The Unified Database

User's Guide

K. V. Gertsenberger

MPD collaboration

1 February 2016

TABLE OF CONTENTS

Page #

1.0 GENERAL INFORMATION 3
1.1 DATABASE PURPOSE 3
1.2 POINTS OF CONTACT 4
1.3 ORGANIZATION OF THE MANUAL 4
1.4 ACRONYMS AND ABBREVIATIONS 4

2.0 DATABASE SUMMARY 5
2.1 DATABASE SCHEME 5
2.2 DATABASE IMPLEMENTATION. 7

3.0 GETTING STARTED 9
3.1 ACCESS LEVELS 9
3.2 TEST CONNECTION 9

4.0 C++ DATABASE INTERFACE 10
4.1 THE STRUCTURE OF C++ INTERFACE 10
4.2 USING C++ DATABASE INTERFACE 13
4.2.1 Reading data objects from the database 13
4.2.2 Getting and writing object attributes to the database of the experiment 14
4.2.3 Searching (selection) runs and detector parameter values by attributes 17
4.2.4 Auxiliary functions 19
4.2.5 Writing a new data (data object) of the experiment to the database 19
4.2.6 Deleting data of the experiment from the database 20
4.3 READING ELECTRONICS PARAMETERS FROM TANGO (SLOW CONTROL) DATABASE 21
4.4 THE WEB-INTERFACE OF THE UNIFIED DATABASE 22

APPENDIX A. C++/ROOT EXAMPLES 23
1. CREATING NEW PARAMETERS. 23
2. WRITING NEW DETECTOR PARAMETER'S VALUES. 23
3. READING DETECTOR PARAMETER'S VALUES. 24
4. WORKING WITH DETECTOR GEOMETRY. 27
5. RUN SELECTION ACCORDING TO THE SPECIFIC CONDITIONS. 28

1.0 GENERAL INFORMATION

1.0 GENERAL INFORMATION

The Unified Database is designed as comprehensive relational data storage for offline data analysis in the high-energy physics experiments. It's developed primarily for the fixed target experiment BM@N and MPD experiment of the NICA project. The use of the Unified Database implemented on the PostgreSQL database management system (DBMS) allows one to provide user access to the actual information of the experiment. The C++ interface is developed for the access to the database from C++/ROOT environment and will be presented in the paper. Also Web-interface is developed and will be available on the web-page of the BM@N experiment in the near future.

1.1 Database Purpose

If you use file storing approach for experimental data, you can encounter with the following main problems:

- The usage of multiple files and arbitrary formats (binary, xml, html, excel, text), duplication of information in different files lead to data isolation, redundancy and inconsistency.
- The data of the experiment can be distributed between many subdivisions.
- Sequential, non-indexed file access and search are not efficient.
- No mechanism exists for relating data between these files.
- It is difficult to access and manipulate the data: one needs some dedicated programs.
- Uncontrolled concurrent access by multiple users also often leads to inconsistencies.

The use of databases is a prerequisite for qualitative management and unified access to the data of modern high-energy physics experiments. The Unified Database designed as central data storage offers solutions to the problems above and provides unified access and data management for all collaboration members, correct multi-user data processing, ensuring the actuality of the information being accessed (run parameters, detector geometries and positions, technical and calibration data, etc.), data consistency and integrity, excluding the multiple duplication and use of outdated data. Furthermore it provides automatic backup of the stored data to ensure that data of the experiment will not be lost due to software or hardware failures.

The parameter data being stored in the Unified Database can be classified into 4 groups:

1. Configuration data is concerned with the detector running mode, i.e. voltage settings as well as some programmable parameters for frontends electronics.
2. Calibration data describes the calibration and the alignment of the different subdetectors. Usually quantities are evaluated by running dedicated offline algorithms.

3. Parameter data presents the state of detector subsystems. They include a variety of detector settings including the geometry and material definitions.
4. Algorithm data is used to control the way algorithms operate. It includes, for example, cuts for selection and production paths of files.

1.2 Points of Contact

Dr. Gertsenberger Konstantin Viktorovich
Email: gertsen@jinr.ru

MPD Collaboration (mpd.jinr.ru)
Laboratory of High Energy Physics
Joint Institute for Nuclear Research, Dubna

1.3 Organization of the Manual

1.0: General Information

This section contains general information about the Unified Database, including the database purpose, points of contact, and acronyms and abbreviations

2.0: Database Summary

This section is an overview of the structure and functionality the Unified Database.

3.0: Getting Started

This section contains the information to get started in using the Unified Database, which includes setting your credentials and testing the database connection.

4.0: C++ Database Interface

This section contains the information about using the Unified Database, going into detail all the possible functions, such as reading and writing data for offline analysis in the experiment.

Appendix A: C++/ROOT examples

This appendix contains examples of using the C++ database interface from ‘examples’ directory.

1.4 Acronyms and Abbreviations

DBMS (Database Management System) – software application that interacts with the user, other applications, and the database itself to capture and analyze data.

ROOT – modular scientific software framework, born at CERN, to deal with data processing, statistical analysis, visualization and storage in the researches on high-energy physics.

2.0 DATABASE SUMMARY

2.1 Database Scheme

Entity-relationship diagram of the Unified Database is shown in Figure 1.

