PUBS: Data Processing Software Framework

Kazuhiro Terao, kazuhiro@nevis.columbia.edu

December 30, 2014

Abstract

This document describes about `python/php/PostgreSQL` UBooNE Software (a.k.a. `pubs`) framework written for online data processing of the experiment. Each step of data processing, defined as a "project", carries a project-specific category of *state* and corresponding actions to deliver data to a next step. Projects' state are stored in `PostgreSQL` database server with a dedicated procedures for data queries. `pubs` provides a generic, `python` based software framework for development of "project" code. In addition, it provides necessary tools for project execution including a daemon program for parallel project execution and framework logger system. Monitoring of project execution is realized through a web-browser using a `php` based toolkit, which can be also used for experts to actively interface with currently running projects. Finally this document assumes you have a modest knowledge about `python` and common programming terminologies.

Contents

Chapter 1

Preparation

This chapter describes a prerequisite for using `pubs`. In particular we cover installation of database related external packages as it uses `PostgreSQL`. Find more about `PostgreSQL` on their home web page [1].

1.1 Prerequisites

In this section, we go over prequisites to use `pubs`. None of those require any particularly intensive computation power, and they can be all installed on a personal laptop.

1.1.1 Operating System And Software

As `pubs` does not require a compilation, it is expected to work on any operating system (OS). That being said, some functionality of underlying `python` does depend on the kernel. The author has not tested on Windows family OS, and he highly doubt it would work on it. There should be no problem on Unix and Linux family OS.

We use `git` for a software repository and version control. As mentioned, `pubs` use `php`, `PostgreSQL`, and `python`. A summary of minimaum version for these software is given below. These are oldest version that the author happened to find on his machines, so it is possible that things work on even older version of kernel/software. Please tell the author or update the document if you find such case.

- Supported Operating Systems
  - Mac OSX 10.5.8 (Darwin 9.8) or later with Xcode 3.1.4 or later
  - Linux 2.6.0 or later with `g++`4.1 or later
- Version Control Tool
  - `git` ... 1.7.1 or later
- Base Software Tools

- `php` ... v2.1 or later
- `PostgreSQL`... v9.1 or later with `contrib` for `HSTORE`
- `python` ... v2.7 or later (but no later than 3.0)
- `psycopg2` ... v2.5 or later

Both recent OSX and Linux distributions have `python` 2.7 and `git`. So we skip those items. The next section discuss how to install some rather uncommon parts including `PostgreSQL`, `HSTORE`, and `psycopg2`.

Both `PostgreSQL` and `python` has a very good documentation and FAQ resource on the web. You may find Ref.[2, 3] useful.

1.2 Installation of `PostgreSQL` And Extensions (`psycopg2` and `HSTORE`)

`psycopg2` is a `python` interface to `PostgreSQL` server (Ref.[4]). As far as the author can tell, this is not a part of `python` 2.7 standard module libraries. So you have to install it as an external product.

`HSTORE` is an external data representation introduced in `PostgreSQL` (Ref.[5]). It is a part of `PostgreSQL` `contrib` which is not included in a default set of libraries for compilation. We use this as well.

`pubs` itself works without a locally running `PostgreSQL` server. However, if you are to work on code development within `pubs` framework, it is handy to work on your own local `PostgreSQL` server because most likely you will be "messing" with the database and some test/dummy projects. So I spend some space here to share how to install `PostgreSQL` and `psycopg2` on your machine.

1.2.1 Words on Installation Method

If you use Linux, you might say "Well I would just use `apt-get` or `yum`!", or "I have a `fink`/`home brew`/`macport`!" if you use OSX family. There's nothing wrong to use `apt-get` or `yum` in principle, but the author has seen some OS specific default configuration imposed when one installs using those tools that is different from what one gets by compiling from the source. To be on the same page among readers with different OS, the author suggests to simply compile from source. And, by the way, in the author's biased mind, `fink` and `home brew` are even out of questions. Sorry.

But a compilation from source obviously requires a compiler. You can obtain this using your package distributor if you use Linux. Or `Xcode` if you use OSX.

1.2.2 Installing `PostgreSQL`

Go visit **http://www.postgresql.org/ftp/source/** to obtaib the source code of `PostgreSQL`. Currently the author sees 9.3.5 as the latest production version and is therefore recommended.

