Process Tracker

Release 0.7.0 - beta

Dec 19, 2019

Contents:

ProcessTracker is a framework for managing data integration processes.

The home for ProcessTracker is on GitHub .

The goal of ProcessTracker is to provide an easy to use framework for auditing and managing data integration processes and their data. This framework has been developed over many iterations and years.

While the initial focus has been on process and data extract management, other features will likely be added and improved.

We are always looking for new feature requests, bugs, and other contributions! To learn more on how to contribute, please refer to our *Contributing* page. To see what is on the roadmap, please check out our Issue Board.

ProcessTracker is being developed for many different languages/tools. Please refer to the *Implementations* page.

CHAPTER 1

Introduction

This section provides high level details on ProcessTracker and it's various components.

1.1 ProcessTracker - The Framework

1.1.1 Why ProcessTracker?

Data integration tools and frameworks do not have a universal way to work together, track processes and interdependencies, and the data those processes utilize. While an enterprise scheduler can help with dependency management, it won't necessarily help with a lot of the other auditing and process management. Enter ProcessTracker.

ProcessTracker is a framework for managing data integration processes and their data in a central place, with the goal being to be as tool agnostic as possible so that it can be used in any data integration environment, regardless of tool stack. As long as the ProcessTracker methodology is followed, it can be adapted to be used by any data integration tool or data driven process - from ETL/ELT tools to tools like R, Spark, and beyond.

It it our hope that other implementations will be released over time for the various tools/languages that are used. As we either build them, or find out about them, we will list them below.

1.1.2 ProcessTracker and ExtractTracker

There are two primary functions of the ProcessTracker framework. ProcessTracker manages and audits data integration processes while ExtractTracker manages and audits the data extracts that are utilized between processes. For tool implementations the same concepts apply even though they are not implemented in quite the same way as the code package implementations.

Implementations

As stated elsewhere, the ultimate goal for ProcessTracker is for any data integration or data processing tool/framework/code to be able to take advantage of universal process tracking. With that in mind, the following frameworks have either been developed as part of this project or created by the community.

Python Implementation

This section provides details on the Python implementation of ProcessTracker.

For issues or feature requests, please check out the Python Issue Board.

Overview

Below is an overview of how to work with the Pythom implementation of ProcessTracker.

How Does It Work?

The Python ProcessTracker implementation consists of two core classes: ProcessTracker and ExtractTracker. Let's start with an example for ProcessTracker. Consider that we are running a pyspark job:

```
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local")\
                            .appName("Lahman Teams Load")\
                            .config("spark.executor.memory", "6gb")\
                            .enableHiveSupport()\
                            .getOrCreate()
```

This builds a pyspark session. To register it within ProcessTracker, we need to modify our pyspark job:

```
from processtracker import ProcessTracker
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local")\
                            .appName("Lahman Teams Load")\
                            .config("spark.executor.memory", "6gb")\
                            .enableHiveSupport()\
                            .getOrCreate()

process_run = ProcessTracker(process_name='Lahman Teams Load'
                                     , process_type='Stage Load'
                                     , actor_name='New ProcessTracker User'
                                     , tool_name='Spark'
                                     , sources='Lahman Baseball Dataset')
```

That will register the process, actor (the person or environment that's doing the processing), tool, and source if they have not been registered before as well as create a new process run. If everything works as expected (because we live in a perfect world, right?) the process run can be finished with a simple command:

```
process_run.change_run_status('completed')
```

This will update the run record to show that it is completed, which will allow dependent processes or new runs of the same process to run.

Now let's say we're actually working with staged data files - either producing them or using them from another process. This is where ExtractTracker will come in handy. Consider that we have the same pyspark process from earlier, which is loading the Teams.csv file:

```
data_set = spark.read.format("csv").option("header", "true") \
                     .load("~/baseballdatabank-master_2018-03-28/baseballdatabank-
↪master/core/Teams.csv")
```

(continues on next page)

(continued from previous page)

Now it would be good to know that not only did this process use the Teams.csv file, but what if another process had already loaded it? Let's modify our pyspark job to also register the Teams.csv file:

```
from processtracker import ExtractTracker, ProcessTracker
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local")\
                            .appName("Lahman Teams Load")\
                            .config("spark.executor.memory", "6gb")\
                            .enableHiveSupport()\
                            .getOrCreate()

process_run = ProcessTracker(process_name='Lahman Teams Load'
                                     , process_type='Stage Load'
                                     , actor_name='New ProcessTracker User'
                                     , tool_name='Spark'
                                     , source_name='Lahman Baseball Dataset')

extract = ExtractTracker(process_run=process_run
                              , filename='Teams.csv'
                              , location_name='Lahman Baseball Databank 2018'
                              , location_path='~/baseballdatabank-master_2018-03-28/
↪baseballdatabank-master/core/')

extract.change_extract_status('loading')

data_set = spark.read.format("csv").option("header", "true") \
                      .load("~/baseballdatabank-master_2018-03-28/baseballdatabank-
↪master/core/Teams.csv")
```

Now our Teams.csv is registered if it wasn't already, along with the location where Teams.csv can be found. The association between the process run and the extract is also tracked. Then to show that we are doing something to the file other than creating it (the extract gets an initial status of initializing), we change it's status to loading. When we are done with the extract, we simply change it's status again and complete the process run. Our example pyspark process now looks like:

```
from processtracker import ExtractTracker, ProcessTracker
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local")\
                            .appName("Lahman Teams Load")\
                            .config("spark.executor.memory", "6gb")\
                            .enableHiveSupport()\
                            .getOrCreate()

process_run = ProcessTracker(process_name='Lahman Teams Load'
                                     , process_type='Stage Load'
                                     , actor_name='New ProcessTracker User'
                                     , tool_name='Spark'
                                     , source_name='Lahman Baseball Dataset')

extract = ExtractTracker(process_run=process_run
                              , filename='Teams.csv'
                              , location_name='Lahman Baseball Databank 2018'
                              , location_path='~/baseballdatabank-master_2018-03-28/
↪baseballdatabank-master/core/')
```

(continues on next page)

(continued from previous page)

```
extract.change_extract_status('loading')

data_set = spark.read.format("csv").option("header", "true") \
                     .load("~/baseballdatabank-master_2018-03-28/baseballdatabank-
↪master/core/Teams.csv")

<load Teams.csv and do other things>

extract.change_extract_status('loaded')
process_run.change_run_status('completed')
```

Command Line Tool

The command line tool is designed to help with adding lookup values as well as work with the data store backend. Once the processtracker package is installed, the following options will be available from the command line.

Data Store Setup

When first setting up processtracker, the CLI can be leveraged to create all the data store objects::

```
processtracker setup
```

If the data store has already been created, but you want to start fresh, pass the overwrite option::

```
processtracker setup --overwrite True
```

There is also a shorthand version::

```
processtracker setup -o True
```

Lookup Objects

The following lookup objects can be modified or created via the command line:

- Actor
- Cluster
- Cluster Process
- Error Type
- Extract Status
- Process Dependency
- Process Status
- Process Type
- Source
- Tool

Please note, any values added during initialization will not be able to be dropped or modified.

Create

This allows for new lookup records to be added.:

```
processtracker create --topic Actor --name 'New Actor'
```

Shorthand can also be used::

```
processtracker create -t Actor -n 'New Actor'
```

Update

This allows for existing lookup records to be modified.:

```
processtracker update --topic Actor --initial-name 'New Actor' --name 'Modified Actor'
```

Shorthand can also be used::

```
processtracker update -t Actor -i 'New Actor' -n 'Modified Actor'
```

Delete

This allows for existing lookup records to be deleted.:

```
processtracker delete --topic Actor --name 'New Actor'
```

Shorthand can also be used::

```
processtracker delete -t Actor -n 'New Actor'
```

Cluster Process

Cluster Process relationships can be added and removed via the CLI. These are a bit more complicated to work with than the lookup objects.