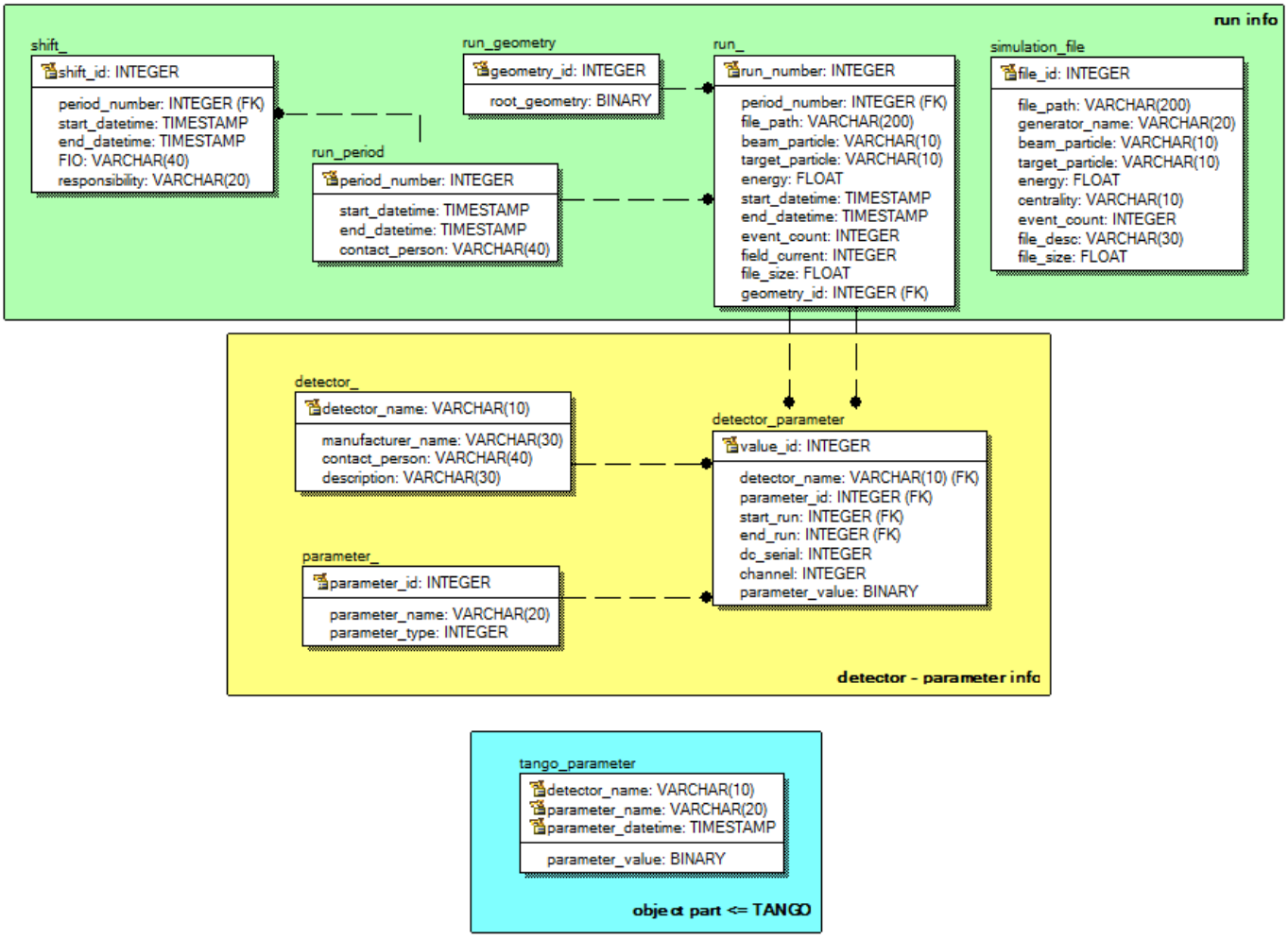

Figure 1. ER-diagram of the Unified Database

The database includes the following tables: run periods, runs, shifts, detector geometries, detectors and its parameters, parameter values, and simulation files. The tables contain the attributes described below:

'run_period' table (common information about run periods of the experiment):

1. period_number: run period number, required;
2. start_datetime: start date and time of the run period, required;
3. end_datetime: end date and time of the run period, optional;
4. contact_person: full name of person responsible for the run period, optional.

‘run_’ table (information about runs of the experiment):

1. run_number: run number, required;
2. period_number: number of the run period including this run, optional;
3. file_path: path to the file with raw data of the run, required;
4. beam_particle: beam particle of the run, required;
5. target_particle: type of target or second particle in the collision, optional;
6. energy: collision energy in GeV, optional;
7. start_datetime: start date and time of the run, required;
8. end_datetime: end date and time of the run, optional;
9. event_count: event count in the run, optional;
10. field_current: current supplied to the magnet in A, optional;
11. file_size: size of the raw file in MB, optional;
12. geometry_id: id of ‘run_geometry’ record with detector geometry of the run, optional.

‘run_geometry’ table (detector geometry in the ROOT format):

1. geometry_id: identifier of the detector geometry, auto increment;
2. root_geometry: binary data containing detector geometry in *.root file format, required.

‘detector_’ table (information about detectors of the experiment):

1. detector_name: name of the detector, required;
2. manufacturer_name: name of the manufacturer producing the detector, optional;
3. contact_person: full name of person responsible for the detector, optional;
4. description: common information about the detector, optional.

‘parameter_’ table (information about all parameters stored in the database):

1. parameter_id: identifier of the parameter, auto increment;
2. parameter_name: name of the parameter, required;
3. parameter_type: type of the parameter value (0 - boolean, 1-integer, 2 - double, 3 - string, 4 - int+int array, 5 - integer array, 6 - double array), required.

‘detector_parameter’ table (values of the detector parameters in the binary format):

1. value_id: identifier of the parameter value, auto increment;
2. detector_name: name of the detector having this parameter value, required;
3. parameter_id: identifier of the parameter with this value, required;
4. start_run: start run number of the range when the parameter value is valid, required;
5. end_run: end run number of the range when the parameter value is valid, required;
6. dc_serial: serial number of ADC/TDC if the parameter corresponds ADC or TDC, optional;
7. channel: ADC/TDC channel number which corresponds the parameter value, optional;
8. parameter_value: parameter’s value storing in the binary view and having the type as defined in the ‘parameter_’ table (‘parameter_type’ attribute).

‘shift_’ table (information about shifts of the experiment):

- shift_id: identifier of the shift, auto increment;
- period_number: run period number for the shift, required;
- start_datetime: start date and time of the shift, required;
- end_datetime: end date and time of the shift, required;
- FIO: full name of person responsible for the shift, required;

9. responsibility: area of person responsibility, optional.

'simulation_file' table (information about simulation files of the experiment):

- file_id: identifier of the file, auto increment;
- file_path: path to the simulation file, required;
- generator_name: name of the event generator used to produce the file, required;
- beam_particle: beam particle of events in the file, required;
- target_particle: type of target or second particle in collision for the file, optional;
- energy: collision energy (in GeV) of events in the file, optional;
- centrality: centrality of events in the file, required;
- event_count: event count in the file, optional;
- file_desc: short description of the file, optional;
- file_size: size of the file in MB, optional.

Tango parameters ('tango_parameter' virtual table) of slow control system are not corresponding to any table of the Unified Database, but its interface was implemented as C++ object class.

2.2 Database implementation.

The Unified Database is implemented on the PostgreSQL DBMS and provides user access to the actual information of the experiment: run parameters, detector geometries changing during the runs, experimental data obtained, etc. The SQL code used for deploying the database is saved in the 'scheme/*uni_db.sql*' file.

The common scheme of the Unified Database is presented in the Figure 2.

