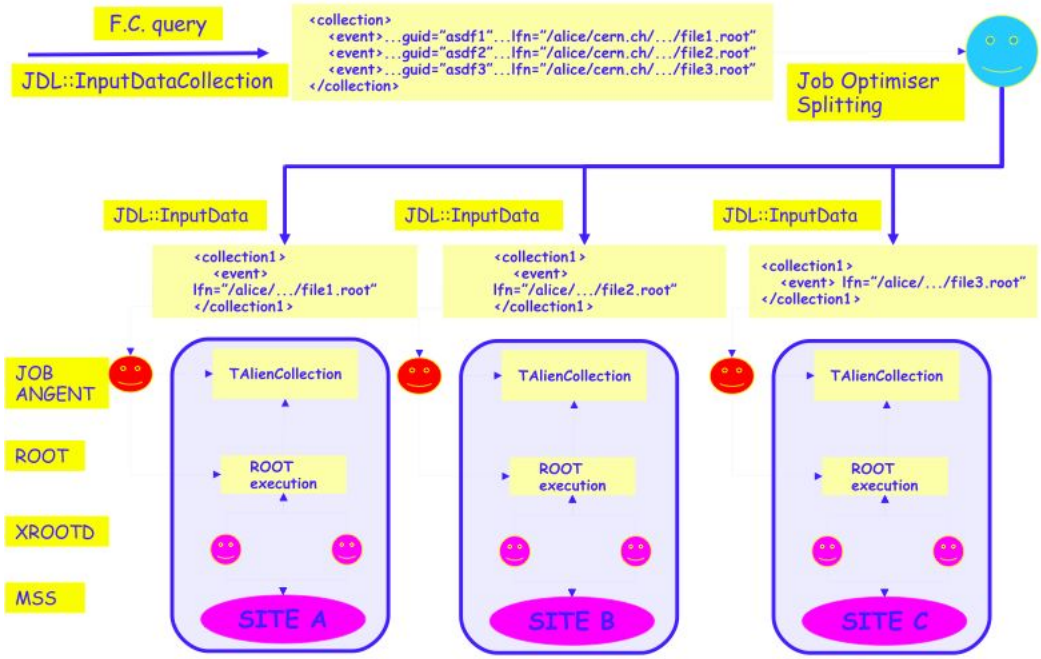


Figure 20: A schematic view of the flow of analysis in a batch session. Following the arrows, we have the initial xml collection that is listed in the jdl as an `InputDataCollection` *field. The optimizer takes this xml and splits the master job into several sub-jobs while in parallel writing new xml collections on every worker node. Then the respective job agents on every site start a ROOT or AliRoot session, read these new xml collections and interact with the xroot servers in order to retrieve the needed files. Finally, after the analysis is completed, a single output file is created for every sub-job.*

shows the flow of a batch session []. We start, as we have explained in section , by

querying the file catalogue and extracting a collection of files. This collection will be referenced by our jdl as an `InputDataCollection` field. Once we have created our jdl (a detailed description of the jdl syntax comes in the next sections) and all the files listed in it are in the proper place, we submit the job. The optimizer of the AliEn task queue parses the xml file and splits the master job into several smaller ones, each one assigned to a different site. In parallel, a new xml collection is written on every site, containing the information about the files to be analyzed on every worker node. This new collection will be noted in our jdl as an `InputDataList` field.

The corresponding job agent of every site starts the execution of the ROOT (in case we use the combination ROOT + `par file`) or AliRoot session, parses this new xml collection and interacts with the xrootd servers in order to retrieve the files that are listed inside these collections from the storage system. The analysis of these different sets of files results into the creation of several output files, each one containing the output of a sub-job. The user is responsible to launch a post-process that will loop over the different output files in order to merge them (an example on how to merge output histograms will be described in the next section).

7.7.2 Using the Event Tag System

To use the Event Tag System, we have to use some AliRoot classes that are in the **STEER** module. The main classes, as described in a previous section (section), are the **AliTagAnalysis**, **AliRunTagCuts**, **AliLHCTagCuts**, **AliDetectorTagCuts** and **AliEventTagCuts**. In order to use the Event Tag System in a batch session, we need to perform an initial step described in on the client side: starting from a tag xml collection obtained by querying the file catalogue, we define our selection criteria according to our physics analysis and we create a new xml collection having this time the information about the AliESDs. The user should realize that the initial xml collection, named in the examples as `tag.xml`, held the information about the location of the tag-files inside the file catalogue. Instead, what we create at this step is a new xml collection that will refer to the location of the ESD files. In this xml collection we also list the events that satisfy the imposed selection criteria for every ESD file. The following lines show how we can generate a new xml collection (you can find these lines in the `CreateXML.C` macro inside the STEER module):

Usage of **AliRunTagCuts** and **AliEventTagCuts** classes

```
// Case where the tag-files are stored in the file catalog
// tag.xml is the xml collection of tag-files that was produced
// by querying the file catalog.
TGrid::Connect("alien://");
TAlienCollection* coll = TAlienCollection::Open("tag.xml");
TGridResult* tagResult = coll->GetGridResult("",0,0);

// Create a new AliTagAnalysis object
AliTagAnalysis *tagAna = new AliTagAnalysis(“ESD”);

// Create a tag chain by providing the TGridResult
// from the previous step as an argument
tagAna->ChainGridTags(tagResult);
```

```
  //Usage of AliRunTagCuts & AliEventTagCuts classes//
  // Create a RunTagCut object
  AliRunTagCuts *runCutsObj = new AliRunTagCuts();
  runCutsObj->SetRunId(340);

  // Create a LHCTagCut object
  AliLHCTagCuts *lhcCutsObj = new AliLHCTagCuts();
  lhcCutsObj->SetLHCStatus(“test”);

  // Create a DetectorTagCut object
  AliDetectorTagCuts *detCutsObj = new AliDetectorTagCuts();
  detCutsObj->SetListOfDetectors(“TPC”);

  // Create an EventTagCut object
  AliEventTagCuts *evCutsObj = new AliEventTagCuts();
  evCutsObj->SetMultiplicityRange(2, 100);

  // Create the esd xml collection:the first argument is the
  // collection name while the other two are the imposed criteria
  tagAna->CreateXMLCollection("global", runCutsObj, lhcCutsObj,
detCutsObj, evCutsObj);
```

Usage of string statements

```
  // Case where the tag-files are stored in the file catalog
  // tag.xml is the xml collection of tag-files that was produced
  // by querying the file catalog.
  TGrid::Connect("alien://");
  TAlienCollection* coll = TAlienCollection::Open("tag.xml");
  TGridResult* tagResult = coll->GetGridResult("",0,0);

  // Create a new AliTagAnalysis object
  AliTagAnalysis *tagAna = new AliTagAnalysis();

  // Create a tag chain by providing the TGridResult
  // from the previous step as an argument
  tagAna->ChainGridTags(tagResult);

  //Usage of string statements//
  const char* runCutsStr = "fAliceRunId == 340";
  const char *lhcCutsStr = "fLHCTag.fLHCState == test";
  const char *detCutsStr = "fDetectorTag.fTPC == 1";
  const char* evCutsStr = "fEventTag.fNumberOfTracks >= 2 &&
  fEventTag.fNumberOfTracks <= 100";

  // Create the esd xml collection:the first argument is the
collection name
  // while the other two are the imposed criteria
  tagAna->CreateXMLCollection("global", runCutsStr, lhcCutsStr,
detCutsStr, evCutsStr);
```

The reader should be familiar by now with the previous lines since they have already

been described in detail in section . The new thing is the very last line of code where we call the `CreateXMLCollection` function of the **AliTagAnalysis** class which takes as arguments the name of the output xml collection (collection of ESDs) and the four run-lhc-detector and event-tag cuts (objects or strings). This output collection will be created in the working directory.

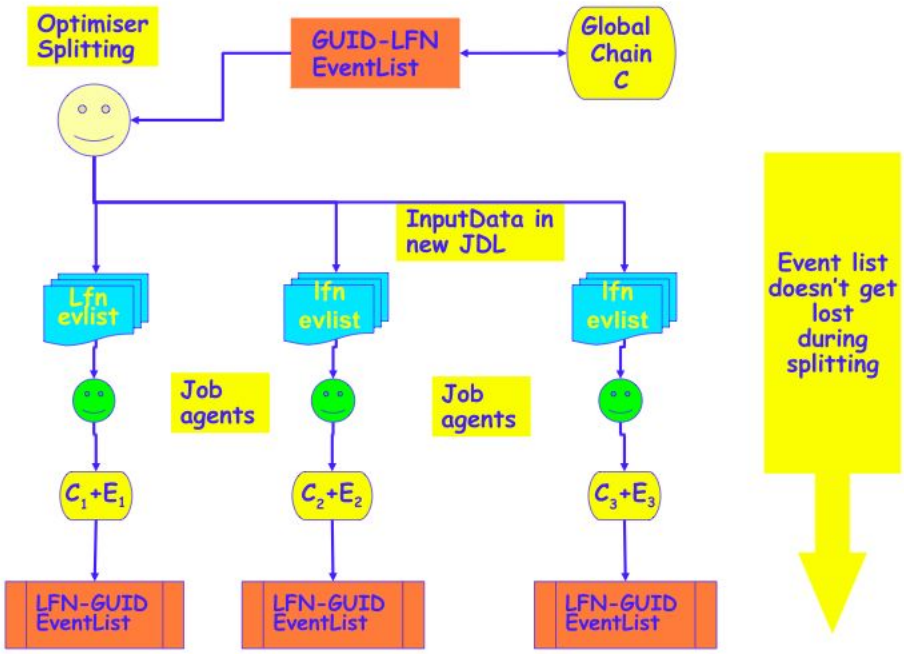


*Figure 21: A schematic view of the flow of the analysis procedure in a batch session using the Event Tag System. Following the arrows, we have the initial xml collection which is created by the **AliTagAnalysis** class listed in the jdl as an **InputDataCollection** field. The optimizer takes this xml once the master job is submitted and splits it into several sub-jobs while in parallel writing new xml collections on every worker node. These xml collections hold the information about the events that satisfy the imposed selection criteria, grouped by file: the analysis is performed only on these events on every worker node.*

The next step will be to create a jdl file inside of which this newly created xml collection (name global.xml in our example) will be define as an `InputDataCollection` field. Then we once again submit the job and the optimizer parses the xml and splits the master job into several sub-jobs. In parallel, a new xml collection is written on every site, containing the information about the files that will be analyzed on every worker node as well as the corresponding list of events that satisfy the imposed selection criteria for every file. Thus, on every worker node, we will analyze the created chain along with the associated event list as described in . Once finished, we will get several output files, over which we will have to loop with a post-process in order to merge them [, , [63]].