Click the source link and download, and untar it wherever you would like to install locally. In case you do not know how to untar it, if your file extension is `.tar.gz`, try:

```
> tar -xvfz postgresql-9.3.5.tar.gz
```

If it has simply `.tar` extension, try:

```
> tar -xvf postgresql-9.3.5.tar
```

which is often the case on OSX because it automatically un-zip it. Now in the extracted directory (in the example above it should be "postgresql-9.3.5"), you find `INSTALL` text file that tells you how to install. You can follow and type the followings.

```
> cd postgresql-9.3.5
> ./configure
> make
> sudo make install
```

Now from the next steps, `INSTALL` probably tells you to make a user `postgres` as the master of the database server. Assuming you are installing this locally on your own laptop/desktop and planning to be the master of the server forever, I suggest you do not make `postgres` and just use your account. That means to do this:

```
> sudo mkdir /usr/local/pgsql/data
> chown $USER /usr/local/pgsql/data
> /usr/local/pgsql/bin/initdb -D /usr/local/pgsql/data
> /usr/local/pgsql/bin/postgres -D /usr/local/pgsql/data >logfile 2>&1 &
```

If above command gives you a trouble, and if you give up solving on your own, feel free to contact the author to solve the mess as you followed his suggestion so far!

Now if above commands are successful, add the following lines in your `$HOME/.bashrc` or `$HOME/.bash_profile`:

```
export PATH=/usr/local/pgsql/bin:$PATH
export LD_LIBRARY_PATH=/usr/local/pgsql/lib:$LD_LIBRARY_PATH
```

If you use OSX, you also need this:

```
export DYLD_LIBRARY_PATH=/usr/local/pgsql/lib:$DYLD_LIBRARY_PATH
```

To enforce these shell environment variables, simply close your current shell and open a new one to work on.

1.2.3 Creating Database

Now that your server is running, let's create a database to work on. Among `pubs` developers, we use a database called `procdb` to develop our codes. You can make your own name, but sharing a database name probably makes it easier to communicate. So the author recommend to use this. Anyhow, you can create a database easily like this:

```
> createdb procdb
```

If you get an error message from your terminal complaining "no such file or command", then that is because you have not yet set your shell environment variables as discussed in the previous paragraph. If you get an error message regarding a permission, that is probably because you made `postgres` account and/or you have not changed a permission of `/usr/local/pgsql/data` directory, where both are discussed in the previous paragraph. Contact the author if you have a hard time solving these issues.

Now you should be able to log-in to your local psql server:

```
> psql -h localhost -d procdb
psql (9.3.5)
Type ‘‘Help’’ for help.
procdb=#
```

which is your `PostgreSQL` interpreter in `procdb` database.

1.2.4 Installing `HSTORE` Extensions

First, let's compile `HSTORE`. To do this, you need your shell environment variable set correctly as mentioned earlier. After you follow those steps, go back to your `PostgreSQL` source directory, then try:

```
> cd contrib/hstore
> make
> sudo make install
```

This should compile `HSTORE` for you. Again, if you have a problem, contact the author. Now this only made a library of `HSTORE` available to your server, and it does not mean `HSTORE` became available in your database. You have to enable `HSTORE` in your database explicitly. Try this:

```
> psql -h localhost -d procdb
procdb=# CREATE EXTENSION HSTORE;
```

Do not forget ";" in the end. This installs `HSTORE` data type and all related functions to your `procdb` database.

1.2.5 Installing `psycopg2`

You can download `psycopg2` source code from **https://pypi.python.org/pypi/psycopg2**. `psycopg22` uses `distutils`, which is a standard `python` package build tool. To compile, simply type:

```
> python setup.py build
> sudo python setup.py install
```

Note, again, this requires you to have set the shell environment variables like `$PATH` and `$LD_LIBRARY_PATH`. Nevertheless this is all you need to do. To test whether this worked or not, try:

```
> python
>>> import psycopg2
```

If this does not throw an error message, you're good. One further test you can do:

```
> import os
>>> host = 'localhost'
>>> user = os.environ['USER']
>>> db = 'procdb'
>>> pass = ''
>>> psycopg2.connect(host=host,user=user,database=db,password=pass)
```

If this does not throw an error message, you're beautiful. This means your local `psycopg2` library is correctly linked against your `PostgreSQL`.

1.3 Install And Run `pubs`

`pubs` is kept at a public github repository. One can access the respotiroy through a web browser as well **https://github.com/drinkingkazu/pubs**. In this section we cover how to obtain `pubs` and configure to use the tools. We also cover how to clean up your local database server (as far as `pubs` concerns).