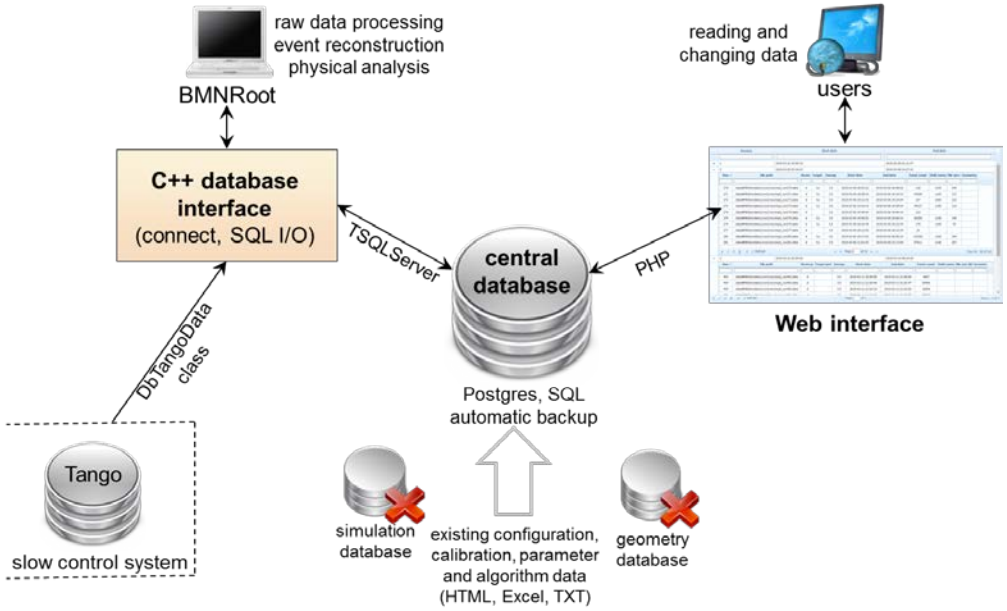

Figure 2. The common scheme of the Unified Database

The figure shows that the central Unified Database can parse experimental data of runs from multiple files. Users can easily read and change the data of the experiment by the Web-interface. The software environment of the experiment can use C++ database interface to get data for raw data processing, event reconstruction and physical analysis tasks. Also the Unified Database proposes the access to data of slow control system Tango by the developed 'UniDbTangoData' class.

3.0 GETTING STARTED

3.1 Access Levels

Users have a read-only access to the Unified Database by default. You can set your login and password to get another privileges in the corresponding ‘db username’ and ‘db password’ fields in the file ‘uni_db/*db_settings.h*’.

3.2 Test connection

In order to test connection to the database, you can run ‘test_db.C’ macro with ROOT CINT by the following command: “root –q uni_db/macros/test_db.C”. ‘Test was successful’ message means that connection to the database was established and test portion of data was read.

4.0 C++ DATABASE INTERFACE

4.1 The structure of C++ interface

The Unified Database main directory (uni_db) has the following structure:

'db_classes' directory contains C++ classes corresponding the database tables for reading and writing data without SQL statements.

UniDbRunPeriod.h(cxx) – class to work with run periods of the experiment. It has the following private members:

- int i_period_number: run period number;
- TDatime dt_start_datetime: start date and time of the run period;
- TDatime* dt_end_datetime: end date and time of the run period;
- TString* str_contact_person: full name of person responsible for the run period.

UniDbRun.h(cxx) – class to work with runs of the experiment:

- int i_run_number: run number;
- int* i_period_number: run period number for this run;
- TString str_file_path: path to the file with raw data of the run;
- TString str_beam_particle: beam particle of the run;
- TString* str_target_particle: type of target or second particle in the collision;
- double* d_energy: collision energy in GeV;
- TDatime dt_start_datetime: start date and time of the run;
- TDatime* dt_end_datetime: end date and time of the run;
- int* i_event_count: event count in the run;
- int* i_field_current: current supplied to the magnet in A;
- double* d_file_size: size of the raw file in MB;
- int* i_geometry_id: id of 'run_geometry' record with detector geometry of the run.

UniDbRunGeometry.h(cxx) – class to read/write detector geometry in the ROOT format:

- int i_geometry_id: identifier of the detector geometry;
- unsigned char* blob_root_geometry: binary data containing detector geometry in *.root file format;
- Long_t sz_root_geometry; size of the binary data with detector geometry.

UniDbDetector.h(cxx) – class to work with detectors of the experiment:

- TString str_detector_name: name of the detector;
- TString* str_manufacturer_name: name of the manufacturer producing the detector;
- TString* str_contact_person: full name of person responsible for the detector;
- TString* str_description: common information about the detector.

UniDbParameter.h(cxx) – class to create and change common information about parameters stored in the database:

- int i_parameter_id: identifier of the parameter;
- TString str_parameter_name: name of the detector parameter;
- int i_parameter_type: type of the parameter value (0 - boolean, 1-integer, 2 - double, 3 - string, 4 - int+int array, 5 - integer array, 6 - double array).

UniDbDetectorParameter.h(cxx) – class to work with values of detector parameters:

- int i_value_id: identifier of the parameter value;
- TString str_detector_name: name of the detector having this parameter value;
- int i_parameter_id: identifier of the parameter;
- int i_start_run: start run number of the range when the parameter value is valid;
- int i_end_run: end run number of the range when the parameter value is valid;
- int* i_dc_serial: serial number of ADC/TDC if the parameter corresponds ADC or TDC;
- int* i_channel: ADC/TDC channel number which corresponds the parameter value;
- unsigned char* blob_parameter_value: parameter's value storing in the binary view and having the type as defined in the 'parameter_' table ('parameter_type' attribute);
- Long_t sz_parameter_value: size of the parameter value in bytes.

UniDbShift.h(cxx) – class to work with information about shifts of the experiment:

- int i_shift_id: identifier of the shift;
- int i_period_number: run period number for the shift;
- TDatime dt_start_datetime: start date and time of the shift;
- TDatime dt_end_datetime: end date and time of the shift;
- TString str_fio: full name of person responsible for the shift;
- TString* str_responsibility: area of person responsibility.

UniDbSimulationFile.h(cxx) – class to work with information about simulation files:

- int i_file_id: identifier of the file;
- TString str_file_path: path to the simulation file;
- TString str_generator_name: name of the event generator used to produce the file;
- TString str_beam_particle: beam particle of events in the file;
- TString* str_target_particle: type of target or second particle in collision for the file;
- double* d_energy: collision energy (in GeV) of events in the file;
- TString str_centrality: centrality of events in the file;
- int* i_event_count: event count in the file;
- TString* str_file_desc: short description of the file;
- double* d_file_size: size of the file in MB.

If class member has a pointer type it means that this private member and corresponding field in the database can be empty (without any value) and can be set to 'NULL' (0x00) value.

'docs' directory contains Unified Database documentation:

Reference Manual.html – generated reference manual with information about classes, their members and functions;
The Unified Database User's Guide.pdf – this document (user's guide on how to use the database interfaces).

