Development of a lightweight database interface for accessing JUNO conditions and parameters data

Tao Lin[1] **(on behalf of the JUNO collaboration)**

[1] IHEP

E-mail: `lintao@ihep.ac.cn`

Abstract.

The Jiangmen Underground Neutrino Observatory (JUNO) has a very rich physics program which primarily aims to the determination of the neutrino mass ordering and to the precisely measurement of oscillation parameters. It is under construction in South China at a depth of about 700 m underground. As data taking will start in 2024, a complete data processing chain is developed before the data taking. Conditions and parameters data, as non-event data, are one of important parts in the data processing chain, which are used by reconstruction and simulation. These data could be accessed via Frontier on JUNO-DCI (Distributed Computing Infrastructure), or via databases, such as MySQL and SQLite in local clusters.

In this contribution, the latest development of a lightweight database interface (DBI) for JUNO conditions and parameters data management system will be shown. This interface provides a unified method to access data from different backends, such as Frontier, MySQL and SQLite: production jobs could run on JUNO-DCI with Frontier; testing jobs could run in a local cluster with MySQL to validate the conditions and parameters data; fast reconstruction could run in a DAQ environment onsite using SQLite without any connections to remote database. Modern C++ template techniques are used in DBI: extension of a new backend is defined by a simple `struct` with two methods `doConnect` and `doQuery`; result sets are binding to `std::tuple` and the types of all the elements are known at compile-time. Finally, DBI is used by high-level user interfaces: data models in the database are mapping to normal C++ classes, so that users could access these objects without knowing DBI.

1. Introduction

The Jiangmen Underground Neutrino Observatory (JUNO) experiment [1] has a rich physics program, including the determination of the neutrino mass ordering, precise measurement of neutrino oscillation parameters, detecting neutrinos from reactor, atmosphere, solar, supernova burst, etc [2, 3]. JUNO is under construction in southern China at a depth of about 700 m underground. It is expected to start data-taking in 2024, running for more than 30 years.

As shown in Figure 1, the JUNO detector consists of a central detector, a water Cherenkov detector, and a top tracker. The innermost part is the central detector with an acrylic spherical vessel filled with 20 kton liquid scintillator (LS), equipped with 17,612 20-inch photomultiplier tubes (LPMT) and 25,600 3-inch photomultiplier tubes (SPMT). The central detector is submerged in a water pool, equipped with 2,400 LPMTs, which is the water Cherenkov detector to detect cosmic ray muons. On the top of the water pool, the top tracker is also used to measure the muons.

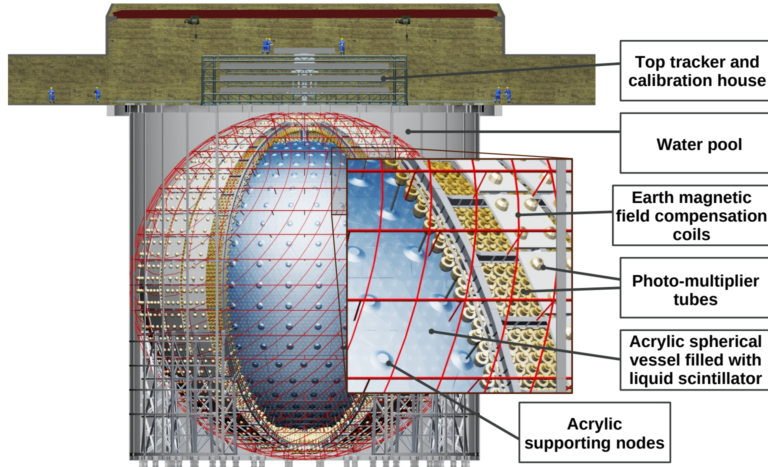

Figure 1. Schematic view of the JUNO detector

The data processing flow is described as follows. Waveforms from PMTs are read out and then processed with Online Event Classification (OEC) with fast reconstruction and classification algorithms by the DAQ servers. Instead of discarding events, OEC will keep all the events and decide whether to save the raw waveforms or not, so that the data volume could be reduced. Then the raw data are transferred to the IHEP computing center with a dedicated network. When the data arrives, the data quality monitoring (DQM) system starts to reconstruct the data. In order to reduce the latency and show the data quality promptly, only parts of the data will be processed. Meanwhile, the raw data are processed with a system called keep-up production (KUP). All the data are processed in a dedicated computer cluster with 5,000 CPU cores. The calibration constants used in KUP are not the latest. After the calibration experts update the calibration constants for all the data, the latest calibration constants are then used in physics production (PP). All data are processed again with the latest calibration constants in a DIRAC based distributed computing system, the so-called JUNO-DCI (Distributed Computing Infrastructure) [4].