1.3.1 Obtain And Configure `pubs`

First of all, make a git account. Then let the author know about your account so he can add you as a collaborator, and you will have a write permission. This makes your life much easier.

If you have a write permission, you can checkout as shown below:

```
> git clone git@github.com:drinkingkazu/pubs pubs
> cd pubs
> git checkout devel
```

If you have only a read permission, try the following instead:

```
> git clone https://github.com/drinkingkazu/pubs pubs
> cd pubs
> git checkout devel
```

Above operations are needed per making a clean checkout from git repository. You do not need to do them if you are to work on previously checked out repository.

Now, per login, you have to source some configuration scripts as shown below.

```
> source config/setup.sh
> source config/personal_conf.sh
```

The first script important shell environmnet variables to use `pubs`. In particular, after running the script, you should see a shell environment variable `$PUB_TOP_DIR` is set:

```
> echo $PUB_TOP_DIR
```

which should be pointing at the top directory of `pubs`. You can execute this script from anywhere (i.e. it does not need to be run from `pubs` top directory and it will figure out the correct path).

The second script should be really made per user (so you can make your own). This shell script sets some environment variables that are used to connect `PostgreSQL` server. Take a look at it. How each environment variable is used should be trivial from their name. If you have a specific user name, password, server DNS, database name etc. configured differently than default, make your own configuration script!

1.3.2 Setting Up `PostgreSQL` Server For `pubs`

The `PostgreSQL` server interaction model in `pubs` is to use a pre-defined set of functions to make data queries rather than allow users to come up with a random (and probably wild) bare query strings. These functions are defined in `SQL` scripts:

- db_scripts/master_creation.sql
- db_scripts/procdb_functions.sql

Uploading these functions to your server is a one-time operation, and you can do:

```
> psql -h localhost -d procdb -f db_scripts/master_creation.sql
> psql -h localhost -d procdb -f db_scripts/procdb_functions.sql
> psql -h localhost -d procdb -f db_scripts/initialize_procdb.sql
```

Now, someday you might be in a situation such that something went wrong on the server and you want to reset all. Well, it is a beauty of working on localhost of your machine: you can go wild and reset *everything*. Here's how you can do this:

```
> dropdb procdb
> createdb procdb
> psql -h localhost -d procdb -f db_scripts/master_creation.sql
> psql -h localhost -d procdb -f db_scripts/procdb_functions.sql
> psql -h localhost -d procdb -f db_scripts/initialize_procdb.sql
```

Try this solution if something weird happened to your `procdb` database. But just remember one thing: this operation removes *everything* including all project tables. Be responsible ;)

1.3.3 Running `pubs`

Make sure you have read and followed Sec.1.3.1 and 1.3.2. Now you should be ready to execute a test script to see things are set up correctly. Try:

```
> cd $PUB_TOP_DIR
> python dstream/ds_daemon.py
```

This should start-up a process that continues indefinitely making `stdout` messages every second. The script you are running is actually one of core tools in `pubs` that can execute all projects that you will develop (or developed by others) within `pubs`. For now, this part is complete. We will come back to in the later chapters.

Chapter 2

Basic of pubs

This chapter describes about building blocks of `pubs` framework. Sec.2.1 covers the data process management model and the framework design. Three following sections 2.2, 2.3, 2.4 cover `PostgreSQL`, `python`, and `php` components as building blocks of the framework respectively. More practical development how-to's are discussed in the next chapter.

2.1 Data Processing Model

`pubs` is a data processing software framework that supports a specific data process model implementation by application developers. In particular, it has following features:

- Use `PostgreSQL` for process management
- Supports `python` code development for implementation
- Provides `php` toolkit for application management and monitoring

This section describes a generic idea of `pubs` data processing model.

2.1.1 Project: A Unit Task of Data Processing

A typical model of data processing is a well defined chain of processes in which a particular process is triggered by a completion or an initiation of another process. For instance, the end of running DAQ may trigger a file transfer protocole to move a raw data file to a storage server. In `pubs`, each of such processes is called a *project*.