'examples' directory provides examples (ROOT macros) to work with the C++ interface:

add_new_parameter.C – creating new double parameter ‘voltage’ in the database;
add_parameter_value.C – adding new value of the ‘on’ parameter for ‘DCH1’ detector in 77 run;
add_parameter_value_complex.C – adding new value of the ‘noise’ parameter having ‘integer+integer’ type for ‘DCH1’ detector in 77 run (15-th slot №33-48, 16-th slot №49-64);
get_parameter_value.C – getting value of the ‘on’ parameter for ‘DCH1’ detector in 77 run;
get_parameter_value_complex.C – getting value of the ‘noise’ parameter having ‘integer+integer’ type for ‘DCH1’ detector in 133 run;
get_parameter_value_inl.C – getting values of the ‘INL’ parameter for all channels of TDC with serial number 0x0168fdca for ‘TOF1’ detector in 12 run;
get_root_geometry.C – getting ROOT file with detector geometry for a given run number;
histo_noise.C – getting noise channels (parameter values) stored as integer+integer (slot:channel) pairs for all runs (from 12 to 688) and filling the histogram;
set_detector_run_on.C – setting value of the ‘on’ parameter to true for different detector in all runs (from 12 to 168);
set_root_geometry.C – setting ROOT file with detector geometry for a given run range (from start number to end number).

‘macros’ directory provides ROOT macros for some specialized actions.

generate_cxx_from_db.C – macro to generate C++ classes of the interface (‘db_classes’ directory) or to update them if Unified Database structure has been changed;
parse_data_to_db.C – macro to convert text (txt, html, xml) files with parameters of the experiment to the database view and write them to the database, it uses xml schemes (some schemes are located in the ‘parse_schemes’ directory here);
test_db.C – macro to test connection to the database.

‘scheme’ directory includes database diagram and SQL script to deploy the Unified Database:

uni_db.png – common scheme (Entity-Relationship diagram) of the database structure described in the Section 2.1;
uni_db.sql – SQL code for deploying the database from scratch.

and the root (‘uni_db’) directory contains the following files:

db_settings.h – connection parameters described in the Section 3.1;
db_structures.h – includes the possible parameter’s types for writing (reading) to the database: BoolType, IntType, DoubleType, StringType, IIArrayType, IntArrayType, DoubleArrayType, and enumerations for searching (selecting) runs or detector parameters in the database (Section 4.2.3).
UniDbConnection.h(cxx) – class to open and close database connections;
UniDbGenerateClasses.h(cxx) – class for generating skeletons of the class wrappers for all tables of the database, it was used to generate classes of the C++ interface stored in the ‘db_classes’ directory;
UniDbParser.h(cxx) – class for converting (parsing) existing data of the experiment from text (txt, html, xml, excel) files with parameters and writing them to the Unified Database, it’s used by ‘parse_data_to_db.C’ macro;
UniDbSearchCondition.h(cxx) – class to form conditions for searching runs and detector parameters in the database;

UniDbTangoData.h(cxx) – class to get experimental data from Tango (‘slow’ control system) database.

4.2 Using C++ database interface

To process data of the experiment, C++ database interface was implemented as a set of C++ class wrappers for all database tables with many specific functions. It allows one to work with the data of the Unified Database without any SQL statements. The classes for reading and writing data to the Unified Database are listed at the beginning of the Section 4.1 (‘db_classes’ directory), they allow to manage information about run periods, shifts, runs, detectors, parameters and its values, detector geometries and generated simulation files.

4.2.1 Reading data objects from the database

To read objects from the database you can use **Get[ObjectName]** static function of the corresponding classes.

To read run period object (class UniDbRunPeriod):

UniDbRunPeriod* GetRunPeriod(int period_number)

To read run object you can use 2 functions (class UniDbRun):

UniDbRun* GetRun(int run_number)

or

UniDbRun* GetRun(TString file_path)

To read object with detector geometry by inner identifier (class UniDbRunGeometry):

UniDbRunGeometry* GetRunGeometry(int geometry_id)

To read detector geometry as binary array (*root_geometry* parameter) from the database for a selected run number (class UniDbRun):

int GetRootGeometry(int run_number, unsigned char* root_geometry, Long_t size_root_geometry)

To read detector geometry from the database for a selected run number and save it to geo file (class UniDbRun):

int ReadGeometryFile(int run_number, char* geo_file_path)

To read detector object (class UniDbDetector):

UniDbDetector* GetDetector(TString detector_name)

To read object of detector parameter (class UniDbParameter):

UniDbParameter* GetParameter(TString parameter_name)

To read object with common parameter value (class UniDbDetectorParameter):

UniDbDetectorParameter* GetDetectorParameter(TString detector_name, TString parameter_name, int run_number)

To read object with TDC/ADC parameter value (class UniDbDetectorParameter):

UniDbDetectorParameter* GetDetectorParameter(TString detector_name, TString parameter_name, int run_number, int dc_serial, int channel)

To read shift object (class UniDbShift):

UniDbShift* GetShift(TDatime shift_datetime)

To read simulation file object (class UniDbSimulationFile):

UniDbSimulationFile* GetSimulationFile(TString file_path)

The functions return pointer to object read from the database, except *GetRootGeometry* and *ReadGeometryFile* functions.

You can print all objects of the selected class on the screen by **PrintAll()** static function, e.g. to display all runs in the database you can write: UniDbRun::PrintAll().

4.2.2 Getting and writing object attributes to the database of the experiment

In order to get or change fields of the objects shown in the Section 4.1, you can use getter and setter functions of the classes. The getter function names are formed as **Get[CorFieldName]**, where fields are private members of the corresponding classes listed at the Section 4.1. Corrected field names are generated without type prefix and have uppercase letters at the beginning of each word in the name. For example, you can call GetRunNumber() (for 'i_run_number' field) function of UniDbRun class instance to get run number of the run object created or read from the database.

The setter function names are formed as **Set[CorFieldName]**, where corrected field names are generated as it's described above for getter functions. For example, you can call SetRunNumber() function of UniDbRun class instance to write this run number both to the run object and to the database for the corresponding run.

The getter and setter functions of all classes are listed below.

UniDbRunPeriod class:

- GetPeriodNumber()/SetPeriodNumber(int period_number): getting/setting run period number;
- GetStartDatetime()/SetStartDatetime(TDatime start_datetime): getting/setting start date and time of the run period;
- GetEndDatetime()/SetEndDatetime(TDatime* end_datetime): getting/setting end date and time of the run period;
- GetContactPerson()/SetContactPerson(TString* contact_person): full name of person responsible for the run period.