2. Motivation and design

The metadata and payloads of calibration constants and parameters are stored separately in the current design [5]. When accessing the calibration constants and parameters, the metadata is queried first, and then these data are read from the file system. The metadata are stored in a MySQL database [6] and distributed by the frontier distributed database caching system [7]. The payloads are distributed by the CVMFS [8]. However, there are some special use cases. As the OEC runs on the DAQ servers onsite, the calibration constants can not be accessed remotely. SQLite database [9] is adopted in such cases. The SQLite database is a regular file, which is easy to copy to the DAQ servers. The necessary constants are dumped from the official database and then saved in the SQLite database. Another case is used for the validation of calibration constants by calibration experts. When the calibration experts produce a new set of constants, these constants need to be validated before being published in the frontier system. That requires the availability of these metadata in the MySQL database.

In order to support the accessing of multiple database backends, including frontier, MySQL, and SQLite in different use cases, a unified method is necessary to avoid duplication and improve the maintenance of the source code. A lightweight database interface (DBI) [10] has been designed and implemented in JUNO offline software (JUNOSW) [11]. It consists of a low-

level database API (DBAPI) and high-level applications within the SNiPER framework [12], as shown in Figure 2. Modern C++ template is used to implement the DBAPI, allowing users to use different database backends and customize the types at compile time. A service named `DBISvc` is developed to manage and configure the DBAPI instances using a YAML file [13]. High-level applications, named `CondDB` and `ParaDB`, have been also developed on top of DBAPI to access the metadata of conditions and parameter data respectively. These applications retrieve DBAPI instances from the `DBISvc`. They are then used by the clients, such as `MCParamSvc` and `PMTCalibSvc`. After getting the metadata, these clients read the payloads according to the paths in the metadata.

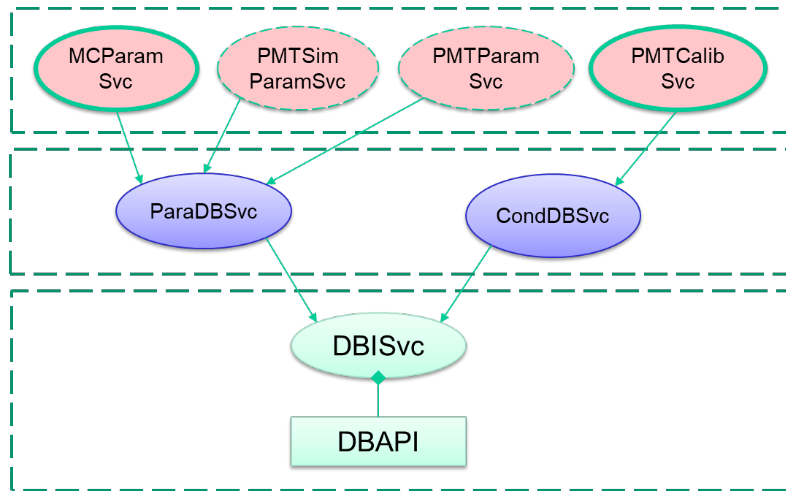

Figure 2. Design of the DBI in JUNOSW

3. Implementation of DBAPI

The DBAPI is a header-only, modern C++ library to access databases with different backends. It contains a unified user interface and several database backend classes.

The class named `DBAPI` is the user interface to execute SQL queries. The queries are actually executed by another class, named the backend class, which is implemented with the underlying database APIs. The C++ templates are used to avoid any inheritance for the database backend classes, so adding a normal C++ class could extend the backend. The normal class needs two methods: one is `doConnect` to setup a connection to the database; another is `doQuery` to execute an SQL statement and return the results with many rows. A class named `ResultSet` is used to store the results in one row, which uses `std::string` to represent each field. For example, several native C-APIs of MySQL [14] are used to implement a MySQL database backend: `mysql_real_query` is used to execute the SQL statement; `mysql_store_result` is then used to store all the results; then, `mysql_fetch_row` is used to get results in one row and put them in `ResultSet`. After getting the results, the `DBAPI` instance is responsible to convert the results into user-defined types. The `std::tuple` is used to contain these values with different types. The `ResultSet` object implements a template method to convert each element to the correct type. Following is a code snippet of `ResultSet`:

```
struct ResultSet {
    template<typename... Args>
    bool to(std::tuple<Args...>& result) const {
        typedef typename std::tuple<Args...> TupleT;
```

```
        if (m_internals.size() != std::tuple_size_v<TupleT>) {
            return false;
        }

        tuple_element_helper<
            std::tuple_size_v<TupleT>, 0>(
                m_internals, result);
        return true;
    };

    std::vector<std::string> m_internals;
};
```