In the following paragraphs we will provide some practical information about the batch

sessions, starting from the files needed to submit a job, the jdl syntax etc.

7.7.3 Files needed

The files needed in order to submit a batch job are listed below []:

- **Executable**: This is a file that should be stored under the $HOME/bin AliEn directory of each user. It is used to start the ROOT/AliRoot session on every worker node. Users can always use existing executables that can be found under /bin. An example is given below.
- **Par file**: A `par file` is a tarball containing the header files and the source code of AliRoot, needed to build a certain library. It is used in the case where we do not want to launch AliRoot but instead we want to be flexible by launching ROOT along with the corresponding AliRoot library (e.g ROOT and the lib*.so). It is not compulsory although it is recommended to use a par file in an analysis.
- **Macro**: It is the file that each user needs to implement. Inside the macro we setup the par file (in case we use it) and we load the needed libraries. Then we open the input xml collection and convert it into a chain of trees. The next step is to create an `AliAnalysisManager`, assign a task to the manager and define the input and output containers. Finally, we process this chain with a selector. A snapshot of such a file has already been given in section .
- **XML collection**: This is the collection created either by directly querying the file catalogue (in the case where we don't use the Event Tag System) or by querying the Event Tag System (case described in the previous paragraph).
- **JDL**: This is a compulsory file that describes the input/output files as well as the packages that we are going to use. A detailed description about the JDL fields is provided in the next lines.

Example of an “executable”

```
#!/bin/bash

echo ============================
echo $PATH
echo $LD_LIBRARY_PATH
echo ============================

root -b -x runBatch.C;
```

7.7.4 JDL syntax

In this section we will try to describe in detail the different jdl fields [64].

- **Executable**: It is the only compulsory field of the JDL where we give the logical file name (lfn) of the executable that should be stored in /bin or $VO/bin or $HOME/bin. A typical syntax can be:

```
Executable="root.sh";
```

- **Packages**: The definition of the packages that will be used in the batch session. The different packages installed can be found by typing `packages` in the AliEn shell [198]. A typical syntax can be:

```
Packages={"APISCONFIG::V2.4","VO_ALICE@ROOT::v5-16-00"};
```

- **Jobtag**: A comment that describes the job. A typical syntax can be:

```
Jobtag={"comment:AliEn Tutorial batch example"};
```

- **InputFile**: In this field we define the files that will be transported to the node where the job will run and are needed for the analysis. A typical syntax can be:

```
InputFile= {
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/AliAnalysisTaskPt.cxx
",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/AliAnalysisTaskPt.h",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/STEERBase.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/ESD.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/AOD.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/ANALYSIS.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/runBatch.C"};
```

- **InputData**: This field, when defined, requires that the job will be executed in a site close to files specified here. This is supposed to be used in the case where you don't want to use a collection of files. It should be pointed out that it is not really practical because it implies that each user writes a large number of lines in the jdl, thus making it difficult to handle. It should be pointed out that this approach can be useful in the case where we use a few files. A typical syntax can be:

```
InputFile= {
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/PDC06/001/AliESDs.root"}
```

- **InputDataList**: This is the name of the xml file created by the job agent after the job has been split, containing the lfn of the files of the closest storage element. A typical syntax can be:

```
InputDataList="wn.xml";
```

- **InputDataListFormat**: The format of the previous field. It can be either ``xml-single'' where we imply that every xml entry corresponds to one file or “xml-group” where we imply that a new set of files starts every time the base filename appears (e.g. xml containing AliESDs.root, Kinematics.root, galice.root). In the context of this note, where we analyze only ESDs and not the generator information, we should use the first option. A typical syntax can be:

```
InputDataListFormat="xml-single";
```

- **OutputFile**: Here we define the files that will be registered in the file catalogue once the job finishes. If we don't define the storage element, then the files will be registered in the default one which at the moment is Castor2 at CERN. A typical syntax can be:

```
OutputFile={"stdout@ALICE::CERN::se",
"stderr@ALICE::CERN::Castor2","*.root@ALICE::CERN::se"};
```

- **OutputDir**: Here we define the directory in the file catalogue under which the output files and archives will be stored. A typical syntax can be:

```
OutputDir="/alice/cern.ch/user/p/pchrist/Balance/output";
```

- **OutputArchive**: Here we define the files that we want to be put inside an archive. It is recommended that the users use this field in order to place all their output files in such an archive which will be the only registered file after a job finishes. This is essential in the case of storage systems such as Castor, which are not effective in handling small files. A typical syntax can be:

```
OutputArchive={"logarchive:stdout,stderr,*.log@Alice::CERN::se",
"rootarchive.zip:*.root@Alice::CERN::se"};
```

- **Validationcommand**: Specifies the script to be used as a validation script (used for production). A typical syntax can be:

```
Validationcommand =
"/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/validation.sh";
```

- **Email**: If this field is defined, then you'll be informed that your job has finished from an e-mail. A typical syntax can be:

```
Email="Panos.Christakoglou@cern.ch";
```

- **TTL**: One of the important fields of the JDL. It allows the user to define the maximum time in seconds the job will run. This field is used by the optimizer for the ordering and the assignment of the priority for each job. Lower value of TTL provides higher probability for the job to run quickly after submission. If the running time exceeds the one defined in this field, then the job is killed automatically. The value of this field should not exceed 100000 sec. A typical syntax can be:

```
TTL = "21000";
```

- **Split**: Used in the case we want to split our master job into several sub-jobs. Usually the job is split per storage element (se). A typical syntax can be:

```
Split="se";
```

- **SplitMaxInputFileNumber**: Used to define the maximum number of files that will be analyzed on every worker node. A typical syntax can be:

```
SplitMaxInputFileNumber="100";
```

In summary, the following lines give a snapshot of a typical jdl:

Example of a JDL file

```
# this is the startup process for root
Executable="root.sh";
Jobtag={"comment:AliEn tutorial batch example - ESD"};

# we split per storage element
Split="se";

# we want each job to read 100 input files
SplitMaxInputFileNumber="5";
```

```
# this job has to run in the ANALYSIS partition
Requirements=( member(other.GridPartitions,"Analysis") );

# validation command
Validationcommand
="/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/validation.sh";

# we need ROOT and the API service configuration package
Packages={"APISCONFIG::V2.4","VO_ALICE@ROOT::v5-16-00"};

# Time to live
TTL = "30000";

# Automatic merging
Merge={"Pt.ESD.root:/alice/jdl/mergerootfile.jdl:Pt.ESD.Merged.root"};

# Output dir of the automatic merging
MergeOutputDir="/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/output/";

#ROOT will read this collection file to know, which files to analyze
InputDataList="wn.xml";

#ROOT requires the collection file in the xml-single format
InputDataListFormat="merge:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH
/global.xml";

# this is our collection file containing the files to be analyzed
InputDataCollection="LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/gl
obal.xml,nodownload";

InputFile= {
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/runBatch.C",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/STEERBase.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/ESD.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/AOD.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/ANALYSIS.par",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/AliAnalysisTaskPt.h",
"LF:/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/AliAnalysisTaskPt.cxx"
};

# Output archive
OutputArchive=
{"log_archive:stdout,stderr,*.log@Alice::CERN::se","root_archive.zip:*.
root@Alice::CERN::se"};

# Output directory
OutputDir="/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/output/#alien_c
ounter#";

# email
Email="Panos.Christakoglou@cern.ch";
```

7.7.5 Job submission - Job status

After creating the jdl and registering all the files needed, we are ready to submit our batch job []. This can be done by typing:

```
submit $<filename>$.jdl
```

at the AliEn shell prompt. If there is no mistake in our JDL, the job will be assigned a JOBID. We can always see what are the jobs submitted by a certain user by typing

```
top -user $<username>$
```

Later on, we can check its status by typing:

```
ps -trace $<JOBID>$
```

The different states are:

- **INSERTING**: The job is waiting to be processed by the optimizer.
- **SPLITTING**: The optimizer starts splitting the job if this is requested in the JDL.
- **SPLIT**: Several sub-jobs were created from the master job.
- **WAITING**: The job is waiting to be assigned to a job agent that fulfils its requirements.
- **ASSIGNED**: A job agent is about to pick up this job.
- **STARTED**: The job agent is preparing the input sandbox and transferring the files listed in the `InputFile` field.
- **RUNNING**: The executable has started running.
- **SAVING**: The executable has finished running and the job agent saves the output to the specified storage elements.
- **SAVED**: The agent has successfully stored all the output files which are not available yet in the file catalogue.
- **DONE**: The central optimizer has registered the output in the catalogue.

Finally, as long as a job status changes to **RUNNING**, a user can check its stdout and stderr by typing:

```
spy $<JOBID>$ stdout
spy $<JOBID>$ stderr
```

at the AliEn prompt.