UniDbRun class:

- GetRunNumber()/SetRunNumber(int run_number): getting/setting run number;

- GetPeriodNumber()/SetPeriodNumber(int* period_number): getting/setting run period number for this run;
- GetFilePath()/SetFilePath(TString file_path): getting/setting path to the file with raw data of the run;
- GetBeamParticle()/SetBeamParticle(TString beam_particle): getting/setting beam particle of the run;
- GetTargetParticle()/SetTargetParticle(TString* target_particle): getting/setting type of target or second particle in collision;
- GetEnergy()/SetEnergy(double* energy): getting/setting collision energy in GeV;
- GetStartDatetime()/SetStartDatetime(TDatime start_datetime): getting/setting start date and time of the run;
- GetEndDatetime()/SetEndDatetime(TDatime* end_datetime): getting/setting end date and time of the run;
- GetEventCount()/SetEventCount(int* event_count): getting/setting event count in the run;
- GetFieldCurrent()/SetFieldCurrent(int* field_current): getting/setting current supplied to the magnet in ampere;
- GetFileSize()/SetFileSize(double* file_size): getting/setting size of the raw file in MB;
- GetGeometryId()/SetGeometryId(int* geometry_id): getting/setting integer identifier of detector geometry of the run.

UniDbRunGeometry class:

- int GetGeometryId():getting identifier of the detector geometry;
- GetRootGeometry()/SetRootGeometry(unsigned char* root_geometry, Long_t size_root_geometry):getting/setting binary data containing detector geometry in *.root file format;
- GetRootGeometrySize(): getting size of the binary data with detector geometry.

UniDbDetector class:

- GetDetectorName()/SetDetectorName(TString detector_name): getting/setting name of the detector;
- GetManufacturerName()/SetManufacturerName(TString* manufacturer_name): getting/ setting name of the manufacturer producing the detector;
- GetContactPerson()/SetContactPerson(TString* contact_person): getting/setting full name of person responsible for the detector;
- GetDescription()/SetDescription(TString* description): getting/setting common information about the detector.

UniDbParameter class:

- GetParameterId():getting inner identifier of the parameter;
- GetParameterName()/SetParameterName(TString parameter_name): getting/setting name of the detector parameter;
- GetParameterType()/SetParameterType(int parameter_type): getting/setting type of the parameter value: BoolType - boolean, IntType - integer, DoubleType - double, StringType - string, IIArrayType - int+int array, IntArrayType - integer array, DoubleArrayType - double array.

UniDbDetectorParameter class:

- GetDetectorName()/SetDetectorName(TString detector_name): getting/setting name of the detector;
- GetParameterId()/SetParameterId(int parameter_id): getting/setting inner identifier of the parameter having this value;
- GetStartRun()/SetStartRun(int start_run): getting/setting start run number of the range when the parameter value is valid;
- GetEndRun()/SetEndRun(int end_run): getting/setting end run number of the range when the parameter value is valid;
- GetDcSerial()/SetDcSerial(int* dc_serial): getting/setting serial number of ADC/TDC if the parameter corresponds ADC or TDC;
- GetChannel()/SetChannel(int* channel): getting/setting number of ADC/TDC channel which corresponds the parameter value;
- GetBool()/SetBool(bool parameter_value) – getting/setting boolean parameter's value, the parameter type is defined in the 'parameter_' table (*parameter_type* attribute);
  GetInt()/SetInt(int parameter_value) – -//- integer parameter's value;
  GetDouble()/SetDouble(double parameter_value) – -//- double parameter's value;
  GetString()/SetString(TString parameter_value) – -//- string parameter's value;
  GetIntArray(int* parameter_value, int element_count)/SetIntArray(int* parameter_value, int element_count) – -//- parameter's value as integer array;
  GetDoubleArray(double* parameter_value, int element_count)/SetDoubleArray(double* parameter_value, int element_count) – -//- parameter's value as double array;
  GetIIArray(IIStructure* parameter_value, int element_count)/SetIIArray(IIStructure* parameter_value, int element_count) – -//- parameter's value as array of integer+integer pairs.

UniDbShift class:

- GetPeriodNumber()/SetPeriodNumber(int period_number): getting/setting run period number for the shift;
- GetStartDatetime()/SetStartDatetime(TDatime start_datetime): getting/setting start date and time of the shift;
- GetEndDatetime()/SetEndDatetime(TDatime end_datetime): getting/setting end date and time of the shift;
- GetFio()/SetFio(TString fio): getting/setting full name of person responsible for the shift;
- GetResponsibility()/SetResponsibility(TString* responsibility): getting/setting personal area of responsibility.

UniDbSimulationFile class:

- GetFilePath()/SetFilePath(TString file_path): path to the simulation file;
- GetGeneratorName()/SetGeneratorName(TString generator_name): type of the event generator used to produce the file;
- GetBeamParticle()/SetBeamParticle(TString beam_particle): beam particle in collision events stored in the file;
- GetTargetParticle()/SetTargetParticle(TString* target_particle): type of target or second particle in collisions events stored in the file;

- GetEnergy()/SetEnergy(double* energy): collision energy (in GeV) for this file;
- GetCentrality()/SetCentrality(TString centrality): centrality of events in the file;
- GetEventCount()/SetEventCount(int* event_count): event count in the file;
- GetFileDesc()/SetFileDesc(TString* file_desc): short description of the file;
- GetFileSize()/SetFileSize(double* file_size): size of the file in MB.

If setter function has a parameter of pointer type it means that private member and corresponding field in the database can be empty (without any value) and can be set to 'NULL' (0x00) value. If getter function returns a variable of pointer type, private member and corresponding field in the database can be empty and the function will return NULL (0x00) value in this case.

You can print all object's attributes of the selected class on the screen by **Print()** member function, e.g. to display all data about selected run, you can call: pRun.Print(), where pRun is a pointer to the UniDbRun class instance.

4.2.3 Searching (selection) runs and detector parameter values by attributes

I) The UniDbRun class has 2 static search functions to get a collection of runs matching given criteria:

static TObjArray* Search(UniDbSearchCondition search_condition) – get runs corresponding to the specified single condition, or
static TObjArray* Search(TObjArray search_conditions) – get runs corresponding to the specified (vector) conditions.

The functions above return pointer to the object array of UniDbRun objects matching to the specified conditions. The searching conditions are defined by UniDbSearchCondition class. To create a class instance, you can use one of the following constructors:

UniDbSearchCondition(enumColumns column, enumConditions condition, int value);
UniDbSearchCondition(enumColumns column, enumConditions condition, double value);
UniDbSearchCondition(enumColumns column, enumConditions condition, TString value);
UniDbSearchCondition(enumColumns column, enumConditions condition, TDatime value).

The enumerations are defined in 'uni_db/*db_structures.h*' file. The first 'column' parameter specifies the attribute (column) for condition to select the runs. You can select the following fields to filter by:

columnRunNumber – run number;
columnPeriodNumber – run period number;
columnFilePath – 'raw' file path;
columnBeamParticle – short name of the beam particle;
columnTargetParticle – short name of the target particle;
columnEnergy – collision energy in GeV;
columnStartDatetime – start date and time of the run;
columnEndDateTime – end date and time of the run;
columnEventCount – event count of the run;
columnFieldCurrent – current supplied to the magnet in ampere;

columnFileSize – size of the raw file in MB.