Create

To create a new cluster process relationship, use the create command as noted above, but with other parameters.:

```
processtracker create --topic "cluster process" --cluster "My Cluster" --child "My␣
↪Process Name"
```

Shorthand can also be used.:

```
processtracker create -t "cluster process" --cluster "My Cluster" -c "My Process Name"
```

Delete

To delete a cluster process relationship, use the delete command as noted above, but with other parameters.:

```
processtracker delete --topic "cluster process" --cluster "My Cluster" --child "My␣
↪Process Name"
```

Shorthand can also be used.:

```
processtracker delete -t "cluster process" --cluster "My Cluster" -c "My Process Name"
```

Process Dependency

Process dependencies can be added and removed via the CLI. These are a bit more complicated to work with than the lookup objects.

Create

To create a new process dependency, use the create command as noted above, but with other parameters.:

```
processtracker create --topic "process dependency" --parent "My Parent Process Name" -
↪-child "My Child Process Name"
```

Shorthand can also be used::

```
processtracker create -t "process dependency" -p "My Parent Process Name" -c "My␣
↪Child Process Name"
```

Delete

To delete a process dependency, use the delete command as noted above, but with other parameters.:

```
processtracker delete --topic "process dependency" --parent "My Parent Process Name" -
↪-child "My Child Process Name"
```

Shorthand can also be used::

```
processtracker delete -t "process dependency" -p "My Parent Process Name" -c "My␣
↪Child Process Name"
```

Password Encryption

Data store passwords can be encrypted using the CLI tool. This encryption is NOT cryptographically secure! This method is just so plain text passwords are not stored in the configuration file.:

```
processtracker encrypt --password "MySecretPassword"
```

This will return back the encrypted cypher::

```
Encrypted password is now:  Encrypted wqfCvsKKwpzCrsOYw4rCvsKBwrHCrMOawr_DlcOZwro=
When storing in your config file, please be sure to include 'Encrypted ' as well as␣
↪the hash.
```

As the message states, take the password, starting with the 'Encrypted ' and paste it into your config file. Again, this is not cryptographically secure. This only helps obscure your password.

Move Process Run From 'On Hold' Status

Process runs that are in 'on hold' status can now be updated to 'completed' thru the CLI tool. To update a process run that is in 'on hold', run the command::

```
processtracker update -t "process run" -n "My Process Name"
```

Note that process runs can only have their status updated. They can not be created, deleted, or updated otherwise.

ExtractTracker - Python

Below is a deeper dive of the capabilities of the Python implementation ExtractTracker submodule. Note that ProcessTracker MUST be used in conjunction with ExtractTracker.

Registering Extracts

Once the process run has been registered, an extract can be registered, provided the following variables are set.:

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                                    , process_type='Stage Load'
                                    , actor_name='New ProcessTracker User'
                                    , tool_name='Spark'
                                    , source_name='Lahman Baseball Dataset')

extract = ExtractTracker(process_run=process_run
                                , filename='Teams.csv'
                                , location_name='Lahman Baseball Databank 2018'
                                , location_path='~/baseballdatabank-master_2018-03-28/
↪baseballdatabank-master/core/')
```

Those variables will be used to populate the data store backend as explained in the following table:

Changing Extract Status

As extract files are used within a process run, their status will need to be modified.:

```
extract.change_extract_status(status='loading')
```

Custom extract status can be entered, but the default status types must be used for ProcessTracker to know what to do with files. As long as the file's status is eventually changed to one of those then the process flow will continue.

Extract Dependencies

Extracts can have dependencies between each other, just like processes. To add dependencies as part of the data pipeline, use the add_dependency function.:

```
extract.add_dependency(dependency_type='parent', dependency=parent_extract)
```

Child dependencies can also be registered.:

```
extract.add_dependency(dependency_type='child', dependency=child_extract)
```

Extract Audit Information

Audit information can be collected on individual extracts.

Setting Extract Data's Low and High Dates

If an extract has a date field, it can be used to store the extract file's lowest and highest datetimes.:

```
extract.set_extract_low_high_dates(low_date="1900-01-01 00:00:00", high_date="2019-12-
↪31 00:00:00")
```

Dates can be refreshed as the data is processed, or the low and high dates can be predetermined and passed to Extract-Tracker before finishing the processing of the file by changing it's status.

Two types of dates are tracked: dates on write and dates on load. set_extract_low_high_dates has a default type of 'load'. To change it, just set audit_type.:

```
extract.set_extract_low_high_dates(low_date="1900-01-01 00:00:00", high_date="2019-12-
↪31 00:00:00", audit_type='write')
```

Setting Extract Record Count

If tracking number of records is required, set_extract_record_count will keep the status of the number of records per file, once it is provided the record count.:

```
extract.set_extract_record_count(num_records=792)
```

As with low and high dates, two types of counts are tracked: write and load. set_extract_record_count has a default type of 'load'. To change it, just set audit_type.:

```
extract.set_extract_record_count(num_records=792, audit_type='write')
```

Extract Helpers

There are also a few helpers available to ExtractTracker objects to assist in using extracts:

Location

The extract's location object can be retrieved by using::

```
extract = ExtractTracker(process_run=process_run
                              , filename='Teams.csv'
                              , location_name='Lahman Baseball Databank 2018'
                              , location_path='~/baseballdatabank-master_2018-03-28/
↪baseballdatabank-master/core/')

extract.location # This is the location object associated to the extract.
```

Attributes of the location can be called by using the attribute's name::

```
extract.location.location_bucket_name # To retrieve the location's bucket (if s3␣
↪location)
```

ProcessTracker - Python

Below is a deeper dive of the capabilities of the Python implementation ProcessTracker submodule.

Initialize Process Run

As referenced in the ::doc::*Overview <overview.rst>*, ProcessTracker must first be imported and then the process can be registered by setting the variables of the ProcessTracker object:

```
from processtracker import ProcessTracker
from pyspark.sql import SparkSession

spark = SparkSession.builder.master("local")\
                            .appName("Lahman Teams Load")\
                            .config("spark.executor.memory", "6gb")\
                            .enableHiveSupport()\
                            .getOrCreate()

process_run = ProcessTracker(process_name='Lahman Teams Load'
                                       , process_type='Stage Load'
                                       , actor_name='New ProcessTracker User'
                                       , tool_name='Spark'
                                       , sources='Lahman Baseball Dataset')
```

Those variables will be used to populate the data store backend as explained in the following table:

Table 1: ProcessTracker object initialization variables

Variable Name	Variable Description	Reference Object	Object Created If Not Exist?
process_name	The name of the process being run. Must be unique.	*Process*	Yes
process_run_name	An optional unique name given to an individual process run	*Process Tracking*	Yes
process_type	The type of process being run. Optional if process already exists.	*Process Type*	Yes
actor_name	The person or thing kicking off the process run. Recommended to be variable driven.	:ref‘actor_lkup‘	Yes
tool_name	The tool being used to kick off the process run. Recommended to be variable driven.	*Tool*	Yes
sources	Single or list of source names where data is read from. Optional.	*Source*	Yes
source_objects	Dictionary of lists of source objects with source name as key where data is read from. Optional.	*Source Object*, *Process Source Object*	Yes
source_object_attributes	Dictionary of lists of source object attributes with source and source object names as keys where data is read from. Optional.	source_object_attribute	Yes
targets	Single or list of target names (alias of source) where data is written to. Optional.	*Source*	Yes
target_objects	Dictionary of lists of target source_objects with target name as key where data is read from. Optional.	*Source Object*, *Process Target Object*	Yes
target_object_attributes	Dictionary of lists of target object attributes with target and target object names as keys where data is read from. Optional.	source_object_attribute	Yes
config_location	Location where Process Tracker configuration file can be found. If not set will use the system home directory and check under .process_tracker for the process_tracker_config.ini file. Optional.	*Configuration*	N/A
dataset_types	Single or list of dataset category types. Will be associated with the process as well as any extracts, sources/targets, and source/target objects that are associated with the process.	*Dataset Type*, *Extract Dataset Type*, *Process Dataset Type*, *Source Dataset Type* , *Source Object Dataset Type*	No
schedule_frequency	The general frequency at which the process should run.	*ScheduleFrequency*	No
process_tracking_id	When recreating a Process Tracking instance, provide the process_tracking_id and it will re-instantiate as it was originally created, with the current status, objects, etc.	*Process Tracking*, plus all other objects created on instantiation of ProcessTracker	No

Once the instance has been instantiated, the rest of the options listed below become available.

Re-initializing An Instance

To set up a previous instance, pass the process tracking id to ProcessTracker. Instead of creating a new instance of the given process, it will retrieve the specific tracking record and all of it's ancillary data.:

```
restored_process_run = ProcessTracker(process_tracking_id=123)
```

This should ideally only be used when a process is still running - like when switching between services within your cloud provider.

Change Process Run Status

Throughout the process run the process run's status will need to be changed, usually to successful completion or to failure. ProcessTracker does allow for user defined process statuses, but the process run must finish with one of the system provided statuses if the process run is to work correctly with the rest of the system.

System provided statuses can be found at *Process Status*.:

```
process_run.change_run_status('completed')
```

Custom status types can be added either with the *CLI* tool or by entering the custom status in the change_run_status command. For instance:

```
process_run.change_run_status('my custom status')
```

On Hold Processes

Processes that are in the 'on hold' status are in that status because the max_concurrent_failures setting was reached. Currently, the only way to move a process out of 'on hold' is to manually change the last process run to 'completed' or some other status than 'running' or 'failed'.

Triggering Errors

Errors are custom failure messages that can be pretty much anything one would want to track during a process run. They do not necessarily trigger a process run to fail.:

```
process_run.raise_run_error(error_type_name='Data Error'
                           , error_description='Data item out of bounds.')
```

This raises an error stating an item was out of bounds for what we normally look for, but doesn't trigger the process run to fail because the hidden flag fail_run is defaulted to false. To fail a run set the flag to True.:

```
process_run.raise_run_error(error_type_name='Data Error'
                           , error_description='Data item out of bounds.'
                           , fail_run=True)
```

Another option for raising a run error is to set an end_date - this is if you want tighter control of the timestamps between ProcessTracker and any other logging you may have. This is not required because we are ideally talking about milliseconds between recording this error and writing to the log file.:

```
process_run.raise_run_error(error_type_name='Data Error'
                            , error_description='Data item out of bounds.'
                            , fail_run=True
                            , end_date=process_specific_datettime)
```

Auditing Processes

Auditing is a key feature of the ProcessTracker framework. Here are the available auditing options.

Setting Data Low/High Dates

It is important to know the data range of the data that is being processed by a run. This is where the low/high dates comes to play. The low date is the lowest date available from the data being processed. The high date is the highest date avilable from the data being processed. If audit dates are not provided with the data then the extract datetime can be utilized. If neither are available, then this audit option can't really be used.:

```
process_run.set_process_run_low_high_dates(low_date=extract_low_datetime
                                           , high_date=extract_high_datetime)
```

If a lower or higher datetime is registered, the previous datetimes will be compared and whichever is lower of the two low dates and higher of the two high dates will be kept. While this can be set via loop, it is recommended to find the low and high dates in the set before calling set_process_run_low_high_dates() as it does make a insert/update per call.

Setting Record Count

It is important to know how many records the process and process run have processed. This can aid capacity and resource planning, especially if the information is tracked over time.:

```
process_run.set_process_run_record_count(num_records=10000)
```

set_process_run_record_count does two things:

- sets the process run record's total record count (wiping out the previous

value)

- sets the process' total record count (cumulative)

It is recommended that the number of records be determined on a per extract file or a cumulative total before setting the record count.

Dataset Types

Dataset types are high level categories for the type(s) of data the process and ancilliary objects like extracts, sources/targets, source objects/target objects, etc. can be associated to. Once the variable dataset_types is set there is nothing else required to associate the process and other entities associated to the process to the dataset type(s).

Tracking Process Sources

Processes can have sources associated for auditing purposes. There are two methods for tracking sources - source level and source object level.

Source Level

Source level tracking can be done by including a single source name or list of source names on process initialization. For example::

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , sources='Lahman Baseball Dataset')
```

For multiple sources::

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , sources={'Lahman Baseball Dataset', 'Another Baseball␣
↪Dataset'})
```

Source Object Level

Source Object level tracking is done in a similar way as above. Regardless of being a single source object, multiple source objects, or multiple sources with single or multiple objects, source object level tracking is done via a dictionary of lists.:

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , source_objects={"Lahman Baseball Dataset": ["Team.csv",
↪ "Player.csv"]}
```

For multiple sources::

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , source_objects={"Lahman Baseball Dataset": ["Team.csv",
↪ "Player.csv"]
                                                 , "Another Baseball Dataset": ["Team",
↪"Player"]}
```

Note that sources is not set. The sources variable will be ignored if source_objects is set.

Tracking Process Targets

Processes can have targets associated for auditing purposes. There are two methods for tracking targets - target level and target object level. Remember target is just an alias of source. All targets and sources are stored in the *Source* table.

Target Level

Target level tracking can be done by including a single target name or list of target names on process initialization. For example::

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , targets='My Baseball Datastore')
```

For multiple targets::

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , targets={'My Baseball Datastore', 'A Different␣
↪Baseball Datastore'})
```

Target Object Level

Target Object level tracking is done in a similar way as above. Regardless of being a single target object, multiple target objects, or multiple targets with single or multiple targets, target object level tracking is done via a dictionary of lists.:

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , target_objects={"My Baseball Datastore": ["team",
↪"player"]}
```

For multiple targets::

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , target_objects={"My Baseball Datastore": ["team",
↪"player"]
                                               , "A Different Baseball Datastore": [
↪"Team", "Player"]}
```

Note that targets is not set. The targets variable will be ignored if target_objects is set.

Process Extracts

The other element to processing data is the extract files that may be used in the process or between processes. Note that using this is not required if extract files are not used. Extracts are always associated with a process run, which is why the extract functionality is primarily tied to the ProcessTracker submodule.

Finding Extracts

Extract files can be found in a few different ways. Finders will return extracts in 'ready' state by default. Other statuses can be searched for if required by adding the *status* variable. The finders also will only return extract files that have been registered in ProcessTracker.

By Filename

Full Filename

So let's say that you know that there is a specific file that needs processing. You can search for a specific file by:

```
process_run.find_extracts_by_filename(filename='my_file.csv')
```

This will return the ExtractTracking object, which includes the location of the file.

Partial Filename

Let's say that you know that the files you are looking for match a specific pattern, for example::

```
my_file_2019_01_01.csv
my_file_2019_02_01.csv
...
```

Instead of looking for each file one at a time, you can use the partial filename::

```
process_run.find_extracts_by_filename(filename='my_file_')
```

This will return the ExtractTracking object, which included the location of the file. This function is greedy meaning it will return ANY files with 'my_file' in the filename. For instance::

```
my_file_2019_01_01.csv
this_is_my_file.xls
2019-01-01-my_file.csv
```

By Location

Locations are the filepaths where extract files are stored. These can be local, a network drive, or a cloud directory.:

```
process_run.find_extracts_by_location(location='My Location')
```

The location name is used and the ExtractTracking object(s) are returned.

By Process

If the process has a parent process that creates files for it, or there is a process that produces files that will be used then the parent process' name can be used to find any ready extracts::

```
process_run.find_extracts_by_process(extract_process_name='My Super Cool Process')
```

This will find all extract files associated to that process that are in 'ready' state and return their ExtractTracking objects.

Finding Extracts By Other Statuses

All finder methods have a status variable with a default of 'ready'. To search by another status type, just modify the variable::