7.7.6 Merging the output

Assuming that everything worked out and that the status of the job we had submitted has turned to **DONE**, we are ready to launch the post-process that will access the different output files from every sub-job and merge them. We will concentrate in the case where the output files contain simple histograms (case which may represent the majority of the physics analyses). If the output files contain other analysis objects, then we should provide our own merge functions. The following lines should be put inside the jdl file:

JDL lines that launch the automatic merging of the output

```
# Automatic merging
Merge={"Pt.ESD.root:/alice/jdl/mergerootfile.jdl:Pt.ESD.Merged.root"};

# Output dir of the automatic merging
MergeOutputDir="/alice/cern.ch/user/p/pchrist/Tutorial/BATCH/output/";
```

The previous lines allow the system to submit the `mergerootfile.jdl` that expects the output files to be named `Pt.ESD.root` and creates the `Pt.ESD.Merged.root` in the directory defined in the `MergeOutputDir` field.

All the needed files to run these examples can be found inside the PWG2 module of AliRoot under the AnalysisMacros/Batch directory.

7.8 Run-LHC-Detector and event level cut member functions

7.8.1 Run level member functions

```
  void SetRunId(Int_t Pid);
  void SetMagneticField(Float_t Pmag);
  void SetRunStartTimeRange(Int_t t0, Int_t t1);
  void SetRunStopTimeRange(Int_t t0, Int_t t1);
  void SetAlirootVersion(TString v);
  void SetRootVersion(TString v);
  void SetGeant3Version(TString v);
  void SetRunQuality(Int_t Pn);
  void SetBeamEnergy(Float_t PE);
  void SetBeamType(TString Ptype);
  void SetCalibVersion(Int_t Pn);
  void SetDataType(Int_t i);
```

7.8.2 LHC level member functions

```
void SetLHCState(TString state);
void SetLHCLuminosityRange(Float_t low, Float_t high);
```

7.8.3 Detector level member functions

```
void SetListOfDetectors(const TString& detectors)
```

7.8.4 Event level member functions

```
  void SetNParticipantsRange(Int_t low, Int_t high);
  void SetImpactParamRange(Float_t low, Float_t high);

  void SetPrimaryVertexXRange(Float_t low, Float_t high);
  void SetPrimaryVertexYRange(Float_t low, Float_t high);
  void SetPrimaryVertexZRange(Float_t low, Float_t high);
  void SetPrimaryVertexFlag(Int_t flag);
```

```
  void SetPrimaryVertexZErrorRange(Float_t low, Float_t high);

  void SetTriggerMask(ULong64_t trmask);
  void SetTriggerCluster(UChar_t trcluster);

  void SetZDCNeutron1Range(Float_t low, Float_t high);
  void SetZDCProton1Range(Float_t low, Float_t high);
  void SetZDCEMRange(Float_t low, Float_t high);
  void SetZDCNeutron2Range(Float_t low, Float_t high);
  void SetZDCProton2Range(Float_t low, Float_t high);
  void SetT0VertexZRange(Float_t low, Float_t high);

  void SetMultiplicityRange(Int_t low, Int_t high);
  void SetPosMultiplicityRange(Int_t low, Int_t high);
  void SetNegMultiplicityRange(Int_t low, Int_t high);
  void SetNeutrMultiplicityRange(Int_t low, Int_t high);
  void SetNV0sRange(Int_t low, Int_t high);
  void SetNCascadesRange(Int_t low, Int_t high);
  void SetNKinksRange(Int_t low, Int_t high);

  void SetNPMDTracksRange(Int_t low, Int_t high);
  void SetNFMDTracksRange(Int_t low, Int_t high);
  void SetNPHOSClustersRange(Int_t low, Int_t high);
  void SetNEMCALClustersRange(Int_t low, Int_t high);
  void SetNJetCandidatesRange(Int_t low, Int_t high);

  void SetTopJetEnergyMin(Float_t low);
  void SetTopNeutralEnergyMin(Float_t low);
  void SetNHardPhotonsRange(Int_t low, Int_t high);
  void SetNChargedAbove1GeVRange(Int_t low, Int_t high);
  void SetNChargedAbove3GeVRange(Int_t low, Int_t high);
  void SetNChargedAbove10GeVRange(Int_t low, Int_t high);
  void SetNMuonsAbove1GeVRange(Int_t low, Int_t high);
  void SetNMuonsAbove3GeVRange(Int_t low, Int_t high);
  void SetNMuonsAbove10GeVRange(Int_t low, Int_t high);
  void SetNElectronsAbove1GeVRange(Int_t low, Int_t high);
  void SetNElectronsAbove3GeVRange(Int_t low, Int_t high);
  void SetNElectronsAbove10GeVRange(Int_t low, Int_t high);
  void SetNElectronRange(Int_t low, Int_t high);
  void SetNMuonRange(Int_t low, Int_t high);
  void SetNPionRange(Int_t low, Int_t high);
  void SetNKaonRange(Int_t low, Int_t high);
  void SetNProtonRange(Int_t low, Int_t high);
  void SetNLambdaRange(Int_t low, Int_t high);
  void SetNPhotonRange(Int_t low, Int_t high);
  void SetNPi0Range(Int_t low, Int_t high);
  void SetNNeutronRange(Int_t low, Int_t high);
  void SetNKaon0Range(Int_t low, Int_t high);
  void SetTotalPRange(Float_t low, Float_t high);
  void SetMeanPtRange(Float_t low, Float_t high);
  void SetTopPtMin(Float_t low);
  void SetTotalNeutralPRange(Float_t low, Float_t high);
```

```
  void SetMeanNeutralPtPRange(Float_t low, Float_t high);
  void SetTopNeutralPtMin(Float_t low);
  void SetEventPlaneAngleRange(Float_t low, Float_t high);
  void SetHBTRadiiRange(Float_t low, Float_t high);
```

7.9 String base object and event level tags

7.9.1 Variables for run cuts

```
  Int_t   fAliceRunId;                  //the run id
  Float_t fAliceMagneticField;          //value of the magnetic field
  Int_t   fAliceRunStartTimeMin;        //minimum run start date
  Int_t   fAliceRunStartTimeMax;        //maximum run start date
  Int_t   fAliceRunStopTimeMin;         //minmum run stop date
  Int_t   fAliceRunStopTimeMax;         //maximum run stop date
  TString fAlirootVersion;              //aliroot version
  TString fRootVersion;                 //root version
  TString fGeant3Version;               //geant3 version
  Bool_t  fAliceRunQuality;             //validation script
  Float_t fAliceBeamEnergy;             //beam energy cm
  TString fAliceBeamType;               //run type (pp, AA, pA)
  Int_t   fAliceCalibrationVersion;     //calibration version
  Int_t   fAliceDataType;               //0: simulation -- 1: data
```

7.9.2 Variables for LHC cuts

To invoke one of these cuts, please make sure to use the `fLHCTag.` identifier. Example: "`fLHCTag.fLHCState == "test"`".

```
TString fLHCState;      //LHC run conditions - comment
Float_t fLHCLuminosity; //the value of the luminosity
```

7.9.3 Variables for detector cuts

To invoke one of these cuts, please make sure to use the `fDetectorTag.` identifier. Example: "`fDetectorTag.fTPC == 1`".

```
Bool_t     fITSSPD;        //ITS-SPD active = 1
  Bool_t     fITSSDD;        //ITS-SDD active = 1
  Bool_t     fITSSSD;        //ITS-SSD active = 1
  Bool_t     fTPC;           //TPC active = 1
  Bool_t     fTRD;           //TRD active = 1
  Bool_t     fTOF;           //TOF active = 1
  Bool_t     fHMPID;         //HMPID active = 1
  Bool_t     fPHOS;          //PHOS active = 1
  Bool_t     fPMD;           //PMD active = 1
  Bool_t     fMUON;          //MUON active = 1
  Bool_t     fFMD;           //FMD active = 1
  Bool_t     fTZERO;         //TZERO active = 1
  Bool_t     fVZERO;         //VZERO active = 1
```

```
  Bool_t     fZDC;           //ZDC active = 1
  Bool_t     fEMCAL;         //EMCAL active = 1
```