The second parameter of the functions specifies comparison with the value passed in the third parameter. The enumeration (enumConditions) includes the following comparison operators:

conditionLess – less than value;
conditionLessOrEqual – less or equal to the value;
conditionEqual – equal to the value;
conditionNotEqual – non-equal to the value;
conditionGreater – greater than value;
conditionGreaterOrEqual –greater or equal to the value;
conditionLike – whether a specific character string matches a specified pattern.

The value passed as the third parameter can be integer, double, TString or TDatime type which corresponds the type of column passed in the first parameter. For example:

1. UniDbSearchCondition searchCondition(columnRunNumber, conditionGreater, 400);
   TObjArray* pRunArray = UniDbRun::Search(search_condition);
   if (pRunArray != NULL) UniDbRun* pFirstSelectedRun = (UniDbRun*) pRunArray[0];
2. TObjArray arrayConditions;
   UniDbSearchCondition* searchCondition =
   new UniDbSearchCondition(columnTargetParticle, conditionEqual, “C”);
   arrayConditions.Add(searchCondition);
   searchCondition = new UniDbSearchCondition(columnFilePath, conditionLike, “mbias”);
   arrayConditions.Add(searchCondition);
   TObjArray* pRunArray = UniDbRun::Search(arrayConditions);
   …
   for (int i = 0; i < arrayConditions->GetEntriesFast(); i++)
   delete (UniDbSearchCondition*) arrayConditions[i];
   arrayConditions.Clear();

II) The UniDbDetectorParameter class also has 2 static search functions to get a collection of parameter values matching given criteria:

static TObjArray* Search(UniDbSearchCondition search_condition) - get parameters’ values corresponding to the specified single condition, or
static TObjArray* Search(TObjArray search_conditions) - get parameters’ values corresponding to the specified (vector) conditions.

The functions above return pointer to the object array of UniDbDetectorParameter objects matching to the specified conditions. The searching conditions are also defined by UniDbSearchCondition class. Creating a class instance of the UniDbSearchCondition is shown above.

The enumerations of UniDbSearchCondition are defined in ‘uni_db/*db_structures.h*’ file. The first parameter of the UniDbSearchCondition constructor specifies the attribute (column) to select parameters’ values by given conditions. You can select the following fields to filter by:

columnDetectorName – name of the detector;

columnParameterName – name of the parameter having this value;
columnStartRun – start run number of the range when the parameter value is valid;
columnEndRun – end run number of the range when the parameter value is valid;
columnDCSerial – serial number of ADC/TDC if the parameter corresponds ADC or TDC;
columnChannel – number of ADC/TDC channel which corresponds the parameter value.

The second parameter of the functions specifies comparison with the value passed in the third parameter. The 'enumConditions' enumeration is described above. The value passed as the third parameter can be integer, double, TString or TDatime type which corresponds the type of column passed in the first parameter.

4.2.4 Auxiliary functions

Some classes have additional functions to extend the functionality of the database interface.

UniDbRun class:

- static int GetRunNumbers(int start_run, int end_run, int* run_numbers) – get numbers of runs existing in the database for a selected range, it returns integer array ('run_numbers') of existing numbers as third parameter.
- static int GetRunNumbers(int* run_numbers) – get all numbers of existing runs in the database, it returns integer array ('run_numbers') of existing numbers as first parameter.

UniDbDetectorParameter class:

- static int GetChannelCount(TString detector_name, TString parameter_name, int run_number, int dc_serial) – get channel count for TDC/ADC parameter value.

4.2.5 Writing a new data (data object) of the experiment to the database

To write new objects to the database you can use **Create[ObjectName]** static function of the corresponding classes:

To add a new run period (class UniDbRunPeriod):

UniDbRunPeriod* CreateRunPeriod (int period_number, TDatime start_datetime, TDatime *end_datetime, TString *contact_person)

To add a new run (class UniDbRun):

UniDbRun* CreateRun (int run_number, int *period_number, TString file_path, TString beam_particle, TString *target_particle, double *energy, TDatime start_datetime, TDatime *end_datetime, int *event_count, int *field_current, double *file_size, int *geometry_id)

To write new detector geometry from geo file to the database for runs from start_run_number to end_run_number (class UniDbRun):

int WriteGeometryFile(int start_run_number, int end_run_number, char* geo_file_path)

To write new detector geometry from binary array (unsigned char*) to the database for runs from start_run_number to end_run_number (class UniDbRun):

int SetRootGeometry(int start_run_number, int end_run_number, unsigned char* root_geometry, Long_t size_root_geometry)

To add a new detector (class UniDbDetector):

UniDbDetector* CreateDetector (TString detector_name, TString *manufacturer_name, TString *contact_person, TString *description)

To add a new parameter (class UniDbParameter):

UniDbParameter* CreateParameter (TString parameter_name, int parameter_type)

To add common parameter values (class UniDbDetectorParameter):

UniDbDetectorParameter* CreateDetectorParameter(TString detector_name, TString parameter_name, int start_run, int end_run, <*parameter type*> parameter_value[, int element_count]),

where <parameter type> can be *bool*, *int*, *double*, *TString*, *int**, *double**, *IIStructure**. The last parameter *element_count* is required only if <parameter type> has a pointer type.

To add TDC/ADC parameter values (class UniDbDetectorParameter):

UniDbDetectorParameter* CreateDetectorParameter(TString detector_name, TString parameter_name, int start_run, int end_run, int dc_serial, int channel, <*parameter type*> parameter_value[, int element_count]),

where <parameter type> can be *bool*, *int*, *double*, *TString*, *int**, *double**, *IIStructure**. The last parameter *element_count* is required only if <parameter type> has a pointer type.

To add a new shift (class UniDbShift):

UniDbShift* CreateShift (int period_number, TDatime start_datetime, TDatime end_datetime, TString fio, TString *responsibility)

To write data about new simulation file (class UniDbSimulationFile):

UniDbSimulationFile* CreateSimulationFile (TString file_path, TString generator_name, TString beam_particle, TString *target_particle, double *energy, TString centrality, int *event_count, TString *file_desc, double *file_size)

The functions return pointer to the new object created in the database, except *WriteGeometryFile* and *SetRootGeometry* functions. If parameter of the functions has a pointer type then you can set it to ‘NULL’ (0x00) value. It means that the corresponding attribute in the database will be empty (without any value).

4.2.6 Deleting data of the experiment from the database

To remove data objects from the database you can use **Delete[ObjectName]** static function of the corresponding classes:

To delete run period (class UniDbRunPeriod):

int DeleteRunPeriod(int period_number)

To delete run you can use 2 functions (class UniDbRun):

static int DeleteRun(int run_number)

or

static int DeleteRun(TString file_path)

To delete detector geometry (class UniDbRun):

int DeleteRunGeometry(int geometry_id)

To delete detector (class UniDbDetector):

int DeleteDetector(TString detector_name)

To delete parameter from the database (class UniDbParameter):

int DeleteParameter(TString parameter_name)

To delete data about simulation file (class UniDbSimulationFile):

int DeleteSimulationFile(TString file_path)

The functions return integer value which is equal to 0 if no error occurred, otherwise it returns the error code.