All of the types need to be decided at compile time and the check of types is done by the compiler. The following shows an example of executing queries using frontier:

```
dbi::FrontierDB frontierdb{url}; // FrontierDB is a backend class
auto dbapi = new dbi::DBAPI(frontierdb); // Creating DBAPI instance
dbapi->connect();

using GlobalTag_t = std::tuple<std::string, std::string, std::string
                               std::string, std::string>;

for (auto result: dbapi->query<GlobalTag_t>(stmt_globaltag)) {
    auto [sftver, condgtag, paragtag, creator, createtime] = result;
}
```

However, some data types are already defined as classes instead of `std::tuple`. To keep the existing classes unchanged, another query interface is added to use a function to create the dedicated objects. The following example shows how to use C++ lambda expression to create the object:

```
for (auto result: dbapi->query<GlobalTag>(stmt_globaltag,
    [](std::string sftver, std::string condgtag, std::string paragtag,
       std::string creator, std::string createtime) -> GlobalTag {
       return GlobalTag{sftver, condgtag, paragtag, creator, createtime};
      })) {
    // result is a GlobalTag instance
}
```

4. Applications in JUNOSW

As mentioned in Section 2, there are several layers on top of `DBAPI`. These components are all implemented as services in SNiPER.

The `DBISvc` is responsible to manage a collection of `DBAPI` instances, which are controlled in the YAML file. A unique connection name is assigned to each instance and the properties of the instance could be configured by the key-value pairs. A required property is `backend`, which is used by `DBISvc` to create the corresponding instances. The other properties are used to initialize the instances, which could be different for different backends. Following example shows several instances are created with different properties in the YAML file.

```
connections:
  frontier_connection:
    backend: frontier
    server: http://put_frontier_server_here:8000/Frontier
    proxy: http://put_squid_server_here:3128

  mysql_connection:
```

```
    backend: mysql
    server: put_mysql_server_here
    username: put_username_here
    password: put_password_here

  sqlite_connection:
    backend: sqlite
    path: /your/work/path/to/cond_and_para.db
    schema_name: OfflineDB
```

The `CondDB` and `ParaDB` use an application name to retrieve the corresponding `DBIAPI` instance from the `DBISvc`. Separating the application name and the connection name makes it easier to configure different backends for the applications. The current implementation supports specifying multiple backends for an application, however, only the first backend will be used by the application. The same backend could be also used by the different applications. Following example shows that both services use the same frontier connection.

```
clients:
  conddb:
    - frontier_connection
  paradb:
    - frontier_connection
```

To switch the configurations easily, different YAML files are used in different use cases. One example is using frontier at different sites. In this case, the proxy of squid servers could be different. By using the different YAML files for different sites, the data production group could maintain the site configurations in a modular way. Another example is using SQLite at DAQ servers onsite. The corresponding backend only needs to specify the path of SQLite file.

5. Conclusions

A lightweight database interface had been implemented in JUNOSW, which is used to support the different database backends in several use cases during real data processing. Even though the database accessing could be changed in the future, the current design is still flexible to meet the requirements. The DBI is open source project on GitHub: `https://github.com/JUNO-collaboration/Database-dbi`.

Acknowledgments

This work is supported by National Natural Science Foundation of China (12025502, 11805223), the Strategic Priority Research Program of the Chinese Academy of Sciences (Grant No. XDA10010900), and Youth Innovation Promotion Association, CAS.

References

[1] Djurcic Z *et al.* (JUNO) 2015 (*Preprint* `1508.07166`)
[2] An F *et al.* (JUNO) 2016 *J. Phys.* **G43** 030401 (*Preprint* `1507.05613`)
[3] Abusleme A *et al.* (JUNO) 2022 *Prog. Part. Nucl. Phys.* **123** 103927
[4] Zhang X (JUNO) 2020 *EPJ Web Conf.* **245** 03007
[5] Huang X (JUNO) 2020 *EPJ Web Conf.* **245** 04030
[6] MySQL Web Site `https://www.mysql.com`
[7] Kosyakov S, Kowalkowski J, Litvintsev D, Lueking L, Paterno M, White S P, Autio L, Blumenfeld B J, Maksimovic P and Mathis M 2004 *14th International Conference on Computing in High-Energy and Nuclear Physics* pp 669–672
[8] Blomer J, Buncic P, Charalampidis I, Harutyunyan A, Larsen D and Meusel R 2012 *J. Phys. Conf. Ser.* **396** 052013
[9] SQLite Web Site `https://www.sqlite.org`
[10] DBI repository `https://github.com/JUNO-collaboration/Database-dbi`
[11] Huang X, Li T, Zou J, Lin T, Li W, Deng Z and Cao G (JUNO) 2017 *PoS* **ICHEP2016** 1051

[12] Zou J H, Huang X T, Li W D, Lin T, Li T, Zhang K, Deng Z Y and Cao G F 2015 *J. Phys. Conf. Ser.* **664** 072053
[13] The Official YAML Web Site `https://yaml.org`
[14] MySQL C-API `https://dev.mysql.com/doc/c-api/5.7/en/`