```
process_run.find_extracts_by_location(location='My Location', status='completed')
```

The status type must exist in *Extract Status*.

Registering Extracts

If your process is creating extract files, they will need to be registered. They can either be registered one at a time as noted in *ExtractTracker* or one of the below helper methods.

By Location

This will attempt to access the given location and find all files stored there. If the files are not already registered they will be processed, otherwise they will be ignored.:

```
process_run.register_extracts_by_location(location_path='/path/to/files')
```

Currently, this only supports local filepaths.

By Process

This method is explained over in *ExtractTracker*.

Bulk Extract Update

Extracts can also be processed in bulk. If you use one of the lookup functions, it returns a list of extract file objects. Passing that list to the bulk_change_extract_status method will associate those extracts with the process and bulk update their status.:

```
process_run.bulk_change_extract_status(extracts=extract_list, extract_status="loading
↪")
```

Please note, that while going through the list if any of the extracts are interdependent of each other and the parent dependency has not been loaded, the process will currently fail to protect data continuity.

Process Helpers

There are several helpers for ProcessTracker objects to assist in working with other components associated with the ProcessTracker instance. Those objects are:

- actor
- process_type
- process
- sources

- targets
- tool

To use attributes of the objects, call the attribute like so::

```
process_run = ProcessTracker(process_name='Lahman Teams Load'
                             , process_type='Stage Load'
                             , actor_name='New ProcessTracker User'
                             , tool_name='Spark'
                             , sources='Lahman Baseball Dataset')

process_run.actor.actor_name # To get the actor_name attribute of actor object␣
↪associated with process_run.
```

Finding Process Contacts

To find a process' contacts as well as that process' source contacts, there is a lookup function that will return the name and email of the contact as well as if the contact is for the process itself or that process' source.:

```
process_run.find_process_contacts(process_id)
```

Finding Process Source Attributes

Processes usually have sources that they work with to pull data out into either extracts or for further processing. To find the source attributes that a process will use there is a helper function.:

```
process_run.find_process_source_attributes(process_id)
```

This will return a list with the following:

Table 2: process_source_attributes returned

Attribute Name	Attribute Description
source_name	The name of the source system
source_type	The type of the source system
source_object_name	The object (i.e. table or dataset) name from the source system
source_object_attribute_name	The attribute (i.e. column) name from the source object
is_key	Is the attribute part of the source object's key?
is_filter	Is the attribute used for record version comparison?
is_partition	Is the attribute used for the partitioning of the data set (i.e. if the filetype is Parquet is the attribute used for the file partition)?

Please note, if the attributes are not registered either during the initialization of the process or thru direct interaction of the data store, no data will be returned.

Finding Process Target Attributes

Processes usually have targets that they work with to push data into or for further processing. To find the target attributes that a process will use there is a helper function.:

```
process_run.find_process_target_attributes(process_id)
```

This will return a list with the following:

Table 3: process_target_attributes returned

Attribute Name	Attribute Description
target_name	The name of the target source system
target_type	The type of the target source system
target_object_name	The object (i.e. table or dataset) name from the target source system
target_object_attribute_name	The attribute (i.e. column) name from the target source object
is_key	Is the attribute part of the source object's key?
is_filter	Is the attribute used for record version comparison?
is_partition	Is the attribute used for the partitioning of the data set (i.e. if the filetype is Parquet is the attribute used for the file partition)?

Please note, if the attributes are not registered either during hte initialization of the process or thru direct interaction of the data store, no data will be returned.

Finding Processes By Schedule Frequency

Processes can have an optional schedule frequency. To find all processes of a given frequency there is a helper function.:

```
process_run.find_process_by_schedule_frequency(frequency="hourly")
```

This will return a list of process objects with the given frequency.

Finding Process Filters

Processes will be querying data from given sources. The source data will likely be filtered on specific criteria. To retrieve the process' filters, use the find_process_filters method.:

```
process_run.find_process_filters(process=process_id)
```

This will return a list of attributes and their requisite filters.

Table 4: filter attributes provided

Attribute name	Attribute description
source_name	The name of the source system
source_object_name	The name of the object (i.e. table or data set) within/from the source system
source_object_attribute_name	The attribute (i.e. column) name from the object in within/from the source system
filter_type_code	Based on the defaults in *Filter Type*, the type of filter being applied to the source object attribute
filter_value_numeric	The value being filtered by, provided the attribute is numeric.
filter_value_string	The value being filtered by, provided the attribute is string.

CHAPTER 2

Setting Up ProcessTracker

Before utilizing one of the implementations of ProcessTracker, there is a little bit of setup that must be completed.

2.1 Data Store Setup

ProcessTracker utilizes a relational database to store and track all of it's Process and Extract information. Eventually, setting up the data store will be enabled in the code implementations of ProcessTracker, but in the meantime, there are prepared scripts for four of the more popular relational databases (Postgres, MySQL, MS-SQL, Oracle) that can be found here <add doc link>.

2.2 Configuration

As far as configuration is concerned, for code package implementations a simple config file (process_tracker.ini) can be found in the user's home directory under .process_tracker once the module has been invoked the first time. For tool implementations, configuration will be handled in the best practice of the tool (i.e. the tool's configuration files, shared modules, etc.)

2.2.1 process_tracker_config.ini

The process_tracker_config.ini file contains the following settings:

Table 1: process_tracker_config.ini

Variable Name	Variable Description
log_level	Logging level of ProcessTracker
max_concurrent_failures	Maximum number of concurrent failures allowed before process is put in status 'on hold'
data_store_type	The database type. Valid options: postgresql, mysql, mssql, oracle, snowflake
data_store_username	The username used to connect to the data store
data_store_password	The password used by the username to connect to the data store
data_store_host	The host server of the data store
data_store_port	The port of the host used to connect to the data store
data_store_name	The name of the database/schema storing ProcessTracking data in the data store

The data_store_* variables are used to build a connection string to the data store.

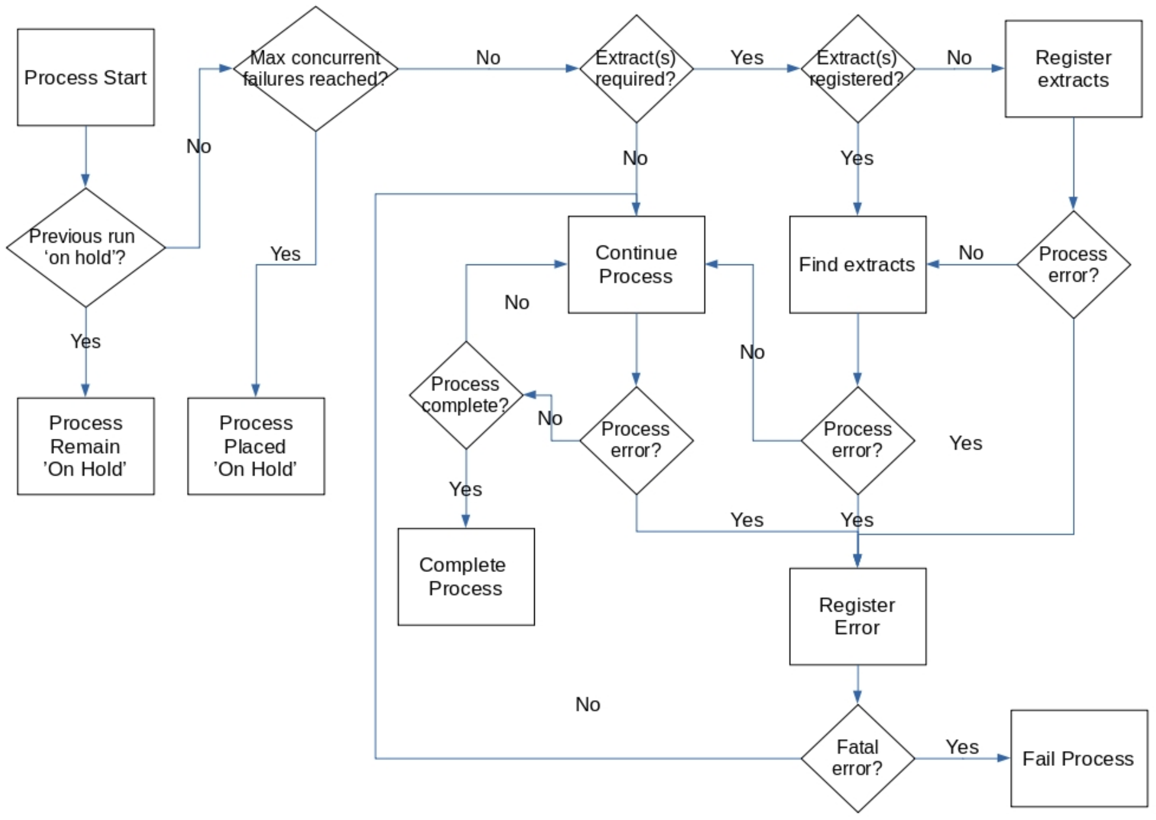
2.2.2 Encrypting Data Store Passwords

Data store passwords can be encrypted so that they are not stored in plain text. The method used is NOT cryptographically secure. This just makes it a bit harder for someone to access your data store password. Please refer to the *Password Encryption* section.

CHAPTER 3

ProcessTracker Workflow

ProcessTracker is designed to have a standardized (yet simple) workflow that all implementations must follow, regardless if the implementation is language or tool specific.



This page will now describe the steps in the workflow above. Examples of the steps are provided in the individual implementation sections, so that users can view examples with the implementation they are using.

3.1 Process Start

When a process run (the instance of a process processing) is first kicking off, it must be registered in the ProcessTracker data store. This includes information such as:

Table 1: process registration entities

Entity Name	Entity Description
process_name	The unique name of the process
process_type	The overall type of process the process is. For instance, if the process is only extracting data it could be classified as of type 'Extract'.
actor_name	The person or thing that initiated the process
tool_name	The type of tool used in this process. For instance, Spark, Talend, Pentaho, DataStage, etc.
sources	A single name or list of names of source systems being used in this process.
targets	A single name or list of names of target systems being used in this process. These are stored as sources since a source can be a target, and a target can be a source.

These entities are described in more detail in the *Model* section.

Once the process run is registered, it must be determined with extract file(s) are required for further processing.

3.2 Previous run 'on hold'?

If the process has failed enough times, then the process should no longer try to process regardless of how many times it has triggered. This status is termed 'on hold'. If the previous run of a process has hit the max_concurrent_failures setting value, then the run will be placed on hold. The process will remain in that status until manually switched over to 'completed' or another status that is not 'running' or 'failed'.

3.3 Max concurrent failures reached?

If the previous run is not in the 'on hold' status, then a number of previous process runs matching the max_concurrent_failures will be checked. If those runs are all failures (matching the max_concurrent_failures value) the current run will be placed 'on hold' until the failure is resolved. Otherwise the process run will proceed as normal.

3.4 Extract(s) required?

Sometimes processes are not straight through from source to target. This is where extracts come into play. An extract is basically a data dump that acts as a intermediary between processes. Extracts can be one time use, reusable, or just available within a single process run - it's really up to how you want to leverage the framework. If extracts are required then they must be registered into the ProcessTracker data store. If they are not required, then the process can continue running.

3.5 Extract(s) registered?

A process run does not have to necessarily register files independently of other processes. The files may have been created beforehand by another process. If the process is not creating files, then it can find the files by location and filename. Otherwise, if files are being created by the process run, the extracts must be registered.

3.6 Register extracts

Similar to how processes/process runs must be registered in the ProcessTracker data store, extract files must as well. Registration includes data such as:

Table 2: extract registration entities

Entity Name	Entity Description
process_run	The process run that's working with the extract file.
filename	The name of the file. Should not include the file path.
location	The file path where the file is located.
status	By default the status is set to 'initialized'.

These entities are described in more detail in the *Model* section.

3.7 Find extracts

If the extract files have already been registered, then they can be located by filename and location. Once found they can be used by the process run to continue processing, provided the location is still accessible to the process.

3.8 Continue process

If all goes well, then the process can continue to run. Otherwise, the error tracking system can be utilized. As extracts are finished being used, their status should be changed to reflect where they are in the process. This can either be done all at once or individually, depending on how fault tolerant the process being run is.

3.9 Register error

Any errors triggered during the process run can be registered in ProcessTracker's data store. These can range from system errors, process errors, data errors, etc. There is no proscription to what can be tracked error-wise. That said, there are some errors that just can't be tracked. For instance, if your process run runs out of memory then things may go seriously bork, to the point that the system can't actually record the error or continue running.

3.10 Fatal error?

Many error classes are bad enough to prompt the process run to come to an end. If that is the case, the process run must also be put in a failed status along with any errors that need to be recorded.

3.11 Fail Process

If the process needs to fail, then the status of the process run should be changed to 'failed'.

3.12 Process complete?

If all processing for the process run successfully completed, then the status of the process run can be changed to ‘complete’.

CHAPTER 4

Model

This section provides details on the data store model for ProcessTracker.

4.1 Overview

ProcessTracker officially supports Postgresql and MySQL as fully tested and verified to work.

Support is also enabled for Oracle, MS-SQL, and Snowflake, but has not been fully tested thru the build stack. While it should Just Work (tm) until they can be tested thru the build stack and not just locally their status must remain ‘available but untested’.

Each subsection of the Model section covers a specific topic of the data store.

New with version 0.9.0, all tables are now equipped with stock audit fields:

Table 1: stock audit fields

Column Name	Column Type	Column Description
created_date_time	timestamp (with timezone, if available)	The date/time that the record was created.
created_by	integer	The id of the actor who created the record. Implied foreign key to the *Actor* table.
update_date_time	timestamp (with timezone, if available)	The date/time that the record was last updated.
updated_by	integer	The id of the actor who updated the record. Implied foreign key to the *Actor* table.

4.1.1 Actor

Actors are people or things that trigger processes off.

Actor

Table 2: actor_lkup

Column Name	Column Type	Column Description
actor_id	Auto incrementing integer sequence	System key for the actor
actor_name	String(250)	Unique name of a person or thing

4.1.2 Capacity Management

Capacity Management involves organization of environment resources and assigning processes to specific groups (‘clusters’).

Cluster

Table 3: cluster_tracking_lkup

Column Name	Column Type	Column Description
cluster_id	Auto incrementing integer sequence	System key for the cluster
cluster_name	String(250)	Unique name of the given cluster
cluster_max_memory	Integer	The maximum amount of memory allocated to the cluster. Optional.
cluster_max_memory_unit	String(2)	The unit of measure for the max memory field. Examples include: MB, GB, TB. Optional.
cluster_max_processing	Integer	The maximum amount of processing allocated to the cluster. Optional.
cluster_max_processing_unit	Integer	The unit of measure for the max processing field. Examples include: Ghz, CPU, DPU. Optional.
cluster_current_memory_usage	Integer	Based on the processes running within the cluster, how much memory are they using. Optional field. Not currently in use.
cluster_current_process_usage	Integer	Based on the processes running within the cluster, how much processing power are they using. Optional. Not currently in use.

ClusterProcess

This table tracks the relationships between clusters and processes. Used for performance balancing.

Table 4: cluster_process

Column Name	Column Type	Column Description
cluster_id	Integer	The cluster of the cluster-process relationship. Foreign key to cluster_tracking_lkup.
process_id	Integer	The process of the cluster-process relationship. Foreign key to *Process*.

4.1.3 Contact

These are the tables associated with contacts. Links to other subjects will be provided as noted.

Contact

This table tracks contact information for notifications/communication.

Table 5: contact_lkup

Column Name	Column Type	Column Description
contact_id	Integer	Auto incrementing integer sequence
contact_name	String(250)	Unique name of the given contact
contact_email	String(750)	Unique email of the given contact

4.1.4 Error Tracking

These are the tables associated with error tracking/handling. Links to other subjects will be provided as noted.

Error Tracking

This is the core error tracking table. This associates errors to process runs.

Table 6: error_tracking

Column Name	Column Type	Column Description
error_tracking_id	Auto incrementing integer sequence	System key for the error tracking record
error_type_id	Integer	Foreign key to *Error Type* lookup table.
error_description	String(750)	Provided description that further defines the error type. Not required.
error_occurrence_date_time	Datetime/timestamp	The date/time that the error occurred
process_tracking_id	Integer	Foreign key to *Process Tracking* table.

Error Type

Error types can be any user defined label of errors.

Table 7: error_type_lkup

Column Name	Column Type	Column Description
error_type_id	Auto incrementing integer sequence	System key for the error type