7.9.4 Variables for event cuts

To invoke one of these cuts, please make sure to use the `fEventTag.` identifier. Example: "`fEventTag.fNParticipants < 100`".

```
  Int_t fNParticipantsMin, fNParticipantsMax;
  Float_t fImpactParamMin, fImpactParamMax;

  Float_t fVxMin, fVxMax;
  Float_t fVyMin, fVyMax;
  Float_t fVzMin, fVzMax;
  Int_t fPrimaryVertexFlag;
  Float_t fPrimaryVertexZErrorMin, fPrimaryVertexZErrorMax;

  ULong64_t fTriggerMask;
  UChar_t fTriggerCluster;

  Float_t fZDCNeutron1EnergyMin, fZDCNeutron1EnergyMax;
  Float_t fZDCProton1EnergyMin, fZDCProton1EnergyMax;
  Float_t fZDCNeutron2EnergyMin, fZDCNeutron2EnergyMax;
  Float_t fZDCProton2EnergyMin, fZDCProton2EnergyMax;
  Float_t fZDCEMEnergyMin, fZDCEMEnergyMax;
  Float_t fT0VertexZMin, fT0VertexZMax;

  Int_t fMultMin, fMultMax;
  Int_t fPosMultMin, fPosMultMax;
  Int_t fNegMultMin, fNegMultMax;
  Int_t fNeutrMultMin, fNeutrMultMax;
  Int_t fNV0sMin, fNV0sMax;
  Int_t fNCascadesMin, fNCascadesMax;
  Int_t fNKinksMin, fNKinksMax;

  Int_t fNPMDTracksMin, fNPMDTracksMax;
  Int_t fNFMDTracksMin, fNFMDTracksMax;
  Int_t fNPHOSClustersMin, fNPHOSClustersMax;
  Int_t fNEMCALClustersMin, fNEMCALClustersMax;
  Int_t fNJetCandidatesMin, fNJetCandidatesMax;

  Float_t fTopJetEnergyMin;
  Float_t fTopNeutralEnergyMin;

  Int_t fNHardPhotonCandidatesMin, fNHardPhotonCandidatesMax;
  Int_t fNChargedAbove1GeVMin, fNChargedAbove1GeVMax;
  Int_t fNChargedAbove3GeVMin, fNChargedAbove3GeVMax;
  Int_t fNChargedAbove10GeVMin, fNChargedAbove10GeVMax;
  Int_t fNMuonsAbove1GeVMin, fNMuonsAbove1GeVMax;
  Int_t fNMuonsAbove3GeVMin, fNMuonsAbove3GeVMax;
  Int_t fNMuonsAbove10GeVMin, fNMuonsAbove10GeVMax;
```

```
Int_t fNElectronsAbove1GeVMin, fNElectronsAbove1GeVMax;
Int_t fNElectronsAbove3GeVMin, fNElectronsAbove3GeVMax;
Int_t fNElectronsAbove10GeVMin,fNElectronsAbove10GeVMax;
Int_t fNElectronsMin, fNElectronsMax;
Int_t fNMuonsMin, fNMuonsMax;
Int_t fNPionsMin, fNPionsMax;
Int_t fNKaonsMin, fNKaonsMax;
Int_t fNProtonsMin, fNProtonsMax;
Int_t fNLambdasMin, fNLambdasMax;
Int_t fNPhotonsMin, fNPhotonsMax;
Int_t fNPi0sMin, fNPi0sMax;
Int_t fNNeutronsMin, fNNeutronsMax;
Int_t fNKaon0sMin, fNKaon0sMax;
Float_t fTotalPMin, fTotalPMax;
Float_t fMeanPtMin, fMeanPtMax;
Float_t fTopPtMin;
Float_t fTotalNeutralPMin, fTotalNeutralPMax;
Float_t fMeanNeutralPtMin, fMeanNeutralPtMax;
Float_t fTopNeutralPtMin;
Float_t fEventPlaneAngleMin, fEventPlaneAngleMax;
Float_t fHBTRadiiMin, fHBTRadiiMax;
```

7.10 Summary

To summarize, we tried to describe the overall distributed analysis framework by also providing some practical examples. The intention of this note, among other things, was to inform the users about the whole procedure starting from the validation of the code which is usually done locally up to the submission of GRID jobs.

We started off by showing how one can interact with the file catalogue and extract a collection of files. Then we presented how one can use the Event Tag System in order to analyze a few ESD/AOD files stored locally, a step which is necessary for the validation of the user's code (code sanity and result testing). The next step was the interactive analysis using GRID stored ESDs/AODs. This was also described in detail in Section . Finally, in Section , we presented in detail the whole machinery of the distributed analysis and we also provided some practical examples to create a new xml collection using the Event Tag System. We also presented in detail the files needed in order to submit a GRID job and on this basis we tried to explain the relevant JDL fields and their syntax.

We should point out once again, that this note concentrated on the usage of the whole metadata machinery of the experiment: that is both the file/run level metadata [] as well as the Event Tag System []. The latter is used because, apart from the fact that it provides us with a fast event filtering mechanism, it also takes care of the creation of the analyzed input sample in the proper format (**TChain**) in a fully transparent way. Thus, it is easy to plug it in as a first step in the new analysis framework [, 199], which is being developed and validated as this note was written.

8 Appendix

8.1 Kalman filter

Kalman filtering is quite a general and powerful method for statistical estimations and predictions. The conditions for its applicability are the following. A certain "system" is determined at any moment in time t_k by a state vector x_k. The state vector varies with time according to an evolution equation

$$x_k = f_k(x_{k-1}) + \varepsilon_k$$

It is supposed that f_k a known deterministic function and ε_k is a random vector of intrinsic "process noise" which has a zero mean value ($\langle \varepsilon_k \rangle = 0$) and a known covariance matrix ($\mathrm{cov}(\varepsilon_k) = Q_k$). Generally, only some function $h_k = h(x_k)$ of the state vector can be observed, and the result of the observation m_k is corrupted by a "measurement noise" δ_k:

$$m_k = h_k(x_k) + \delta_k$$

The measurement noise is supposed to be unbiased ($\langle \delta_k \rangle = 0$) and have a definite covariance matrix ($\mathrm{cov}(\delta_k) = V_k$). In many cases, a matrix H_k can represent the measurement function h_k:

$$m_k = H_k x_k + \delta_k$$

If, at a certain time t_{k-1}, we are given some estimates of the state vector $\tilde{x}_{k-1}$ and of its covariance matrix $\tilde{C}_{k-1} = \mathrm{cov}(\tilde{x}_{k-1} - x_{k-1})$, we can extrapolate these estimates to the next time slot t_k by means of formulas (this is called "prediction"):

$$\tilde{x}_k^{k-1} = f_k(\tilde{x}_{k-1})$$

$$\tilde{C}_k^{k-1} = F_k \tilde{C}_{k-1} F_k^T + Q_k, \quad F_k = \frac{\partial f_k}{\partial x_{k-1}}$$

The value of the predicted χ^2 increment can be also calculated:

$$(\chi^2)_k^{k-1} = (r_k^{k-1})^T (R_k^{k-1}) r_k^{k-1}, \quad r_k^{k-1} = m_k - H_k \tilde{x}_k^{k-1}, \quad R_k^{k-1} = V_k + H_k \tilde{C}_k^{k-1} H_k^T$$

The number of degrees of freedom is equal to the dimension of the vector m_k.

If at the moment t_k, together with the results of prediction, we also have the results of the state vector measurement, this additional information can be combined with the prediction results (this is called "filtering"). As a consequence, the estimation of the state vector improves with respect to the previous step:

$$\tilde{x}_k = {}_k^{k=1} + K_k\left(m_k - H_k \tilde{x}_k^{k-1}\right)$$
$$C_k^{\circ} = C_k^{k-1} - K_k H_k C_k^{k-1}$$

where K_k is the Kalman gain matrix $K_k = C_k^{k-1} H_k^T \left(V_k + H_k C_k^{k-1} H_k^T\right)^{-1}$. Finally, the next formula gives us the value of the filtered χ^2 increment:

$$\chi_k^2 = \left(r_k\right)^T \left(R_k\right)^{-1} r_k, \quad r_k = m_k - H_k \tilde{x}_k, \quad R_k = V_k - H_k C_k^{\circ} H_k^T$$

It can be shown that the predicted χ^2 value is equal to the filtered one:

$$\left(\chi^2\right)_k^{k-1} = \chi_k^2$$

The "prediction" and "filtering" steps are repeated as many times as we have measurements of the state vector.

8.2 Bayesian approach for combined particle identification

Particle identification over a large momentum range and for many particle species is often one of the main design requirements of high energy physics experiments. The ALICE detectors are able to identify particles with momenta from 0.1 GeV/c up to 10 GeV/c. This can be achieved by combining several detecting systems that are efficient in some narrower and complementary momentum sub-ranges. The situation is complicated by the amount of data to be processed (about 10^7 events with about 10^4 tracks in each). Thus, the particle identification procedure should satisfy the following requirements:

- It should be as much as possible automatic.
- It should be able to combine PID signals of different nature (e.g. dE/dx and time-of-flight measurements).
- When several detectors contribute to the PID, the procedure must profit from this situation by providing an improved PID.
- When only some detectors identify a particle, the signals from the other detectors must not affect the combined PID.
- It should take into account the fact that, due to different event and track selection, the PID depends on the kind of analysis.

In this report we will demonstrate that combining PID signals in a Bayesian way satisfies all these requirements.

8.2.1 Bayesian PID with a single detector

Let $r(s \mid i)$ be a conditional probability density function to observe in some detector a PID signal s if a particle of i-type ($i = e, \mu, \pi, K, p, \mathrm{K}$) is detected. The probability to be a particle of i-type if the signal s is observed, $w(i \mid s)$, depends not only on $r(s \mid i)$, but also on how often this type of particles is registered in the considered

experiment ("a priory" probability C_i to find this kind of particles in the detector). The corresponding relation is given by the Bayes' formula:

$$w(i \mid s) = \frac{r(s \mid i) C_i}{\sum_{k=e,} r(s \mid k) C_k} \qquad (1)$$

Under some reasonable conditions, C_i and $r(s \mid i)$ are not correlated so that one can rely on the following approximation:

- The functions $r(s \mid i)$ reflect only properties of the detector ("detector response functions") and do not depend on other external conditions like event and track selections.
- On contrary, the quantities C_i ("relative concentrations" of particles of i-type) do not depend on the detector properties, but do reflect the external conditions, selections etc}.

The PID procedure is done in the following way. First, the detector response function is obtained. Second, a value $r(s \mid i)$ is assigned to each track. Third, the relative concentrations C_i of particle species are estimated for a subset of events and tracks selected in a specific physics analysis. Finally, an array of probabilities $w(i \mid s)$ is calculated (see Equation 1 for each track within the selected subset.

The probabilities $w(i \mid s)$ are often called PID weights. The conditional probability density function $r(s \mid i)$ (detector response function) can be always parameterized with sufficient precision using available experimental data.

In the simplest approach, the *a-priori* probabilities C_i (relative concentrations of particles of i-type) to observe a particle of i-type can be assumed to be equal.