Project Status

Each `pubs` project carries a specific *status* code represented by an integer. A unit of status is defined by, in case of MicroBooNE, a unique combination of four integeres: run, sub-run, *sequence* (or simply *seq*), and the project version number. This combination is referred to as *TaskID* in this document. A run and sub-run numbers defines a boundary of data taking defined by a DAQ as you might guess. A sequence number, on the other hand, is a possible sub-set of

run and sub-run number combination, and is defined by a project. It is there to support some projects that may need to sub-divide a process to deal with a particular DAQ run. As its name says, the project version is an integer representing different version of the same project.

Each `pubs` project carries a status for each TaskID. Special status code 1 is reserved to represent the initial status for any project. Similarly, code 0 is reserved to represent the completion status for any project. Any other integer values may be used to represent various status which meaning may be defined by a project.

Project Version

One important feature of data processing framework is an ability to roll-back and re-process some older data files. Reprocessing of data is supported in `pubs`, but it requires to change a project version number. The basic assumption for forcing the version update for re-processing is that something must have changed to roll back and re-process data. Figure2.1 shows an example of reprocessing associated with a specific project's version update.

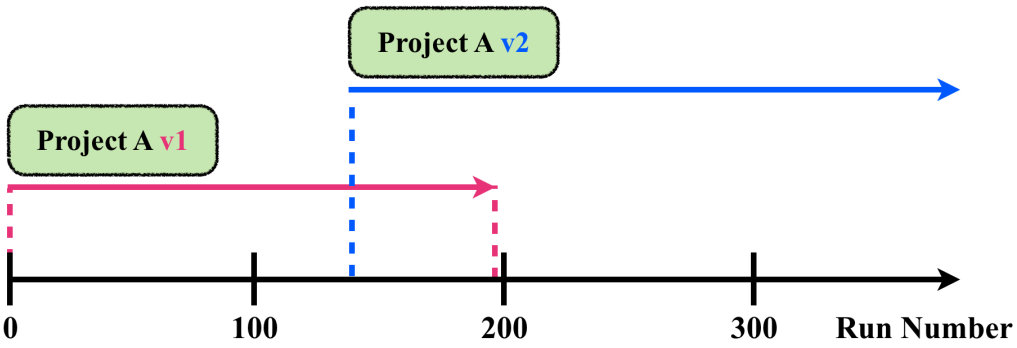

Figure 2.1: Example diagram showing how project version may be used. The horizontal axis shows the time in the unit of a run number (sub-run number ignored for simplicity). The diagram shows that the 1st version of project A processing data up to around run 200. Then the updated 2nd version starts re-processing runs just before around run 150 and takes over to process all future runs.

Project Table

Each project status is stored in `PostgreSQL` database table with a name same as the project's name, called a project table. For this reason, **a project name must be unique**. A project table contains several information per unique combination of run, sub-run, seq numbers, and they are shown below with corresponding `PostgreSQL` data type.

- Run ... INT ... DAQ run number
- SubRun ... INT ... DAQ sub-run number

- Seq ... SMALLINT ... project sequence number
- Status ... SMALLINT ... project status code
- Data ... TEXT ... Generic "data" associated with a particular run, sub-run combination
- ProjectVer ... SMALLINT ... a version number for a project

A project may record some "data" in string representation for each TaskID. However, it is recommended to avoid to store such information when possible as this makes the table size to become larger. Finally, as mentioned before, a project is allowed to have a different version number.

Project Execution Model

Each project is expected to have a single `python` script to be executed. This `python` script should fetch an array of target TaskID from the database based on a status code. Depending on the status code, then, the script may take a certain action, and possibly update the status code in the database upon success. In short, the status code is used as a trigger for a specific action by projects.

Now, information needed to execute each such script is called *project information*, and is stored in a separate database table called the *ProcessTable*. In the next section, we discuss about this table and also a machinary to automatically execute a project's `python` script using a daemon tool.

2.1.2 ProcessTable And Daemon Execution

Information about individual project is stored in a dedicated database table called "ProcessTable". Unlike project tables that exist one per project, this is a unique table in the database that holds all projects' information. The table schema is shown below:

- ID ... SERIAL ... a unique integer key for each project
- Project ... TEXT ... the name of a project
- ProjectVer ... SMALLINT ... the version number of a project
- Command ... TEXT ... a project execution command to be run
- Frequency ... INT ... latency in seconds between each execution of a project
- EMail ... TEXT ... email contact(s) in case of a trouble
- StartRun ... INT ... the first run-number to be processed
- StartSubRun ... INT ... the first sub-run number to be processed
- Resource ... HSTORE ... a dynamic string-to-string map data container

- Enabled ... BOOLEAN ... a boolean flag for project execution (only if it is true)
- Running ... BOOLEAN ... a boolean flag enabled during project execution
- LogTime ... TIMESTAMP ... a time-stamp logged per entry update

where, among many self-descriptive columns, "Resource" column holds HSTORE type variable that can be used to store any project-specific information. Such information includes, for example, the path to a specific data directory and expected file name format that requires run and sub-run numbers such as

`Run%05d_SubRun%03d.bin`.

with which your project can figure out the expected file name for a given TaskID (i.e. run and sub-run numbers).