error_type_name	String(250)	Unique label for the class of error type

Some default error types are provided on initialization.

Table 8: Default Error Types

Error Type	Description
File Error	Errors associated with file problems - opening, writing, reading, missing, etc.
Data Error	Errors associated with data problems - format, quality, out of range, etc.
Process Error	Errors associated with process problems.

Custom error types can be added as needed to provide finer-grained recording and monitoring.

4.1.5 Extract Tracker

These are the tables associated with extract tracking/handling. Links to other subjects will be provided as noted.

Extract Compression Type

This table tracks the valid compression types available.

Table 9: extract_compression_type_lkup

Column Name	Column Type	Column Description
extract_compression_type_id	Integer	Auto incrementing integer sequence
extract_compression_type	Integer	Unique compression type name

Some default extract compression types are provided on initialization.

Table 10: Default Extract Compression Types

Extract Compression Type	Description
zip	Zip Compression

Extract Dataset Type

This table tracks the relationship between extract files and dataset types.

Table 11: extract_dataset_type

Column Name	Column Type	Column Description
extract_id	Integer	Foreign key to the *Extract Tracking* table.
dataset_type_id	Integer	Foreign key to the *Dataset Type* table.

Extract Dependency

This table tracks the interdependencies between extract files.

Table 12: extract_dependency

Column Name	Column Type	Column Description
parent_extract_id	Integer	Foreign key to the *Extract Tracking* table. The parent extract of the relationship.
child_extract_id	Integer	Foreign key to the *Extract Tracking* table. The child extract of the relationship.

Extract File Type

This table tracks the valid types of files and their formats that available.

Table 13: extract_filetype_lkup

Column Name	Column Type	Column Description
extract_filetype_id	Integer	Auto incrementing unique sequence
extract_filetype_code	String(25)	The file extension used by the filetype (i.e. csv)
extract_filetype	String(75)	The unique name of the filetype.
delimiter_char	String(1)	For filetypes like csv, the character used to delimit fields.
quote_char	String(1)	For filetypes like csv, the character used to quote fields.
escape_char	String(1)	For filetypes like csv, the character used to escape fields.

Some default extract file types are provided on initialization.

Table 14: Default Extract File Types

Extract File Type	Description
csv	Comma Separated Values

Extract Process Tracking

This table tracks the association between extract files and process runs.

Table 15: extract_process_tracking

Column Name	Column Type	Column Description
extract_tracking_id	Integer	Foreign key to the *Extract Tracking* table.
process_tracking_id	Integer	Foreign key to the *Process Tracking* table.
extract_process_status_id	Integer	Status of the extract from the process run. Foreign key to the *Extract Status* table.
extract_process_event_date_time	Datetime/timestamp	The date/time of the status change for the extract.

Extract Source

This table tracks the relationship between extracts and their sources.

Table 16: extract_source

Column Name	Column Type	Column Description
extract_id	Integer	Foreign key to *Extract Tracking*
source_id	Integer	Foreign key to *Source*

Extract Source Object

This table tracks the relationship between extracts and their source objects.

Table 17: extract_source_object

Column Name	Column Type	Column Description
extract_id	Integer	Foreign key to *Extract Tracking*
source_id	Integer	Foreign key to *Source Object*

Extract Status

This table is a lookup of system and user provided extract statuses.

Table 18: extract_status_lkup

Column Name	Column Type	Column Description
extract_status_id	Auto incrementing integer sequence	System key for the extract status
extract_status_name	String(75)	Unique name of the extract status type

Some default extract status types are provided on initialization.

Table 19: Default Extract Status Types

Extract Status Type	Description
initializing	The extract file is being written to and/or is not ready for use.
ready	The extract file is ready to be used.
loading	The extract file is being used/loaded by a process run.
loaded	The extract file has successfully been loaded by a process run.
archived	The extract file has successfully been archived and can only be reprocessed if moved back out of archive location.
deleted	The extract file has successfully been removed from the archive and can no longer be retrieved.
error	Something went wrong in the writing/processing of the extract file. Until resolved, file is unusable.

Custom extract status types can be added, but can not currently be utilized by the ProcessTracker framework.

Extract Tracking

This table is the core of the extract tracking subsystem.

Table 20: extract_tracking

Column Name	Column Type	Column Description
extract_id	Auto incrementing integer sequence	System key for the extract file
extract_filename	String(750)	The unique filename of the extract file
extract_location_id	Integer	Where the extract file can be located. Foreign key to *Location*
extract_status_id	Integer	The current status of the extract file. Foreign key to *Extract Status*
extract_registration_date_time	Datetime/timestamp	The date/time that the extract was initially registered into the system.
extract_write_low_date_time	Datetime/timestamp	The earliest derived datetime for data processed in this extract at write. Optional audit field.
extract_write_high_date_time	Datetime/timestamp	The latest derived datetime for data processed in this extract at write. Optional audit field.
extract_write_record_count	Integer	For the given extract file at write, the total number of records processed. Optional audit field.
extract_read_low_date_time	Datetime/timestamp	The earliest derived datetime for data processed in this extract at read. Optional audit field.
extract_read_high_date_time	Datetime/timestamp	The latest derived datetime for data processed in this extract at read. Optional audit field.
extract_read_record_count	Integer	For the given extract file at read, the total number of records processed. Optional audit field.
extract_compression_type_id	Integer	Optional compression type used on the extract. Foreign key to *Extract Compression Type*
extract_filetype_id	Integer	File type/format used by the extract. Foreign key to *Extract File Type*
extract_filesize	Numeric	The size of the extract
extract_filesize_type_id	Integer	The measure of the extract filesize. Foreign key to *File size Type*

File size Type

This table provides file sizes for extracts.

Table 21: filesize_type_lkup

Column Name	Column Type	Column Description
filesize_type_id	Integer	Primary key for the file size type
filesize_type_name	String(75)	Full name of the file size type
filesize_type_code	String(2)	Code used by the file size type

There are defaults provided for file sizes on initialization:

Table 22: filesize_type_lkup defaults

System Key	System Value	System Code
1	kilobytes	KB
2	megabytes	MB
3	gigabytes	GB
4	bytes	B

Location

This table tracks extract file locations.

Table 23: location_lkup

Column Name	Column Type	Column Description
location_id	Auto incrementing integer sequence	System key for the file location
location_name	String(750)	Unique optional name of the location. Will be derived from the filepath if not provided.
location_path	String(750)	Unique filepath.
location_type_id	Integer	The type of location for given filepath. Foreign key to *Location Type*.
location_file_count	The number of files currently in the given location.	Integer

Location Type

This table tracks extract file location types.

Table 24: location_type_lkup

Column Name	Column Type	Column Description
location_type_id	Auto incrementing integer sequence	System key for the location type
location_type_name	String(25)	The unique name of the type of location.

Some default location types are provided on initialization.

Table 25: Default Location Types

Location Type	Description
S3	S3 bucket location
Local Filesystem	Local filesystem location

4.1.6 Process Tracking

These are the tables associated with process tracking/handling. Links to other subjects will be provided as noted.

Process

This table tracks the unique processes being tracked by ProcessTracker.

Table 26: process

Column Name	Column Type	Column Description
process_id	Auto incrementing integer sequence	System key for the process
process_name	String(250)	Unique name of the process
total_record_count	Integer	Audit field that tracks the total number of records processed throughout the lifetime of the process.
process_type_id	Integer	The type of process. Foreign key to *Process Type*.
process_tool_id	Integer	The type of tool used to run the process. Foreign key to *Tool*.
last_failed_run_date_time	Datetime/timestamp	The date/time of the last failed run of this process.
schedule_frequency_id	Integer	The schedule frequency of the process. Foreign key to *ScheduleFrequency*
last_completed_run_date_time	Datetime/timestamp	The date/time of the last successful completion of this process
last_errored_run_date_time	Datetime/timestamp	The date/time of the last errored run of this process

Process Contact

This table tracks the relationship between processes and their contacts.

Table 27: process_contact

Column Name	Column Type	Column Description
process_id	Integer	The contact's process. Foreign key to *Process*.
contact_id	Integer	The process' contact. Foreign key to contact.

Process Dataset Type

This table tracks the relationship between process and dataset types.

Table 28: process_dataset_type

Column Name	Column Type	Column Description
process_id	Integer	Foreign key to the *Process* table.
dataset_type_id	Integer	Foreign key to the *Dataset Type* table.

Process Dependency

This table tracks the interdependencies between processes, regardless of tool/method to execute said processes.

Table 29: process_dependency

Column Name	Column Type	Column Description
parent_process_id	Integer	The parent process of the parent-child relationship. Foreign key to *Process*.
child_process_id	Integer	The child process of the parent-child relationship. Foreign key to *Process*.

Please note - the dependency hierarchy can theoretically go on infinitely. In reality only a few levels either way would realistically be used, but this type of relationship can cause performance issues.

Process Filter

This table tracks query filters for a given process.

Table 30: process_filter

Column Name	Column Type	Column Description
process_filter_id	Integer	Auto incrementing unique sequence
process_id	Integer	The filter's process. Foreign key to *Process*.
source_object_attribute_id	Integer	The process filter's source_object attribute. Foreign key to source_object_attribute.
filter_type_id	Integer	The filter's type. Foreign key to *Filter Type*
filter_value_string	String(250)	For character based attributes, the string comparator.
filter_value_numeric	Numeric	For numeric based attributes, the numeric comparator.

Process Source

This table tracks what sources are used by a given process.

Table 31: process_source

Column Name	Column Type	Column Description
source_id	Integer	The source system utilized by the process. Foreign key to *Source*.
process_id	Integer	The process utilizing the source. Foreign key to *Process*.

Process Source Object

This table tracks the finer grained relationship between process and source object.

Table 32: process_source_object

Column Name	Column Type	Column Description
process_id	Integer	The process utilizing the source object. Foreign key to *Process*.
source_object_id	Integer	The source object being utilized by the process. Foreign key to *Source Object*.

Process Source Object Attribute

This table tracks even finer grained relationships between process and source object attributes.

Table 33: process_source_object_attribute

Column Name	Column Type	Column Description
process_id	Integer	The Process associated to the source object attribute. Foreign key to *Process*.
source_object_attribute_id	Integer	The Source Object Attribute associated to the process. Foreign key to source_object_attribute.
source_object_attribute_alias	String(250)	The optional alias used by the process on the attribute.
source_object_attribute_expression	String(250)	The optional expression (calculation) used on the attribute.

Process Status