However, in many cases one can do better. Thus, for example in ALICE, when doing PID in the TPC for the tracks that are registered both in the TPC and in the Time-Of-Flight detector (TOF), these probabilities can be estimated using the measured time-of-flight. One simply fills a histogram of the following quantity:

$$m = \frac{p}{\beta\gamma} = p\sqrt{\frac{c^2 t^2}{l^2} - 1}$$

where p and l are the reconstructed track momentum and length and t is the measured time-of-flight. Such a histogram peaks near the values m that correspond to the masses of particles.

Forcing some of the C_i to be exactly zeros excludes the corresponding particle type from the PID analysis and such particles will be redistributed over other particle classes (see Equation 1). This can be useful for the kinds of analysis when, for the particles of a certain type, one is not concerned by the contamination but, at the same time, the efficiency of PID is of particular importance.

8.2.2 PID combined over several detectors

This method can be easily applied for combining PID measurements from several detectors. Considering the whole system of N contributing detectors as a single "super-detector" one can write the combined PID weights $W(i \mid \bar{s})$ in the form similar to that given by Equation 1.

$$W(i \mid \bar{s}) = \frac{R(\bar{s} \mid i) C_i}{\sum_{k=e,\mu,p,\mathrm{K}} R(\bar{s} \mid k) C_k}$$

where $\bar{s} = s_1, s_2, \mathrm{K}, s_N$ is a vector of PID signals registered in the first, second and other contributing detectors, C_i are the *a priory* probabilities to be a particle of the i-type (the same as in Equation 1) and $R(\bar{s} \mid i)$ is the combined response function of the whole system of detectors.

If the single detector PID measurements s_j are uncorrelated (which is approximately true in the case of the ALICE experiment), the combined response function is product of single response functions $r(s_j \mid i)$ (the ones in Equation 1):

$$R(\bar{s} \mid i) = \prod_{j=1}^{N} r(s_j \mid i) \qquad (2)$$

One obtains the following expression for the PID weights combined over the whole system of detectors:

$$W(i \mid s_1, s_2, \mathrm{K}, s_N) = \frac{C_i \prod_{j=1}^{N} r(s_j \mid i)}{\sum_{k=e,\mu,\pi,\mathrm{K}} C_k \prod_{j=1}^{N} r(s_j \mid k)} \qquad (3)$$

In the program code, the combined response functions $R(\bar{s} \mid i)$ do not necessarily have to be treated as analytical. They can be "procedures" (C++ functions, for example). Also, some additional effects like probabilities to obtain a mis-measurement (mis-matching) in one or several contributing detectors can be accounted for.

The formula Equation 3 has the following useful features:

- If for a certain particle momentum one (or several) of the detectors is not able to identify the particle type (i.e. the $r(s \mid i)$ are equal for all $i = e, \mu, \mathrm{K}$), the contribution of such a detector cancels out from the formula.
- When several detectors are capable of separating the particle types, their contributions are accumulated with proper weights, thus providing an improved combined particle identification.
- Since the single detector response functions $r(s|i)$ can be obtained in advance at the calibration step and the combined response can be approximated by Equation 2, a part of PID (calculation of the $R(\bar{s} \mid i)$) can be done track-by-track "once and forever" by the reconstruction software and the

results can be stored in the Event Summary Data. The final PID decision, being dependent via the *a priory* probabilities C_i on the event and track selections, is then postponed until the physics analysis of the data.

8.2.3 Stability with respect to variations of the *a priory* probabilities

Since the results of this PID procedure explicitly depend on the choice of the *a priori* probabilities C_i (and, in fact, this kind of dependence is unavoidable in any case), the question of stability of the results with respect to the almost arbitrary choice of C_i becomes important.

Fortunately, there is always some momentum region where the single detector response functions for different particle types of at least one of the detectors do not significantly overlap, and so the stability is guaranteed. The more detectors enter the combined PID procedure, the wider this momentum region becomes and the results are more stable.

Detailed simulations using the AliRoot framework show that results of the PID combined over all the ALICE central detectors are, within a few per cent, stable with respect to variations of C_i up-to at least 3 Gev/c.

8.2.4 Features of the Bayesian PID

Particle Identification in ALICE experiment at LHC can be done in a Bayesian way. The procedure consists of three parts:

- First, the single detector PID response functions $r(s \mid i)$ are obtained. The calibration software does this.
- Second, for each reconstructed track the combined PID response $R(\bar{s} \mid i)$ is calculated and effects of possible mis-measurements of the PID signals can be accounted for. The results are written to the Event Summary Data and, later, are used in all kinds of physics analysis of the data. This is a part of the reconstruction software.
- And finally, for each kind of physics analysis, after the corresponding event and track selection is done, the *a priori* probabilities C_i to be a particle of a certain i-type within the selected subset are estimated and the PID weights $W(i \mid \bar{s})$ are calculated by means of formula Equation 3. This part of the PID procedure belongs to the analysis software.

The advantages of the particle identification procedure described here are:

- The fact that, due to different event and rack selection, the PID depends on a particular kind of performed physics analysis is naturally taken into account.
- Capability to combine, in a common way, signals from detectors having quite different nature and shape of the PID response functions (silicon, gas, time-of-flight, transition radiation and Cerenkov detectors).

- No interactive multidimensional graphical cuts are involved. The procedure is fully automatic.

8.3 Vertex estimation using tracks

Each track, reconstructed in the TPC and in the ITS, is approximated with a straight line at the position of the closest approach to the nominal primary vertex position (the nominal vertex position is supposed to be known with a precision of 100–200 μm). Then, all possible track pairs (i, j) are considered and for each pair, the centre $C(i, j) \equiv (x_{ij}, y_{ij}, z_{ij})$ of the segment of minimum approach between the two lines is found. The coordinates of the primary vertex are determined as:

$$x_v = \frac{1}{N_{pairs}} \sum_{i,j} x_{ij}; \quad y_v = \frac{1}{N_{pairs}} \sum_{i,j} y_{ij}; \quad z_v = \frac{1}{N_{pairs}} \sum_{i,j} z_{ij}$$

where N_{pairs} is the number of track pairs. This gives an improved estimate of the vertex position.

Finally, the position $\mathbf{r}_v = (x_v, y_v\ z_v)$ of the vertex is reconstructed minimizing the χ^2 function (see [65]):

$$\chi^2(\mathbf{r}_v) = \sum_i (\mathbf{r}_v - \mathbf{r}_i)^T \mathbf{V}_i^{-1} (\mathbf{r}_v - \mathbf{r}_i)$$

where $\mathbf{r}_i$ is the global position of the track i (i.e. the position assigned at the step above) and $\mathbf{V}_i$ is the covariance matrix of the vector $\mathbf{r}_i$.

In order not to spoil the vertex resolution by including in the fit tracks that do not originate from the primary vertex (e.g. strange particle decay tracks), the tracks giving a contribution larger than some value $\chi^2_{\max}$ to the global χ^2 are removed one-by-one from the sample, until no such tracks are left. The parameter $\chi^2_{\max}$ was tuned, as a function of the event multiplicity, so as to obtain the best vertex resolution.

8.4 Alignment framework

8.4.1 Basic objects and alignment constants

The purpose of the ALICE alignment framework is to offer all the functionality related to storing alignment information, retrieving it from the Offline Conditions Data Base (OCDB) and consistently applying it to the ALICE geometry in order to improve the knowledge of the real geometry by means of the additional information obtained by survey and alignment procedures, without needing to change the hard-coded implementation of the detector's geometry. The ALICE alignment framework is based on the **AliAlignObj** base class and its derived classes; each instance of this class is an *alignment object* storing the so called *alignment constants* for a single alignable volume, that is the information to uniquely identify the physical volume (specific instance of the volume in the geometry tree) to be displaced and to unambiguously describe the delta-

transformation to be applied to that volume. In the ALICE alignment framework an alignment object holds the following information:

- a unique volume identifier
- a unique global index
- a delta-transformation

In the following we describe the meaning of these variables, how they are stored and set and the functionality related to them.

8.4.1.1The unique volume identifier

The unique volume identifier is the character string which allows the user to access a specific physical volume inside the geometry tree. For the ALICE geometry (which is a ROOT geometry) this is the *volume path* that is the string containing the names of all physical volumes in the current branch in the directory tree fashion. For example "/A_1/B_i/.../M_j/Vol_k" identifies the physical volume "kth copy of the volume Vol" by listing its container volumes; going from right to left in the path corresponds to going from the innermost to the outermost containers and from the lowest to the upper level in the geometry tree, starting from the mother volume "M_j" of the current volume "Vol_k" up to the physical top volume "A_1", the root of the geometry tree.

The unique volume identifier stored by the alignment object is not the volume path but a *symbolic volume name*, a string dynamically associated to the corresponding volume path by a hash table built at the finalization stage of the geometry (the physical tree needs to be already closed) and stored as part of it. The choice of the symbolic volume names is constrained only by the following two rules:

1. Each name has to contain a leading sub-string indicating its pertaining sub-detector; in this way the uniqueness of the name inside the sub-detector scope guarantees also its uniqueness in the global scope of the whole geometry.
2. Each name has to contain the intermediate alignable levels, separated by a slash ("/"), in case some other physical volume on the same geometry branch is in turn alignable.

There are two considerable advantages deriving from the choice to introduce the symbolic volume names as unique volume identifiers stored in the alignment object in place of the volume path:

1. The unique volume identifier has no direct dependency on the geometry; in fact changes in the volume paths reflect in changes in the hash table associating the symbolic names to them, which is built and stored together with the geometry. As a consequence the validity of the alignment objects is not affected by changes in the geometry and hence is in principle unlimited in time.
2. The unique volume identifier can be freely chosen, according to the two simple rules mentioned above, thus allowing to assign meaningful names to the alignable volumes, as opposed to the volume paths which inevitably are long strings of often obscure names.