Note that storing such information in ProcessTable is much lighter than storing the actual filename in a project table's "Data" column because the "Data" column stores information for every single TaskID while ProcessTable entry is made only one per project. Again, store as much information as possible in ProcessTable and avoid using "Data" column of a project table when possible.

Daemon: Master Scheduler

An execution of a project should be as simple as:

```
> python my_project.py
```

where my_project.py is the executor of a project. In `pubs`, however, there is a simple daemon project management tool that periodically looks up the ProcessTable and executes the enabled projects in parallel. So it is a lot like `cron` in Unix/Linux but with a process manager aspect.

The daemon process is responsible for project management: it keeps track of history of running each project, and it ensures its resoure usage. The project information is updated every 120 seconds (configurable) and synchronized with the ProcessTable contents in the database. An expert can impose a change in project information without interrupting the daemon process. Another important role of the daemon is to keep the integrity of the DAQ's run table with all enabled projects' table. Once in every 300 seconds (configurable), the daemon process synchronizes run and sub-run number entries in each project table and match with the DAQ run table.

That being said, it is extremely important that each project is a truly modulated action, and does not depend on an execution by the daemon. In other words, one should be able to always run a project by simply invoking from a command line. This allows us to take any emergency treatment when, somehow, the daemon procedure cannot be used.

2.1.3 Web Monitoring And Process Management

Figure2.2 shows a brief data flow from and into the `PostgreSQL` database. As described in Sec.2.1.1 and 2.1.2, each project and daemon accesses the database server individually. In addition to the project execution framework, `pubs` provides `php` based monitoring and management

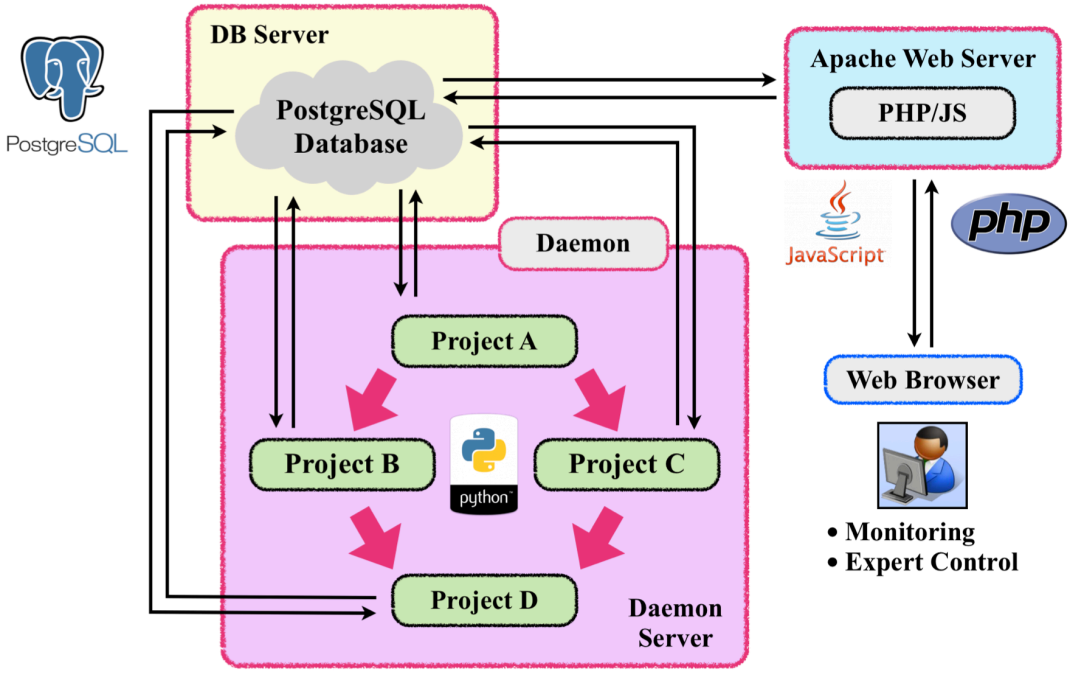

Figure 2.2: An abstract drawing that shows how data flows into and from the `PostgreSQL` database within `pubs` framework. Each project and daemon process all talk to the database server. The `php` web-interface provides monitoring and project management tools.

interface through a web server machine and end-user's web browser application. Accordingly there is a database connection between the web server and `PostgreSQL` server machines.

