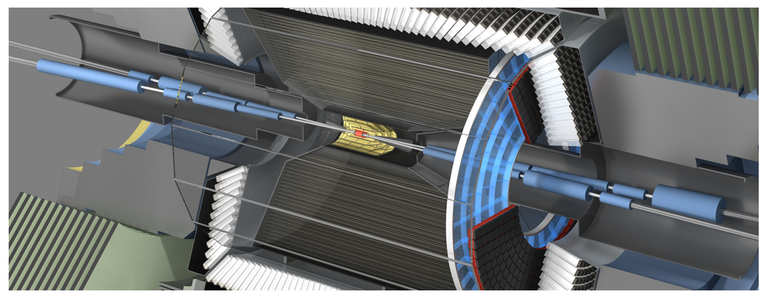
# Belle II Conditions Database Interface

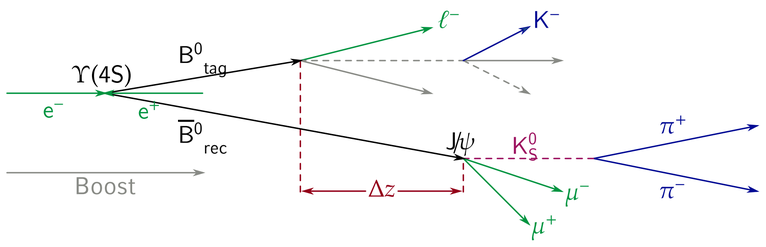


18th International Workshop on Advanced Computing and Analysis Techniques in Physics Research

Martin Ritter[1], Thomas Kuhr[1], Christian Pulvermacher[2] for the Belle II Collaboration
[1]Ludwig-Maximilians-University, Munich [2]High Energy Accelerator Research Organization, Japan

## The Belle II Experiment

An electron positron collider with asymmetric energies located in Japan to test the standard model with high precision.

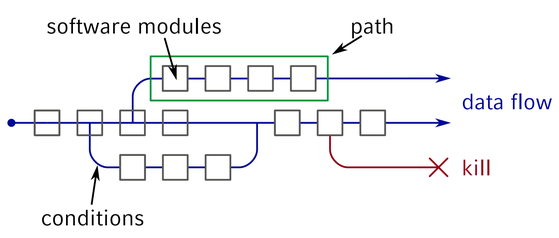



- start in 2018, collect 50 $ab^{-1}$ until 2024
- record $4 \times 10^{11}$ events, 60 PB of data
- generate simulated data with at least the same statistics

## Software Framework

Software framework written from scratch using experience from Belle and other HEP experiments.

- core framework implemented in C++14 and including the boost libraries
- use ROOT 6 framework for serialization of event data, Geant4 for simulation
- Python 3 interface for configuration and high level program steering
- different algorithms (called modules) are executed sequentially for each event




## Conditions Data at Belle II

Conditions data are configuration/calibration items which depend on the conditions during data taking.

- database contains information on *run* granularity
- finer granularity to be handled on client side
- should be transparent to the user
- needs to work on closed DAQ network without outside connection

➡ See Wood et al, "Implementing the Belle II Conditions Database Using Industry-Standard Tools" for details on the server

➡ See Bilka et al, "Alignment and Calibration Framework for the Belle II Detector" for details on calibration procedure.

## Design Decisions

- use ROOT objects for conditions data
- identify by name
- default name is the class name

## Read Access of Conditions Objects

Two template classes which always provide pointer to correct payload

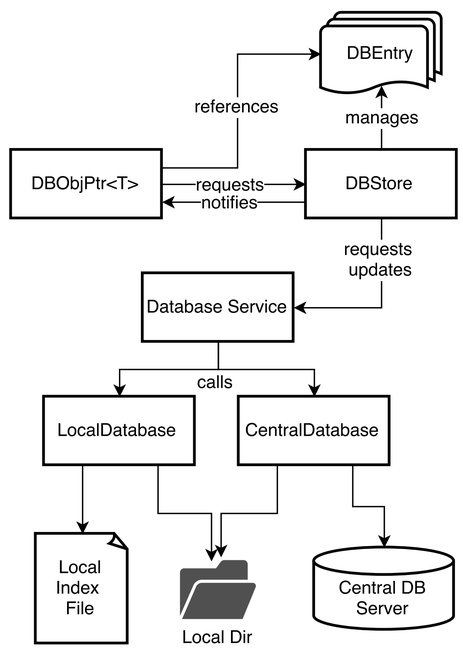
- `DBObjPtr` for single objects
- `DBArray` for arrays of objects
- shallow class, just points to common area

```
class MyConditionsClass: public TObject {
  public:
    MyConditionsClass(const std::string &string):
      TObject(), m_string(string) {}
    const std::string& getString() const {
      return m_string;
    }
  private:
    std::string m_string;
};

DBObjPtr<MyConditionsClass> myObj;
if(!conditionsObj){
  B2ERROR("No Conditions data Available");
}else{
  B2INFO("Conditions: " << myObj->getString());
}
DBArray<MyConditionsClass> myList("SpecialName");
B2INFO("Found " << myList.getEntries() " objects");
```

Hides updates from the user

- user can check if payload changed
- user can register callback on change




## Creation of Payloads

similar classes to create payloads

- `DBImportObjPtr` and `DBImportArray`
- allow to create payloads with simple interface

## IntraRun Dependency

Some payloads might change more frequent than per *run*, for example Beamspot positions

- handled completely on client side
- different types of dependencies: event number, time stamp, ...
- usage completely transparent

```
DBImportObjPtr<MyConditionsClass> myObjImport;
myObjImport.construct("initial value");
myObjImport.addEventDependency(10, "from event 10");
myObjImport.addEventDependency(50, "from event 50");
myObjImport.import(iov);
```

## Different Storage Backends

Software offers different storage backends

- using REST api to obtain payloads from central database
- using local folder with payloads and text file defining validity.

This simplifies development and debugging:

- users can create and test their payloads locally without uploading
- users can continue to develop without internet connection
- snapshots of the database can be downloaded for isolated environments

## Configuration and Usage

Usage of the database can be easily configured from the steering file

- several backends can be searched in order
- access to central database can be completely disabled

```
import basf2
# clear defaults
basf2.reset_database()
# use more then one source for payloads
basf2.use_database_chain()
# local fallback database looking in folder "db/"
basf2.use_local_database("db/payloads.txt", "db/")
# use central database as primary source and obtain
# payloads from global tag "mytag"
basf2.use_central_database("mytag")
```

## Distribution of Payloads

Payload files are downloaded from the server using http and stored in a local directory for caching

- if all payloads are found locally only metadata is obtained from server
- md5sum of file is checked before opening
- alternative distribution paths possible (cvmfs, xrootd, MICA, git-packfiles)

## Command Line Interface

Rest api very well suited for standalone command line interface

➡ very easy to have independent implementations

- git like cli for management
- written in Python using requests
- manage/modify/show content of database
- show differences between global tags
- batch upload/download of payloads
- dump payload content in readable form

Bundesministerium für Bildung und Forschung