The geometry then provides the user with some methods to query the hash table linking the symbolic volume names to the corresponding volume paths; in particular the user can

- get the number of entries in the table;
- retrieve a specific entry (symbolic volume name, volume path) either by index or by symbolic name.

8.4.1.2The unique global index

Among the alignment constants we store a numerical index uniquely identifying the volume to which those constants refer; being a “short”, this numerical index has 16 bits available which are filled from the index of the “layer” or sub-detector to which the volume belongs (5 bits) and from the “local index”, i.e. the index of the volume itself inside the sub-detector (the remaining 11 bits). Limiting the range of sub-detectors to 2^5=32 and of alignable volumes inside each sub-detector to 2^{11}=2048, this suites our needs.

The aim of indexing the alignable volumes is to have a fast iterative access during alignment procedures. The framework allows to easily browse through the look-up table mapping indexes to symbolic volume names by means of methods which return the symbolic volume name for the present object given either its global index or both, its layer and local indexes. For these methods to work, the only condition is that at least one instance of an alignment object has been created, so that the static method building the look-up table has been called.

8.4.1.3The delta-transformation

The delta-transformation is the transformation that defines the displacement to be applied to the given physical volume. During the alignment process we want to correct the hard-coded, ideal position of some volume, initially fixed according to the engineers' drawings, by including the survey and alignment information related to those volumes; we say that we want to align the ideal geometry. With this aim, we need here to describe how the delta-transformations are defined and thus how they have to be produced and applied to the ideal geometry in order to correct the global and local ideal transformations into global and local aligned transformations.

For the representation of the delta-transformation there are several possible conventions and choices, in particular:

1. to use the local-to-global or the global-to-local convention and “active-” or “passive-transformations” convention;
2. to use the local or global delta-transformation to be stored in the alignment object and to be passed when setting the object itself;
3. the convention used for the Euler angles representing the delta-transformation;
4. the use of a matrix or of a minimal set of parameters (three orthogonal shifts plus three Euler angles) to be stored in the alignment object and to be passed when setting the object itself.

The choices adopted by the framework are explained in the remainder of this section.

Use of the global and local transformations

Based on the ROOT geometry package, the framework keeps the “local-to-global” convention; this means that the *global transformation* for a given volume is the matrix $\mathbf{G}$ that, as in TGeo, transforms the local vector $\mathbf{l}$ (giving the position in the local reference system, i.e. the reference system associated to that volume) into the global vector $\mathbf{g}$, giving the position in the global (or master) reference system (“MARS”), according to:

$$\mathbf{g} = \mathbf{G} \cdot \mathbf{l}$$

Similarly, the *local transformation* matrix is the matrix $\mathbf{L}$ that transforms a local vector $\mathbf{l}$ into the corresponding vector in the mother volume RS, $\mathbf{m}$, according to:

$$\mathbf{m} = \mathbf{L} \cdot \mathbf{l}$$

If furthermore $\mathbf{M}$ is the global transformation for the mother volume, then we can write:

$$\mathbf{g} = \mathbf{G} \cdot \mathbf{l} = \mathbf{M} \cdot \mathbf{m} = \mathbf{M} \cdot \mathbf{L} \cdot \mathbf{l} \qquad (10)$$

Recursively repeating this argument to all the parent volumes, that is to all the volumes in the branch of the geometry tree which contains the given volume, we can write:

$$\mathbf{g} = \mathbf{G} \cdot \mathbf{l} = \mathbf{M}_0 \cdot \mathrm{K} \cdot \mathbf{M}_n \cdot \mathbf{L} \cdot \mathbf{l} \qquad (11)$$

which shows that the global matrix is given by the product of the matrices of the parent volumes on the geometry branch, from the uppermost to the lowest level.

Let's now denote by $\mathbf{G}$ and $ $\mathbf{L}$ the ideal global and local transformations of a specific physical volume (those relative to the reference geometry) and let's put the superscript "a" to the corresponding matrices in the aligned geometry, so that $\mathbf{G}^a$ and $\mathbf{L}^a$ are the aligned global and aligned local transformations which relate the position of a point in the local RS to its position in the global RS and in the mother's RS respectively, after the volume has been aligned, according to:

$$\begin{aligned} \mathbf{g} &= \mathbf{G}^a \cdot \mathbf{l} \\ \mathbf{m} &= \mathbf{L}^a \cdot \mathbf{l} \end{aligned} \qquad (12)$$

Equations 12 are the equivalent of Equations 10 and 11 after the volume has been displaced.

There are two possible choices for expressing the delta-transformation:

- Use of the *global delta-transformation* Δ^g, that is the transformation to be applied to the ideal global transformation $\mathbf{G}$ in order to get the aligned global transformation:

$$\mathbf{G}^g = \Delta^g \cdot \mathbf{G} = \Delta^g \cdot \mathbf{M} \cdot \mathbf{L} \qquad (13)$$

- Use of the *local delta-transformation* Δ^l, that is the transformation to be applied to the ideal local transformation $\mathbf{L}$ to get the aligned local transformation:

$$\mathbf{L}^a = \mathbf{L} \cdot \Delta^l \qquad (14)$$

Equations 13 and 14 allow rewriting:

$$\mathbf{G}^a = \mathbf{M} \cdot \mathbf{L}^a \tag{15}$$

as:

$$\Delta^g \cdot \mathbf{M} \cdot \mathbf{L} = \mathbf{M} \cdot \mathbf{L} \cdot \Delta^l \tag{16}$$

or equivalently:

$$\begin{aligned} \Delta^g &= \mathbf{G} \cdot \Delta^l \cdot \mathbf{G}^{-1} \\ \Delta^l &= \mathbf{G}^{-1} \cdot \Delta^g \cdot \mathbf{G} \end{aligned} \tag{17}$$

to relate global and local alignment.

The alignment object stores as delta-transformation the global delta-transformation; nevertheless both global and local delta-transformations can be used to construct the alignment object or to set it. The reasons for this flexibility in the user interface is that the local RS is sometimes the most natural one for expressing the misalignment, as e.g. in the case of a volume rotated around its centre; however the use of the local delta-transformation is sometimes error-prone; in fact the user has to be aware that he is referring to the same local RS which is defined in the hard-coded geometry when positioning the given volume, while the local RS used by simulation or reconstruction code can in general be different. In case the alignment object is constructed or its delta-transformation is set by means of the local delta-transformation, the framework will then use Equation 17 to perform the conversion into global alignment constants.

As for the choice of storing a symbolic volume name instead of the volume path as volume identifier, we would like to also make the delta-transformation stored in the alignment objects independent from the geometry, keeping thus their validity unconstrained. This is possible if we store in the geometry itself a matrix for the ideal global transformation related to that volume (this possibility is offered by the class storing the link between symbolic volume names and volume paths, see Section 8.4.2.

Matrix or parameters for the delta-transformation

The global delta-transformation can be saved both

- as a **TGeoMatrix** and
- as a set of six parameters, out of which three define the translation, by means of the shifts in the three orthogonal directions, and three define the rotation by means of three Euler angles.

These two cases correspond to choosing one of the following two **AliAlignObj**-derived classes:

- **AliAlignObjMatrix**: stores a **TGeoHMatrix**
- **AliAlignObjAngles**: stores six double precision floating point numbers;

While storing the alignment constants in a different form, they appear with the same user interface, which allows setting the delta-transformation both via the matrix and via the six parameters that identify it.

Choice for the Euler angles

A general rotation in three-dimensional Euclidean space can be decomposed into and

represented by three successive rotations about the three orthogonal axes. The three angles characterizing the three rotations are called Euler angles; however there are several conventions for the Euler angles, depending on the axes about which the rotations are carried out, right/left-handed systems, (counter-)clockwise direction of rotation, order of the three rotations.

The convention chosen in the ALICE alignment framework for the Euler angles is the *xyz convention* (see [66]), also known as *pitch-roll-yaw* or *Tait-Bryan angles,* or *Cardano angles* convention. Following this convention, the general rotation is represented as a composition of a rotation around the *z*-axis (yaw) with a rotation around the *y*-axis (pitch) with a rotation around the *x*-axis (roll). There is an additional choice to fully specify the convention used, since the angles have opposite sign whether we consider them bringing the original RS in coincidence with the aligned RS (*active-transformation* convention) or the other way round (*passive-transformation* convention). In order to maintain our representation fully consistent with the **TGeoRotation** methods we choose the *active-transformation* convention, that is the opposite convention as the one chosen by the already referenced description of the pitch-roll-yaw angles [199].