The capability to monitor each projects' status through a web browser is quite useful for a routine inspection, such as shifters' check list. Similarly a formatted control pannel for managing project execution is useful for experts to take action without logging into the machine. The details of `php` implementation is quite experiment specific, and the MicroBooNE usage will be discussed in Ch.3.

2.2 `PostgreSQL` Functions

Now that we spent many pages to discuss about the `pubs` model, let's talk about something real and practical. This section presents a list of `pubs` functions implemented on the `PostgreSQL` server. About a half of them are for experts' use (in fact mostly for daemon and automated scripts since human hands are one of last things to be trusted), and the other half is for project scripts to use.

If you are a project code developper and do not find a function of your need, please contact the author and he will be more than happy to assist how the existing function may solve the problem or implement a brand new function to make your life easier.

2.2.1 Project Information/Status Query

These are functions that can be used by projects upon execution. That being said, however, it is **strongly recommended to use `python` API within `pubs` to execute these functions**. They should not be executed from `PostgreSQL` interpreter or directly executing from an SQL script. If the list lacks any function needed for a project execution, please contact the author with a request. Functions and corresponding `python` API will be provided.

- **DoesTableExist(name TEXT)**
  - Checks if a table of the *name* exists or not in the database by checking the administrative master table. The table name is required to be in lowercase (there is no uppercase vs. lowercase in distinction among `PostgreSQL` server objects).
- **DoesProjectExist(name TEXT)**
  - Checks if a project with the *name* exists or not in the database. In addition to DoesTableExist(), this function checks if a specified project exists or not.
- **GetRunTimeStamp(Run INT, SubRun INT)**
  - A function to retrieve the run start and end time stamp.
- **ProjectResource(name TEXT)**
  - Returns a project resource (information needed for an execution) for a specified project name.
- **IncreaseProjSequence(name TEXT, run INT, subrun INT, nseq SMALLINT, status SMALLINT)**
  - Increase number of sequence count in the specified project table for the specified run/sub-run number combination. Input status code is used for all newly created TaskIDs.
- **UpdateProjStatus(name TEXT, run INT, subrun INT, seq SMALLINT, status SMALLINT, data TEXT)**
  - Update the specified project's status for the specified TaskID. At the same time, a TaskID specific data can be also stored although that is not necessary (by default the last argument is set to NULL).
- **GetProjectData(name TEXT, run INT, subrun INT, seq SMALLINT)**
  - Retrieve project data for a specified TaskID. Only accessible to The data from the latest version number to avoid a confusion (and hence version number cannot be specified).
- **GetRuns(name TEXT, status SMALLINT)**

  - Returns a table of TaskID (run, sub-run, seq., project-version) for which the specified project carries the specified status code.

- **GetRuns(TEXT[]::ARRAY, SMALLINT[]::ARRAY)**

  - Similar to GetRuns and it returns a table of run/sub-run number combinations for which all specified projects in the first argument carry specified status code in the second argument. This function is useful to obtain a list of run/sub-run numbers across multiple project tables for specific combination of status code. Because a sequence number is project dependent, it returns run/sub-run for which all belonging sequence status uniquely matches with the specified status code.

2.2.2 Functions For Project Management

These are functions to be used by daemon process to maintaine/running the projects. In principle these should not be used by a project execution.

- **RemoveProject(name TEXT)**

  - Properly remove a project: drop a project table and remove the project information entry from the ProcessTable.

- **ListProject()**

  - List all projects with the latest version number from ProcessTable.

- **ListEnabledProject()**

  - List currently enabled project information with the latest version number from the ProcessTable.

- **DefineProject(name TEXT, command TEXT, frequency INT, email TEXT, start_run INT, start_subrun INT, resource HSTORE, enabled BOOLEAN)**

  - A function to define a new project. It takes in project information and registers into the ProcessTable. It also calls **MakeProjTable** function to create a project table.