4.3 Reading electronics parameters from Tango (slow control) database

The interface to the database of the slow control system is implemented as C++ object class – UniDbTangoData. UniDbTangoData class is used to read electronics data from the Tango control system based on MySQL DBMS.

Figure 3 presents an example of using GetTangoParameter function to get the high voltage of ZDC for a selected period of time.

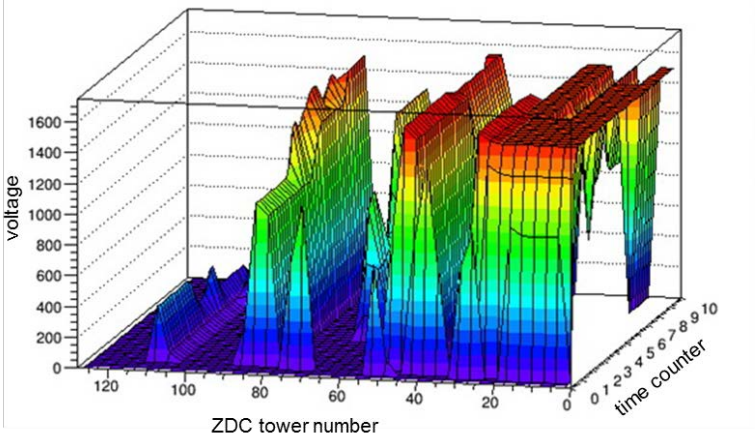

Figure 3. The High Voltage of Zero Degree Calorimeter

4.4 The Web-interface of the Unified Database

The Web-interface of the Unified Database is being developed to simplify reading and changing data of the experiment over the Web page.

APPENDIX A. C++/ROOT EXAMPLES

1. Creating new parameters.

Creating ‘voltage’ parameter of double type in the database (*add_new_parameter.C* macro).

```
// load UniDb library
gSystem->Load("libUniDb");

// add 'voltage' parameter,
UniDbParameter* pParameter = UniDbParameter::CreateParameter("voltage", DoubleType);
if (pParameter == NULL)
{
   cout << "\nMacro finished with errors" << endl;
   return;
}

// clean memory after work
delete pParameter;
```

2. Writing new detector parameter’s values.

a) Adding value of the ‘on’ parameter for ‘DCH1’ detector in 77 run (*add_parameter_value.C*).

```
// load UniDb library
gSystem->Load("libUniDb");

// set 'on' parameter value (boolean value)
UniDbDetectorParameter* pDetectorParameter =
        UniDbDetectorParameter::CreateDetectorParameter("DCH1", "on", 77, 77, true);
if (pDetectorParameter == NULL)
{
   cout << "\nMacro finished with errors" << endl;
  return;
}

// clean memory after work
delete pDetectorParameter;
```

b) Adding value of the ‘on’ parameter (true value) for different detector in all runs (from 12 to 168) (*set_detector_run_on.C*).

```
// load UniDb library
gSystem->Load("libUniDb");

// add 'on' parameter value for detectors
UniDbDetectorParameter* pDetectorParameter =
UniDbDetectorParameter::CreateDetectorParameter("DCH1", "on", 12, 688, true);
// clean memory
if (pDetectorParameter) delete pDetectorParameter;
```

```
    pDetectorParameter = UniDbDetectorParameter::CreateDetectorParameter("DCH2", "on",
12, 688, true);
    // clean memory
    if (pDetectorParameter) delete pDetectorParameter;

    pDetectorParameter = UniDbDetectorParameter::CreateDetectorParameter("TOF1", "on", 12,
688, true);
    // clean memory
    if (pDetectorParameter) delete pDetectorParameter;

    pDetectorParameter = UniDbDetectorParameter::CreateDetectorParameter("TOF2", "on", 12,
688, true);
    // clean memory
    if (pDetectorParameter) delete pDetectorParameter;

    pDetectorParameter = UniDbDetectorParameter::CreateDetectorParameter("ZDC", "on", 12,
688, true);
    // clean memory
    if (pDetectorParameter) delete pDetectorParameter;
```

c) Adding value of the ‘noise’ parameter of ‘integer+integer’ type for ‘DCH1’ detector in 77 run (15-th slot no. 33-48, 16-th slot no. 49-64) (*add_parameter_value_complex.C*).

```
    // load UniDb library
    gSystem->Load("libUniDb");

    bool return_error = false;

    // add noise parameter value presented by IIStructure: Int+Int pair (slot:channel)
    IIStructure* pValues = new IIStructure[32];
    AssignIIStructure(pValues, 0, 15, 33, 48); // slot: 15, channel: 33-48
    AssignIIStructure(pValues, 16, 16, 49, 64); // slot: 16, channel: 49-64

    UniDbDetectorParameter* pDetectorParameter =
      UniDbDetectorParameter::CreateDetectorParameter("DCH1", "noise", 77, 77, pValues, 32);
     if (pDetectorParameter == NULL)
       return_error = true;

    // clean memory after work
    delete [] pValues;
    if (pDetectorParameter)
      delete pDetectorParameter;

    if (return_error) cout << "\nMacro finished with errors" << endl;
```

3. Reading detector parameter’s values.

a) Getting value of the ‘on’ parameter for ‘DCH1’ detector in 77 run (*get_parameter_value.C*).

```
    // load UniDb library
    gSystem->Load("libUniDb");
```

```
// get 'on' parameter value (boolean value)
   UniDbDetectorParameter* pDetectorParameter =
UniDbDetectorParameter::GetDetectorParameter("DCH1", "on", 77); //(detector_name,
parameter_name, run_number)
if (pDetectorParameter == NULL)
{
    cout << "\nMacro finished with errors" << endl;
    return;
 }

bool is_on = pDetectorParameter->GetBool();
if (is_on)
    cout<<"Detector DCH1 was turned on in run n.77"<<endl;
else
    cout<<"Detector DCH1 was turned off in run n.77"<<endl;

 // clean memory after work
 delete pDetectorParameter;
```