To summarise, the three angles – ψ, ϑ, ϕ – used by the framework to represent the rotation part of the delta-transformation, unambiguously represent a rotation $\mathbf{A}$ as the composition of the following three rotations:

1. a rotation $\mathbf{D}$ by an angle ϕ (yaw) around the *z*-axis

$$\mathbf{D} = \begin{pmatrix} \cos\phi & -\sin\phi & 0 \\ \sin\phi & \cos\phi & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

2. a rotation $\mathbf{C}$ by an angle ϑ (pitch) around the *y*-axis

$$\mathbf{C} = \begin{pmatrix} \cos\vartheta & 0 & \sin\vartheta \\ 0 & 1 & 0 \\ -\sin\vartheta & 0 & \cos\vartheta \end{pmatrix}$$

3. a rotation $\mathbf{B}$ by an angle ψ (roll) around the *x*-axis

$$\mathbf{B} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & \cos\psi & -\sin\psi \\ 0 & \sin\psi & \cos\psi \end{pmatrix}$$

which leads to:

$$\mathbf{A} = \mathbf{B}\cdot\mathbf{C}\cdot\mathbf{D} = \begin{pmatrix} \cos\vartheta\cos\phi & -\cos\vartheta\sin\phi & \sin\vartheta \\ \sin\psi\sin\vartheta\cos\phi + \cos\psi\sin\phi & -\sin\psi\sin\vartheta\sin\phi + \cos\psi\cos\phi & -\cos\vartheta\sin\psi \\ -\cos\psi\sin\vartheta\cos\phi + \sin\psi\sin\phi & \cos\psi\sin\vartheta\sin\phi + \sin\psi\cos\phi & \cos\vartheta\cos\psi \end{pmatrix}$$

8.4.2 Use of ROOT geometry functionality

The ALICE geometry is implemented via the ROOT geometrical modeller (often referred to as **TGeo**), a framework for building, browsing, navigating and visualising a detector's

geometry, which is independent from the Monte Carlo transport (see [67] and the dedicated chapter in [68]). This choice allows the ALICE alignment framework to take advantage of using ROOT features such as its I/O, histogramming, browsing, GUI, ... However, the main advantage of this choice is that the ALICE alignment framework can provide its specific functionality as a rather thin layer on top of already existing features which allow to consistently and efficiently manage the complexity related to modifying a tree of some million of physical nodes. The ALICE alignment framework takes in particular advantage of the possibility:

- to save the geometry to a file and upload it from a file;
- to check the geometry for overlaps and extrusions exceeding a given threshold;
- to query the geometry for the global and local matrix of a given physical volume;
- to make a physical node out of a specific physical volume and change the local and global transformation associated to it, while keeping track of the original transformations;
- to store a hash table of links between symbolic volume names and volume paths which can be queried in an efficient way.

Concerning this last issue, the class representing the objects linking the symbolic volume names and the volume paths provides in addition the possibility to store a transformation. This feature turns out to be very useful if it is used to store the matrix relating the RS stored in the geometry (global transformation matrix for that volume) with the RS used in simulation and reconstruction (the two things in general differ).

8.4.3 Application of the alignment objects to the geometry

The base class provides a method to apply the single alignment object to the geometry present in memory, loaded from file or constructed; the method accesses the geometry to change the position of the volume referred by the unique volume identifier according to Equation (13). However this method alone cannot guarantee that the single object is applied correctly; the most common case is indeed the application of a set of alignment objects. In this case the framework has to check that the application of each object in the set does not invalidate the application of the others; when applying a set of alignment objects during a simulation or reconstruction run the framework transparently performs the following two checks:

1. In case of alignment objects referring to physical volumes on the same branch, they have to be applied starting from the one which refers to a volume at the uppermost level in the physical tree (container volume) down to the one at the lowest level (contained volume). On the contrary, if the contained volume is displaced first the subsequent displacement of the container volume would change its temporarily correct position;
2. In no case, two alignment objects should be applied to the same physical volume separately.

The reason for the first limitation is, in short, that the position of the contained volumes

depend on the position of the container volumes. The reason for the second limitation is that the delta-transformations are relative to the ideal global position of the given volume (see Equation (13)), which then need not to have been previously modified by the previous application of an alignment object referring to the same volume. The tools used by the framework for checking that the two previous conditions are fulfilled are respectively:

1. Sorting the alignment objects based on a method which compares the depth of the physical volume to which the given alignment object refers.
2. Combining more alignment objects referring to the same volume before applying them to the geometry.

During a simulation or reconstruction run the user can consistently apply the objects to the geometry, having the two checks described above transparently performed.

An additional check is performed during a simulation or reconstruction run to verify that the application of the alignment objects did not introduce big overlaps or extrusions which would invalidate the geometry (hiding some sensitive parts or changing the material budget during tracking). This check is done by means of the overlap checker provided by the ROOT geometry package; a default threshold below which overlaps and extrusions are accepted is fixed; the **TGeo** overlap checker favours speed (checks the whole ALICE geometry in few seconds) at the expense of completeness, thus same rare overlap topologies can eventually escape the check.

8.4.4 Access to the Conditions Data Base

An important task of the ALICE alignment framework is to intermediate between the simulation and reconstruction jobs and the objects residing on the Offline Conditions Data Base (OCDB), both for defining a default behaviour and for managing specific use cases. The OCDB is filled with conditions (calibration and alignment) objects; the alignment objects in the OCDB are presently created by macros to reproduce two possible misalignment scenarios: the initial misalignment, according to expected deviations from the ideal geometry just after the sub-detectors are positioned and the residual misalignment, trying to reproduce the deviations which can not be resolved by the alignment procedures. The next step is to fill the OCDB with the alignment objects produced from the survey procedures, as soon as survey data are available to the offline. Finally these objects and those produced by alignment procedures will fill the OCDB to be used by the reconstruction of the real data in its different passes.

The OCDB stores the conditions making use of the database capabilities of a file system three-level directory structure; the run and the version are stored in the file name. If not otherwise specified, the OCDB returns the last version of the required object and in case of an object being uploaded it is automatically saved with increased version number.

The ALICE alignment framework defines a specific default storage from which to load the alignment objects for all the sub-detectors; the user can set a different storage, either residing locally or on the Grid if he has the permissions to access it. The definition of a non-default storage for the OCDB, as well as its deactivation can also be given for

specific sub-detectors only, The user can also just switch off the loading of alignment objects from a OCDB storage or as a side-effect of passing to the simulation or reconstruction run an array of alignment objects available in memory.

8.4.5 Summary

The ALICE alignment framework, based on the ROOT geometry package (see [199, 199]), aims at allowing a consistent and flexible management of the alignment information, while leaving the related complexity as much as possible hidden to the user. The framework allows:

- Saving and retrieving the alignment constants relative to a specific alignable volume (automatic retrieval from a Conditions Data Base is handled);
- Apply the alignment objects to the current (ideal) geometry;
- Get from the current geometry the alignment object for a specified alignable volume;
- Transform positions in the ideal global RS into positions in the aligned global RS;
- Set the objects by means of both global and local delta-transformations.

These functionalities are built on the **AliAlignObj** base class and its two derived classes, which store the delta-transformation by means of the transformation matrix (**AliAlignObjMatrix**) or by means of the six transformation parameters (**AliAlignObjAngles**). The user interface is the same in both cases; it fixes the representation of the delta-transformation while leaving several choices to the user which have been explained in this note together with their implementation.

The ALICE alignment framework fixes the following conventions:

- The transformations are interpreted according to the local-to-global convention;
- The delta-transformation stored is the global delta-transformation;
- The three parameters to specify the rotation are the roll-pitch-yaw Euler angles, with the *active-transformations* convention.

The framework fixes also the following default behaviours in simulation and reconstruction runs:

- Objects are loaded from a default Conditions Data Base storage, on a sub-detector basis;
- The set of loaded objects is sorted for assuring the consistency of its application to the geometry;
- The ideal and aligned geometries are saved.

Several choices related to the delta-transformation are left to the user, who:

- Can choose to set the alignment object either by passing a **TGeoMatrix** or by giving the six parameters which uniquely identify the global delta-transformation;

- Can choose if he wants the object to store either the **TGeoMatrix**, using an **AliAlignObjMatrix** or the six parameters, using an **AliAlignObjAngles**;
- Can choose if the transformation he is passing is the global delta-transformation or the local delta-transformation; in this latter case the framework converts it to the global one to set the internal data members.

9 Glossary

ADC Analogue to Digital Conversion/Converter

AFS Andrew File System
http://en.wikipedia.org/wiki/Andrew_file_system

ALICE A Large Ion Collider Experiment
http://aliceinfo.cern.ch

AOD Analysis Object Data

API Application Program Interface

ARDA Architectural Roadmap towards Distributed Analysis
http://lcg.web.cern.ch/LCG/activities/arda/arda.html

AliRoot ALICE offline framework
http://aliceinfo.cern.ch/offline

CA Certification Authority

CASTOR CERN Advanced STORage
http://castor.web.cern.ch/castor

CDC Computing Data Challenge

CDF Collider Detector at Fermilab

CE Computing Element
http://aliceinfo.cern.ch/static/AliEn/AliEn_Instalation/ch06s07.html

CERN European Organization for Nuclear Research
http://www.cern.ch

CINT C/C++ INTerpreter that is embedded in ROOT
http://root.cern.ch/root/Cint.html

CRT Cosmic Ray Trigger, the official name is ACORDE
url??

CVS Concurrent Versioning System
http://www.nongnu.org/cvs

DAQ Data AcQuisition system
http://cern.ch/alice-daq

DATE Data Acquisition and Test Environment
http://cern.ch/alice-daq

DCA Distance of Closest Approach

DCS Detector Control System
http://alicedcs.web.cern.ch/alicedcs

DPMJET Dual Parton Model monte carlo event generator
http://sroesler.web.cern.ch/sroesler/dpmjet3.html

EGEE Enabling Grid for E-sciencE project
http://public.eu-egee.org

EMCal Electromagnetic Calorimeter

ESD Event Summary Data

FLUKA A fully integrated particle physics Monte Carlo simulation package
http://www.fluka.org

FMD Forward Multiplicity Detector
http://fmd.nbi.dk

FSI Final State Interactions

GAG Grid Application Group
http://project-lcg-gag.web.cern.ch/project-lcg-gag

GUI Graphical User Interface

GeVSim fast Monte Carlo event generator, base on MEVSIM

Geant 4 A toolkit for simulation of the passage of particles through matter
http://geant4.web.cern.ch/geant4

HBT Hanbury Brown and Twiss

HEP High Energy Physics

HEPCAL HEP Common Application Area

HERWIG Monte Carlo package for simulating Hadron Emission Reactions With Interfering Gluons
http://cernlib.web.cern.ch/cernlib/mc/herwig.html

HIJING Heavy Ion Jet Interaction Generator

HLT High Level Trigger
http://wiki.kip.uni-heidelberg.de/ti/HLT/index.php/Main_Page