- **MakeProjTable(name TEXT)**

  - Function dedicated to create a project table. This function is to be called by **DefineProject** and not to be called by hand!

- **UpdateProjectConfig(name TEXT, command TEXT, frequency INT, email TEXT, resource HSTORE, enabled BOOLEAN, version INT)**

  - A function to alter and update project configuration. As seen in the function arguments, start run/sub-run number cannot be altered by design.

- **ProjectVersionUpdate(name TEXT, command TEXT, frequency INT, email TEXT, run INT, subrun INT, resource HSTORE, enable BOOLEAN)**
  - Increment the project version number and store new project information. Unlike **UpdateProjectConfig**, this function can register any project information as there will be a distinct row to be inserted in the ProcessTable.
- **GetVersionRunRange(name TEXT)**
  - For a specified project name, returns multiple result sets each representing a specific run number range with the corresponding project version number.
- **InsertIntoProjTable(name TEXT, run INT, subrun INT)**
  - Insert a new run/sub-run number entry into a project table with the default status code of 1. The latest version number for the subject run/sub-run is also taken from the ProcessTable.
- **OneProjectRunSynch()**
  - Make sure one particular project table has run/sub-run numbers that currently appears in the MainRun table and above the specified run/subrun numbers in the argument.
- **AllProjectRunSynch()**
  - Make sure all project table has run/sub-run numbers that currently appears in the MainRun and above the specified start run/sub-run numbers in the project information.
- **ProjectInfo(name TEXT, ver INT)**
  - Returns project information for a specified version number. By default the version number does not need to be specified. If not given, it is set to the latest version number. This function is used to run a project via daemon.

2.2.3 Admin Functions

Functions prepared for the top-level administrative purposes. These functions should be executed by database admins only.

- **RemoveProcessDB()**
  - "Properly" remove *everything*. This function drops all projects registered in ProcessTable using **RemoveProject** function. Then it drops an empty ProcessTable.
- **CreateProcessTable()**
  - A simple function to create the ProcessTable.

- **CreateTestRunTable()**
  - A function to create "fake" MainRun table. This is for development work, and not for an official operation. In the official production, MainRun table is slave-copied from the configuration database automatically.
- **InsertIntoTestRunTable(Run INT, SubRun INT, TimeStart TIMESTAMP, TimeEnd TIMESTAMP)**
  - A function to insert a new entry into the "fake" MainRun table. This is not meant to be used for the offial production.
- **FillTestRunTable(NRuns INT, NSubRuns INT)**
  - A function to fill the "fake" MainRun table with multiple entries at once. It fills the table with NRuns, each with NSubRuns.
- **CheckDBIntegrity()**
  - Returns a boolean after checking the process DB integrity. In particular it checks if ProcessTable exists or not, and then checks if all projects registered in ProcessTable have own project tables.

2.3 `python` Software Framework

The "software framework" part of `pubs`, which provides the code base for application development, is really in `python`. This section describes `python` tools in `pubs` that can be used to develop a project execution code and set up the data processing chain of multiple projects.

There are three (somewhat) big `python` modules in `pubs`:

- `pub_util` ... basic framework tools
- `pub_dbi` ... generic database interface based using `psycopg2`
- `dstream` ... data processing framework toolkit

A project code developer interfaces with `dstream` directly while that itself depends on basic tools defined in `pub_util` and `pub_dbi`. We go over each of these in the following sections.

2.3.1 `pub_util`: Basic Toolkit

This module introduce 3 objects: `pub_logger`, `pub_smtp`, and `pub_exception`. They are framework logging tool, email sender function via SMTP, and a base exception class definition.

pub_logger ... Logging Module

This is the framework logger tool, and uses a popular python's logging module. pub_logger is a factory class that can instansiate an individual logger instance with a specific message format and a choice of stream: either stdout/stderr or output file stream.