b) Getting value of the ‘noise’ parameter having ‘integer+integer’ type for ‘DCH1’ detector in 133 run (*get_parameter_value_complex.C*).

```
// load UniDb library
gSystem->Load("libUniDb");

bool return_error = false;

// get noise parameter values presented by IIStructure: Int+Int pair (slot:channel)
UniDbDetectorParameter* pDetectorParameter =
      UniDbDetectorParameter::GetDetectorParameter("DCH1", "noise", 133);
 if (pDetectorParameter != NULL)
{
    IIStructure* pValues;
    int element_count = 0;
    pDetectorParameter->GetIIArray(pValues, element_count);

    // e. g. print values
    for (int i = 0; i < element_count; i++)
       cout<<"Slot:Channel "<<pValues[i].int_1<<":"<<pValues[i].int_2<<endl;

   // clean memory after work
    delete pValues;
    if (pDetectorParameter)
       delete pDetectorParameter;
}
else
    return_error = true;

if (return_error) cout << "\nMacro finished with errors" << endl;
```

c) Getting values of the INL parameter for all channels of TDC with serial number 0x0168fdca for ‘TOF1’ detector in 12 run (*get_parameter_value_inl.C*).

```
// load UniDb library
gSystem->Load("libUniDb");

// get 'INL' parameter value (int array) for 72 channels
bool is_error = false;
int TDC_SERIAL = (int)0x0168fdca;

int channel_count = UniDbDetectorParameter::GetChannelCount("TOF1", "inl", 12,
TDC_SERIAL);
if (channel_count == 0)
   cout<<"The detector parameter wasn't found"<<endl;
else
   cout<<"Channel number is equal "<<channel_count<<endl;

for (int i = 1; i <= channel_count; i++)
{
   UniDbDetectorParameter* pDetectorParameter =
       UniDbDetectorParameter::GetDetectorParameter("TOF1", "inl", 12, TDC_SERIAL, i);
   if (pDetectorParameter == NULL)
   {
      is_error = true;
      continue;
   }

   int inl_size = -1;
   double* inl_for_channel = NULL;
   int res_code = pDetectorParameter->GetDoubleArray(inl_for_channel, inl_size);
   if (res_code != 0)
   {
      is_error = true;
      continue;
   }

   cout<<"TDC: "<<int_to_hex_string(TDC_SERIAL)<<". Channel: "<<i<<endl<<"INL:";
   for (int j = 0; j < inl_size; j++)
      cout<<" "<<inl_for_channel[j];
   cout<<endl<<endl;

   // clean memory after work
   delete pDetectorParameter;
   delete [] inl_for_channel;
}

if (is_error) cout << "\nMacro finished with errors" << endl;
```

d) Getting noise channels (parameter values) presented by integer+integer pairs (slot:channel) for all runs (from 12 to 688) and filling the histogram (*histo_noise.C*).

```
// load UniDb library
gSystem->Load("libUniDb");

TCanvas* c = new TCanvas("c", "Noise channels", 0, 0, 640, 480);
TH3I* histo =  new TH3I("h3","Noise channels",20,1,20,70,1,70,689,0,688);
histo->SetMarkerColor(kBlue);
histo->GetXaxis()->SetTitle("Slot");
histo->GetYaxis()->SetTitle("Channel");
histo->GetZaxis()->SetTitle("Run №");

int* run_numbers;
int run_count = UniDbRun::GetRunNumbers(12, 688, run_numbers);
if (run_count <= 0)
   return;

for (int i = 0; i < run_count; i++)
{
   int run_number = run_numbers[i];
   // get noise parameter values presented by IIStructure: Int+Int (slot:channel)
   UniDbDetectorParameter* pDetectorParameter =
        UniDbDetectorParameter::GetDetectorParameter("DCH1", "noise", run_number);
   if (pDetectorParameter != NULL)
  {
      IIStructure* pValues;
      int element_count = 0;
      pDetectorParameter->GetIIArray(pValues, element_count);

      cout<<"Element count: "<<element_count<<" for run number: "<<run_number<<endl;
      for (int j = 0; j < element_count; j++)
         histo->Fill(pValues[j].int_1, pValues[j].int_2, run_number);

      // clean memory after work
      delete pValues;
     delete pDetectorParameter;
  }
}

delete [] run_numbers;

histo->Draw();
```

4. Working with detector geometry.

a) Adding ROOT file with detector geometry for a given run range (*set_root_geometry.C*).

```
// load UniDb library
gSystem->Load("libUniDb");
```

```
// write ROOT file with detector geometry for run range
int res_code = UniDbRun::WriteGeometryFile(start_run, end_run, root_file_path);
if (res_code != 0)
{
   cout << "\nMacro finished with errors" << endl;
   exit(-1);
}
```

b) Getting ROOT file with detector geometry for given run number (*get_root_geometry.C*).

```
// load UniDb library
gSystem->Load("libUniDb");

int res_code = UniDbRun::ReadGeometryFile(run_number, root_file_path);
if (res_code != 0)
{
   cout << "\nMacro finished with errors" << endl;
   exit(-1);
}

// get gGeoManager from ROOT file (if required)
TFile* geoFile = new TFile(root_file_path, "READ");
if (!geoFile->IsOpen())
{
   cout<<"Error: could not open ROOT file with geometry!"<<endl;
   exit(-2);
}

TList* keyList = geoFile->GetListOfKeys();
TIter next(keyList);
TKey* key = (TKey*)next();
TString className(key->GetClassName());
if (className.BeginsWith("TGeoManager"))
   key->ReadObj();
else
{
   cout<<"Error: TGeoManager isn't top element in given file "<<root_file_path<<endl;
   exit(-3);
}

TGeoNode* N = gGeoManager->GetTopNode();
cout<<"The top node of gGeoManager is "<<N->GetName()<<endl;
```

5. Run selection according to the specific conditions.

a) Adding ROOT file with detector geometry for a given run range (*set_root_geometry.C*).

```
// load UniDb library
gSystem->Load("libUniDb");

TObjArray arrayConditions;
```

```
        UniDbSearchCondition* searchCondition = new UniDbSearchCondition(columnBeamParticle,
conditionEqual, TString("d"));
        arrayConditions.Add((TObject*)searchCondition);
        //searchCondition = new UniDbSearchCondition(columnTargetParticle, conditionNull);
        searchCondition = new UniDbSearchCondition(columnTargetParticle, conditionEqual,
TString("Cu"));
        arrayConditions.Add((TObject*)searchCondition);

        TObjArray* pRunArray = UniDbRun::Search(arrayConditions);

        // clean memory for conditions after search
        for (int i = 0; i < arrayConditions.GetEntriesFast(); i++)
            delete (UniDbSearchCondition*) arrayConditions[i];
        arrayConditions.Clear();

        // print run numbers obtained
        for (int i = 0; i < pRunArray->GetEntriesFast(); i++)
        {
            UniDbRun* pRun = (UniDbRun*) pRunArray->At(i);
            cout<<"Run (d-Cu): number - "<<pRun->GetRunNumber()<<", file path - "<<pRun-
>GetFilePath()<<endl;
        }

        // clean memory after work
        for (int i = 0; i < pRunArray->GetEntriesFast(); i++)
            delete (UniDbRun*)pRunArray->At(i);
        delete pRunArray;
```