HMPID High Momentum Particle Identification
http://alice-hmpid.web.cern.ch/alice-hmpid

ICARUS Imaging Cosmic And Rare Underground Signals
http://pcnometh4.cern.ch

IP Internet Protocol

ITS Inner Tracking System; collective name for SSD, SPD and SDD

JETAN JET ANalysis module

LCG LHC Computing Grid
http://lcg.web.cern.ch/LCG

LDAP Lightweight Directory Access Protocol

LHC Large Hadron Collider
http://lhc.web.cern.ch/lhc

LSF Load Sharing Facility
http://wwwpdp.web.cern.ch/wwwpdp/bis/services/lsf

MC Monte Carlo

MoU Memorandum of Understanding

OCDB Offline Calibration DataBase
http://aliceinfo.cern.ch/Offline/Activities/ConditionDB.html

OO Object Oriented

OS Operating System

PAW Physics Analysis Workstation
http://paw.web.cern.ch/paw

PDC Physics Data Challenge

PDF Particle Distribution Function

PEB Project Execution Board

PHOS PHOton Spectrometer

PID Particle IDentity/IDentification

PMD Photon Multiplicity Detector
http://www.veccal.ernet.in/~pmd/ALICE/alice.html

PPR Physics Performace Report
http://alice.web.cern.ch/Alice/ppr

PROOF Parallel ROOT Facility
http://root.cern.ch/root/doc/RootDoc.html

PWG Physics Working Group
http://aliceinfo.cern.ch/Collaboration/PhysicsWorkingGroups

PYTHIA event generator

QA Quality Assurance

QCD Quantum ChromoDynamics

QS Quantum Statistics

RICH Ring Imaging CHerenkov
http://alice-hmpid.web.cern.ch/alice-hmpid

ROOT A class library for data analysis
http://root.cern.ch

RTAG Requirements and Technical Assessment Group

SDD Silicon Drift Detector

SDTY Standard Data Taking Year

SE Storage Element

SI2k SpecInt2000 CPU benchmark
http://cerncourier.com/articles/cnl/1/11/9/1

SLC Scientific Linux CERN
http://linuxsoft.cern.ch

SOA Second Order Acronym

SPD Silicon Pixel Detector
http://www.pd.infn.it/spd

SSD Silicon Strip Detector

TDR Technical Design Report
http://alice.web.cern.ch/Alice/TDR

TOF Time Of Flight Detector
http://alice.web.cern.ch/Alice/Projects/TOF

TPC Time Projection Chamber
http://alice.web.cern.ch/Alice/Projects/TPC

TRD Transition Radiation Detector
http://www-alice.gsi.de/trd/index.html

UI User Interface

UID Unique IDentification number

URL Universal Resource Locator

VMC Virtual Monte Carlo

VO Virtual Organization

VOMS Virtual Organization Membership Service

WAN Wide Area Network

XML Extensible Markup Language
http://www.w3.org/XML

ZDC Zero Degree Calorimeter

10References

1 CERN/LHCC 2003-049, ALICE Physics Performance Report, Volume 1 (7 November 2003);
ALICE Collaboration: F. Carminati *et al.* J. Phys. G: Nucl. Part. Phys. **30** (2004) 1517–1763.

2 CERN-LHCC-2005-018, ALICE Technical Design Report: Computing, ALICE TDR 012 (15 June 2005).

3 http://root.cern.ch

4 http://wwwasdoc.web.cern.ch/wwwasdoc/geant_html3/geantall.html

5 http://www.fluka.org

6 http://cewrn.ch/geant4

7 H.-U. Bengtsson and T. Sjostrand, Comput. Phys. Commun. **46** (1987) 43;
T. Sjostrand, Comput. Phys. Commun. **82** (1994) 74;
the code can be found in
http://www.thep.lu.se/~torbjorn/Pythia.html

8 X. N. Wang and M. Gyulassy, Phys. Rev. **D44** (1991) 3501.
M. Gyulassy and X. N. Wang, Comput. Phys. Commun. **83** (1994) 307-331.
The code can be found in
http://www-nsdth.lbl.gov/~xnwang/hijing

9 http://alien.cern.ch
P. Saiz *et al.*, Nucl. Instrum. Meth. A **502** 2003 437–440

10 http://www.linux.org

11 http://linux.web.cern.ch/linux

12 http://www.redhat.com

13 http://fedora.redhat.com

14 http://gcc.gnu.org

15 http://www.intel.com/cd/software/products/asmo-na/eng/compilers/index.htm

16 http://www.intel.com/cd/software/products/asmo-na/eng/vtune/index.htm

17 http://www.intel.com/products/processor/itanium2/index.htm

18 http://www.amd.com

19 http://ximbiot.com/cvs/manual

20 http://cern.ch/clhep

21 http://cern.ch/castor

22 J. Ranft, Phys. Rev. **D 51** (1995) 64.

23 ALICE-INT-2003-036

24 A.~Morsch, http://home.cern.ch/~morsch/AliGenerator/AliGenerator.html and http://home.cern.ch/~morsch/generator.html

25 B. Andersson, *et al.*, Phys. Rep. **97** (1983) 31.

26 B. Andersson, *et al.*, Nucl. Phys. **B281** (1987) 289;

B. Nilsson-Almqvist and E. Stenlund, Comput. Phys. Commun. **43** (1987) 387.

27 A. Capella, *et al.*, Phys. Rep. **236** (1994) 227.

28 http://arxiv.org/abs/hep-ph/0312045.

29 HERWIG 6.5, G. Corcella, I.G. Knowles, G. Marchesini, S. Moretti, K. Odagiri, P. Richardson, M.H. Seymour and B.R. Webber, JHEP 0101 (2001) 010 [hep-ph/0011363]; hep-ph/0210213

30 L. Ray and R.S. Longacre, STAR Note 419.

31 S. Radomski and Y. Foka, ALICE Internal Note 2002-31.

32 http://home.cern.ch/~radomski

33 L. Ray and G.W. Hoffmann. Phys. Rev. **C 54**, (1996) 2582, Phys. Rev. **C 60**, (1999) 014906.

34 P. K. Skowrónski, ALICE HBT Web Page, http://aliweb.cern.ch/people/skowron

35 A.M. Poskanzer and S.A. Voloshin, Phys. Rev. **C 58**, (1998) 1671.

36 A. Alscher, K. Hencken, D. Trautmann, and G. Baur. Phys. Rev. A **55**, (1997) 396.

37 K. Hencken, Y. Kharlov, and S. Sadovsky, ALICE Internal Note 2002-27.

38 http://root.cern.ch/root/doc/RootDoc.html

39 L. Betev, ALICE-PR-2003-279

40 CERN/LHCC 2005-049, ALICE Physics Performance Report, Volume 2 (5 December 2005);

41 P. Billoir; NIM **A225** (1984) 352,
P. Billoir *et al.*, NIM **A241** (1985) 115,
R. Fruhwirth, NIM **A262** (1987) 444,
P. Billoir; **CPC** (1989) 390.

42 http://alien.cern.ch/download/current/gClient/gShell_Documentation.html

43 http://glite.web.cern.ch/glite

44 http://root.cern.ch/root/PROOF.html

45 CERN/LHCC 99-12.

46 CERN/LHCC 2000-001.

47 P.Skowrónski, PhD Thesis.

48 http://indico.cern.ch/conferenceDisplay.py?confId=a055286

49 P. Christakoglou, P. Hristov, ALICE-INT-2006-023

50 http://www.star.bnl.gov

51 A. Shoshani, A. Sim, and J. Wu, “Storage resource managers: Middleware components for Grid storage”, in Proceedings of Nineteenth IEEE Symposium on Mass Storage Systems, 2002 (MSS 2002).

52 K. Wu *et al.*, “Grid collector: An event catalog with automated file management”.

53 http://agenda.cern.ch/fullAgenda.php?ida=a055638

54 http://aliceinfo.cern.ch/offline

55 http://pcaliweb02.cern.ch/Offline/Analysis/RunEventTagSystem

56 http://root.cern.ch/root/htmldoc//TGridResult.html

57 {RefAnalysisFramework}http://pcaliweb02.cern.ch/Offline/Analysis/CAF

58 http://agenda.cern.ch/askArchive.php?base=agenda\&categ=a045061\&id=a045061s0t5/transparencies
http://project-arda-dev.web.cern.ch/project-arda-dev/alice/apiservice/AA-UserGuide-0.0m.pdf

59 {RefFileCatalogMetadataNote}M. Oldenburg, ALICE internal note --- to be submitted to EDMS
http://cern.ch/Oldenburg/MetaData/MetaData.doc

60 {RefEventTagNote}P. Christakoglou and P. Hristov, "The Event Tag System", ALICE-INT-2006-023.

61 {RefFileCatalogMetadataWeb} http://pcaliweb02.cern.ch/Offline/Analysis/RunEventTagSystem/\\RunTags.html\#Run/File\%20metadata

62 {RefAlienTutorial}http://pcaliweb02.cern.ch/Offline/Analysis/Tutorial
http://indico.cern.ch/conferenceDisplay.py?confId=8546

63 {RefEventTagWeb} http://pcaliweb02.cern.ch/Offline/Analysis/RunEventTagSystem/\\EventTagsAnalysis.html\#Analysis\%20with\%20tags

64 {Note:RefGSHELL}http://project-arda-dev.web.cern.ch/project-arda-dev/alice/apiservice/AA-UserGuide-0.0m.pdf

65 V. Karimäki, CMS Note 1997/051 (1997).

66 http://mathworld.wolfram.com/EulerAngles.html

67 R. Brun, A. Gheata and M. Gheata, The ROOT geometry package, NIM **A502** (2003) 676-680

68 ROOT User's Guide, http://root.cern.ch/root/doc/RootDoc.html