Each logger instance created by pub_logger factory has a unique name, and can be instantiated by a factory function call:

```
>>> pubs_logger.get_logger('my_logger')
```

for a logger named "my_logger". pub_logger keeps track of all created loggers in its class variable _loggers. When there is a request for a logger with the same name created in the past, it returns the same instance.

```
>>> from pub_util import pub_logger
>>> pub_logger.get_logger('a')
[ INFO ] pub_logger (L: 81 ) >> {_add_logger} OPENED LOGGER a
<logging.Logger object at 0x10b903f90>
>>> pub_logger.get_logger('a')
<logging.Logger object at 0x10b903f90>
>>>
```

As you might expect in any similar tool, pub_logger has several message levels: debug, info, warning, error, and critical. The default message level is set via shell environment variable $PUB_LOGGER_LEVEL, which is automatically set in setup.sh configuration script. You may change the level if you wish. You do not have to change the configuration script, but instead just change the shell environment variable in any way you want (for instance by hand on your terminal instead of sourcing a script). The set shell environment value is parsed in pub_util/pub_env.py script to an appropriate value. Similarly, the stream destination (either stdout or file stream) is set via shell environment variable $PUB_LOGGER_DRAIN, again set automatically in setup.sh. In case the drain is chosen to be a text file stream, $PUB_LOGGER_FILE_LOCATION environment variable's value is used as the log file location.

Each message level has a dedicated logger function call to parse an output message through your logger. Here is an example of formatted output :

```
>>> a.debug('This is debug')
[ DEBUG ] <stdin> (L: 1 ) >> {<module>} This is debug
>>> a.info('This is info')
[ INFO ] <stdin> (L: 1 ) >> {<module>} This is info
>>> a.warning('This is warning')
[ WARNING ] <stdin> (L: 1 ) >> {<module>} This is warning
>>> a.error('This is error')
[ ERROR ] <stdin> (L: 1 ) >> {<module>} This is error
>>> a.critical('This is critical')
[ CRITICAL] <stdin> (L: 1 ) >> {<module>} This is critical
```

The logger specifies the message level, and prints out three more information in addition to the sent message by the caller. The first '<stdin>' tells where the message is sent from. '(L: 1)'

tells us which line in the caller's module code this function is called from. Then '{<module>}' tells us the name of the caller's module. In the above example, this is called from the main, and hence it is not really useful. However, these information help us to track down problems easily as you can identify where each function call is made. For instance, running `ds_daemon.py` to test the installation (see Sec.1.3.3), you have probably seen this message:

```
[ DEBUG ] ds_daemon (L: 128) >> {load_projects} Updating project dummy_daq ...
```

This menas that a logger function "debug" was called by a function `load_projects` and the exact location is in line number 128 of the module code `ds_daemon.py`.

`pub_smtp` ... Simple SMTP Protocole

`pub_smtp` is a simple function that uses the Simple Mail Transfer Protocole to send an email message. One can specify the recipients, subject and text message to be sent. By default, the SMTP account, server address (and port), and password are taken from, again, shell environment variables: `$PUB_SMTP_ACCT`, `$PUB_SMTP_SRVR`, and `$PUB_SMTP_PASS` respectively. These are currently set to an email account created for a common use. You may change this to your account if you wish, or we use an official account when in production.

`pub_exception` ... Simple Exception

This is an exception class that inherits from the base `python Exception`. It is nothing special but takes an error message in the constructor argument. Only purpose is to have a common exception within `pubs` so that we can catch `pubs` specific exception.

2.3.2 `pub_dbi`: Generic DB Interface

`pub_dbi` module defines client tools used to interface with `PostgreSQL` database server. In particular it contains following classes

- `pubdb_conn` ... for connecting database server
- `pubdb_data` ... class that encapsulates connection information
- `pubdb_exception` ... exception dedicated for `pub_dbi` module
- `pubdb_reader` ... "read-only" database query API instantiated by `pubdb_conn`
- `pubdb_writer` ... "read/write" database query API instantiated by `pubdb_conn`

Note that a project code developer should use a wrapper API that is under `dstream`. This section covers base components that are usually hidden in behind. OK let's go through them. Hope I can contribute to your good sleep.

`pubdb_conn` database connection factory

This is a factory class that connects to the database using `psycopg2` and generate a connection cursor (`psycopg2.cursor`) for `pubdb_reader` and `pubdb_writer` APIs.

2.3.3 `dstream`: Data Processing Framework

2.4 `php`-based Web Interface

Chapter 3

MicroBooNE Implementation

Bibliography

[1] Postgresql. *Web*, http://www.postgresql.org.

[2] Python v2.x documentation. *Web*, https://docs.python.org/2/.

[3] Postgresql document repository. *Web*, http://www.postgresql.org/docs/.

[4] Postgresql adaptor for python. *Web*, http://initd.org/psycopg/.

[5] Hstore in postgresql. *Web*, http://www.postgresql.org/docs/9.0/static/hstore.html.