This table is a lookup table for the types of process statuses available in the system.

Table 34: process_status_lkup

Column Name	Column Type	Column Description
process_status_id	Auto incrementing integer sequence	System key for the process
process_status_name	String(75)	Unique name of the process status

Some default process status types are provided on initialization.

Table 35: Default Process Status Types

Process Status Type	Description
running	The process is running. No other instances or child dependencies can be run.
completed	The process completed successfully. Other instances and child dependencies can be run.
failed	The process did not complete successfully. Other instances may be run, but child dependencies will be blocked.

Other custom process status types can be added, but the system can not currently take advantage of them.

Process Target

This table tracks the targets that processes write to. Target is an alias of source since sources can be targets and vice-versa.

Table 36: process_target

Column Name	Column Type	Column Description
target_source_id	Integer	The source system the process is writing to. Foreign key to *Source*.
process_id	Integer	the process utilizing the source. Foreign key to *Process*.

Process Target Object

This table tracks the finer grained relationship between process and source target object.

Table 37: process_target_object

Column Name	Column Type	Column Description
process_id	Integer	The process utilizing the source object. Foreign key to *Process*.
target_object_id	Integer	The target object being utilized by the process. Foreign key to *Source Object*.

Process Target Object Attribute

This table tracks even finer grained relationships between process and target source object attributes.

Table 38: process_target_object_attributes

Column Name	Column Type	Column Description
process_id	Integer	The Process associated to the target source object attribute. Foreign key to *Process*.
target_object_attribute_id	Integer	The Target Source Object Attribute associated to the process. Foreign key to source_object_attribute.
target_object_attribute_alias	String(250)	The optional alias used by the process on the attribute.
target_object_attribute_expression	String(250)	The optional expression (calculation) used on the attribute.

Process Tracking

This table is the core of the process tracking subsystem.

Table 39: process_tracking

Column Name	Column Type	Column Description
process_tracking_id	Auto incrementing integer sequence	System key for the process run
process_id	Integer	The process being run. Foreign key to *Process*.
process_status_id	Integer	The current status of the process run. Foreign key to *Process Status*.
process_run_id	Integer	Unique sequence of the given process' runs.
process_run_low_date_time	Datetime	The earliest derived datetime for data processed in this process run. Optional audit field.
process_run_high_date_time	Datetime	The latest derived datetime for data processed in this process run. Optional audit field.
process_run_start_date_time	Datetime/timestamp	The date/time that the process run was registered.
process_run_end_date_time	Datetime/timestamp	The date/time that the process finished running, regardless of success or failure.
process_run_record_count	Integer	For the given process run, the total number of records processed. Optional audit field.
process_run_actor_id	Integer	The person or thing that kicked off the process run. Foreign key to *Actor*.
is_latest_run	Boolean	Bit to determine if for the given process if the record is the latest run or not.
process_run_name	String(250)	Unique process instance name, optional.

Process Type

This table is a lookup of the various process types available.

Table 40: process_type_lkup

Column Name	Column Type	Column Description
process_type_id	Auto incrementing integer sequence	System key for the process type
process_type_name	String(250)	Unique name of the process type

Some default process types are provided on initialization.

Table 41: Default Process Types

Process Type	Description
Extract	Process that is focused on extracting data.
Load	Process that is focused on loading data.

Custom process types can be added.

4.1.7 Schedule

These are the tables associated with process schedules. Links to other subjects will be provided as noted.

ScheduleFrequency

This table tracks general schedule frequencies.

Table 42: schedule_frequency_lkup

Column Name	Column Type	Column Description
schedule_frequency_id	Integer	Auto incrementing interger sequence.
schedule_frequency_name	String(25)	Unique name of the schedule frequency.

Some default schedule frequencies are provided on initialization.

Table 43: Default Schedule Frequencies

Schedule Frequency	Description
unscheduled	Processes are not run on any schedule.
hourly	Processes should run on an hourly basis.
daily	Processes should run on a daily basis.
weekly	Processes should run on a weekly basis.
monthly	Processes should run on a monthly basis.
quarterly	Processes should run on a quarterly basis.
annually	Processes should run on an annual basis.

4.1.8 Source

Sources are data stores that can either be the source of data to be processed or targets for processes to write data to.

Character Set

This table tracks the character set used by data

Table 44: character_set_lkup

Column Name	Column Type	Column Description
character_set_id	Integer	Primary key
character_set_name	String(75)	~Unique name of the given character set

Data Type

This table tracks the unique data types of source object attributes.

Table 45: data_type_lkup

Column Name	Column Type	Column Description
data_type_id	Integer	Auto incrementing unique sequence.
data_type	String(75)	Unique data type name.

Dataset Type

This table tracks dataset types (i.e. Customer, Sales, Employee, etc.).

Table 46: dataset_type_lkup

Column Name	Column Type	Column Description
dataset_type_id	Auto incrementing integer sequence	System key for the dataset type
dataset_type	String(250)	Unique name of the given dataset type

Filter Type

This table tracks the unique filter types used by processes.

Table 47: filter_type_lkup

Column Name	Column Type	Column Description
filter_type_id	Integer	Auto incrementing unique sequence
filter_type_code	String(3)	The unique code for a given filter type.
filter_type_name	String(75)	The unique filter type name.

Some default filter types are provided on initialization.

Table 48: Default Filter Types

Filter Type Code	Filter Type Name	Description
eq	equal to	Does the attribute equal to a given value?
lt	less than	Is the attribute less than a given value?
gt	greater than	Is the attribute greater than a given value?
lte	less than or equal to	Is the attribute less than or equal to a given value?
gte	greater than or equal to	Is the attribute greater than or equal to a given value?
not	not equal	Does the attribute not equal to a given value?
lke	like	Is the attribute like a given value?
in	in set	Is the attribute in a given value set?

Source

This is the core table tracking sources/targets. Note that one data flow's source is likely another data flow's target. They are all stored here.

Table 49: source_lkup

Column Name	Column Type	Column Description
source_id	Auto incrementing integer sequence	System key for the source
source_name	String(250)	Unique name of the given source.
character_set_id	Integer	The source's character set. Foreign key to *Character Set*
source_type_id	Integer	The source's type. Foreign key to *Source Type*.

Source Contact

This table tracks the relationship between sources and contacts.

Table 50: source_contact

Column Name	Column Type	Column Description
source_id	Integer	The Contact's source system. Foreign key to *Source*.
contact_id	Integer	The Source system's contact(s). Foreign key to *Contact*.

Source Dataset Type

This table tracks the relationship between sources and dataset types.

Table 51: source_dataset_type

Column Name	Column Type	Column Description
source_id	Integer	Foreign key to the *Source* table
dataset_type_id	Integer	Foreign key to the *Dataset Type* table

Source Location

This table tracks relationships between sources and the location(s) where their data is stored.

Table 52: source_location

Column Name	Column Type	Column Description
source_id	Integer	Foreign key to the *Source* table
location_id	Integer	Foreign key to the *Location* table

Source Object

This is the core table tracking source/target objects.

Table 53: source_object_lkup

Column Name	Column Type	Column Description
source_object_id	Auto incrementing integer sequence	System key for the source object
source_id	Integer	Foreign key to *Source*.
source_object_name	String(250)	Unique object name from given source.

Source Object Attribute

This is the core table tracking source/target object attributes.

Table 54: source_object_attribute_lkup

Column Name	Column Type	Column Description
source_object_attribute_id	Integer	Auto incrementing integer sequence.
source_object_attribute_name	String(250)	Name of the source object attribute. Must be unique to the source_object.
source_object_id	Integer	The attribute's source object. Foreign key to *Source Object*
attribute_path	String(750)	For attributes from sources like json, the path to get to the attribute.
data_type_id	Integer	The data type of the attribute. Foreign key to *Data Type*
data_length	Integer	The length of the attribute.
data_decimal	Integer	How many decimal places of the attribute.
is_pii	Boolean	Is the attribute Personally Identifiable Information (PII)?
default_value_string	String(250)	For string based attributes, the default value.
default_value_number	Numeric	For numeric based attributes, the default value.
is_key	Boolean	Is this attribute part of the key for the object?
is_filter	Boolean	Is this attribute part of the set used to determine if changes have occurred?
is_partition	Boolean	Is this attribute used to partition the data set?

Source Object Dataset Type

This table tracks the relationship between source/target objects and dataset types.

Table 55: source_object_dataset_type

Column Name	Column Type	Column Description
source_object_id	Integer	Foreign key to the *Source Object* table
dataset_type_id	Integer	Foreign key to the *Dataset Type* table

Source Object Location

This table tracks relationships between source objects and the location(s) where their data is stored.

Table 56: source_object_location

Column Name	Column Type	Column Description
source_object_id	Integer	Foreign key to the *Source Object* table
location_id	Integer	Foreign key to the *Location* table

Source Type

This table provides unique source types to classify sources.

Table 57: source_type_lkup

Column Name	Column Type	Column Description
source_type_id	Integer	Primary key
source_type_name	String(75)	Unique source type for classification of sources

The following defaults are provided on initialization.

Table 58: system_lkup defaults

System Key	System Value	Description
1	Undefined	The source does not have a source type defined
2	Database	The source is a relational database
3	Internal	The source is internal to your company
4	External	The source is external to your company

4.1.9 System Settings

For future updates, system tables are provided. This is only used by the system. Anything added to these tables will be ignored.

System

This is the core system table, to be used in future updates.

Table 59: system_lkup

Column Name	Column Type	Column Description
system_id	Auto incrementing integer sequence	System key for the system key record
system_key	String(250)	Unique key record for system value
system_value	String(250)	Value for the given key.

The following defaults are provided on initialization.

Table 60: system_lkup defaults

System Key	System Value	Description
version	<system version>	the version of the ProcessTracking system in use.

No custom system key/value pairs are used. This is purely for system use.

4.1.10 Tool

Tools are the applications or code bases used to run data integration processes.

Tool

This is the core table tracking integration tools.

Table 61: tool_lkup

Column Name	Column Type	Column Description
tool_id	Auto incrementing integer sequence	System key for the tool
tool_name	String(250)	Unique name of the given tool.

CHAPTER 5

How to contribute

Want to participate/contribute to ProcessTracking? Feel free to add any enhancements, feature requests, etc.

5.1 Getting Started

- Check out the project's development branch and create a new Python 3.7+ virtualenv.
- Install pipenv:

```
$ pip install pipenv
```

- Install all necessary requirements:

```
$ pipenv install
```

- Make sure you have a GitHub account
- Submit issues/suggestions to the Github issue tracker
  - For bugs, clearly describe the issue including steps to reproduce. Please include stack traces, logs, screen shots, etc. to help us identify and address the issue.
  - Please ensure that your contribution is added to the correct project (i.e. docs, workflow, etc. in process_tracker , python bug or implementation changes goes to process_tracker_python, etc.)
  - For text based artifacts, please use: Gist or Pastebin
  - For enhancement requests, be sure to indicate if you are willing to work on implementing the enhancement
  - Fork the repository on GitHub if you want to contribute code/docs

5.2 Making Changes

- **ProcessTracking** uses git-flow as the git branching model

- **All commits should be made to the dev branch**
- Install git-flow and create a *feature* branch with the following command:

```
$ git flow feature start <name of your feature>
```

- Make commits of logical units with complete documentation.
  - Check for unnecessary whitespace with *git diff –check* before committing.
  - Make sure you have added the necessary tests for your changes.
  - Test coverage is currently tracked via coveralls.io
  - Aim for 100% coverage on your code
    - If this is not possible, explain why in your commit message. This may be an indication that your code should be refactored.
- To make sure your tests pass, run:

```
$ python setup.py test
```

- If you have the *coverage* package installed to generate coverage data, run:

```
$ coverage run --source=process_tracker_python setup.py test
```

- Check your coverage by running:

```
$ coverage report
```

5.3 Submitting Changes

- Push your changes to the feature branch in your fork of the repository.
- Submit a pull request to the main repository
- You will be notified if the pull was successful. If there are any concerns or issues, a member of the ProcessTracker maintainer group will reach out.

5.4 Additional Resources

- General GitHub documentation
- GitHub pull request documentation

CHAPTER 6

Coding Standards

Generally, we will follow the best practices of the given language/tool ProcessTracker is being implemented in.

Unit testing is required for any code to be submitted for peer review/admission into any given implementation.

6.1 Python

- Code formatter is Black .
- Unit testing framework is currently the built in Python unittest library.

CHAPTER 7

Indices and tables

- genindex
- modindex
- search