sqltask

Ville Brofeldt

Feb 08, 2020

CONTENTS:

SqTtask is an extensible ETL library based on SqlAlchemy to help build robust ETL pipelines with high emphasis on data quality.

CHAPTER

ONE

SUPPORTED ENGINES

SqlTask supports all databases that have a SqlAlchemy dialect and driver. The following engines have dedicated support for the following insert modes:

Database	Insert mode			
	Single	Multirow	CSV	Parquet
BigQuery	Yes	Yes	Yes	
Postgres	Yes	Yes	Yes	
Snowflake	Yes	Yes	Yes	
Sql Server	Yes	Yes		
Sqlite	Yes	Yes		

Engines not listed above will default to using multirow inserts if supported, falling back to single row inserts as a last resort.

1.1 Engine customization

engine_params: Optional dict of parameter that get destructured as keyword arguments for create_engine call in the engine. Allows customization of engine connection pool and such.

For example, to set engine to use a connection pool size of one, pass engine_params = {“pool_size”: 1}

CHAPTER

TWO

TUTORIAL - CREATING A SIMPLE ETL TASK

This tutorial shows how to construct a typical ETL task. A fully functional example can be found in the main repo of SqlTask: https://github.com/villebro/sqltask/tree/master/example

2.1 Introduction

When creating a task, you will start by extending the *sqltask.SqlTask* class. A task constitutes the the sequential execution of the following stages defined as methods:

1. *__init__(**kwargs)*: define the target table(s), row source(s) and lookup source(s). *kwargs* denote the batch parameters based on which a single snapshot is run, e.g. *report_date=date(2019, 12, 31)*.
2. *transform()*: All transformation operations, i.e. reading inputs row-by-row and mapping values (transformed or not) to the output columns. During transformation data can be read from multiple sources, and can be mapped to multiple output tables, depending on what the transformation logic is. During transformation row-by-row data quality issues can be logged to the output table if using the *DqTableContext* target table class.
3. *validate()*: Post transformation data validation step, where the final output rows can be validated prior to insertion. In contrast to the data quality logging in the transform phase, validation should be done on an aggregate level, i.e. checking that row counts are in line with what is acceptable, null counts are acceptable etc.
4. *delete_rows()*: If an exception hasn't been raised before this step, the rows corresponding to the batch parameters will be deleted from the target table. If the task is defined to have one batch parameter *report_date*, this step in practice issues a *DELETE FROM tbl WHERE report_date = 'yyyy-mm-dd'* statement.
5. *insert_rows()*: This step inserts any rows that have been appended to the output tables using whichever insertion mode has been specified. Generic SqlAlchemy drivers will fall back to single or multirow inserts if supported, but engines with dedicated upload support will perform file-based uploading.

2.2 Base task

For DAGs consisting of multiple tasks, it is commonly a good idea to create a base task on which all tasks in the DAG are based, fixing the batch parameters in the constructor as follows:

```
from datetime import date

from sqltask import SqlTask


class MyBaseTask(SqlTask):
    def __init__(report_date: date):
        super().__init__(report_date: report_date)
```

This way developers will have less ambiguity on which parameters the DAG tasks are based on. For a regular batch task this is usully the date of the snapshot in question. It is also perfectly fine to have no parameters or multiple parameters. Typical scenarios:

- No parameters: Initialization of static dimension tables
- Single parameter: Calculation of a single snapshot, typically the snapshot date
- Multiple parameters: If data is further partitioned, it might be feasible to split up the calculation into further batches, e.g. per region, per hour.

In this example, the the unit of work for the task constitutes creating a single snapshot for a certain *report_date*.

2.3 Creating an actual task

In the following example, we will construct a task that outputs data into a single target table, reads data from a SQL query and uses a CSV table as a lookup table. The class is based on *MyBaseTask* defined above. We will do the following

- Define a target table *my_table* based on *DqTableContext* into which data is inserted.
- Define a *SqlRowSource* instance that reads data from a SQL query.
- Define a *CsvLookupSource* instance that is used as a lookup table.

We have chosed *DqTableContext* as our target table class, as it can be used for logging data quality issues. If we have our primary row data in CSV format, we could also have used a *CsvRowSource* instance as the primary data source. Similarly we could also use *SqlLookupSource* to construct our lookup table from a SQL query.

```
class MyTask(MyBaseTask):
    def __init__(self, report_date: date):
        super().__init__(report_date)

        # Define the metadata for the main fact table
        self.add_table(DqTableContext(
            name="my_table",
            engine_context=self.ENGINE_TARGET,
            columns=[
                Column("report_date", Date, comment="Date of snapshot", primary_
↪key=True),
                Column("etl_timestamp", DateTime, comment="Timestamp when the row was␣
↪created", nullable=False),
                Column("customer_name", String(10), comment="Unique customer␣
↪identifier (name)", primary_key=True),
                Column("birthdate", Date, comment="Birthdate of customer if defined␣
↪and in the past", nullable=True),
                Column("age", Integer, comment="Age of customer in years if birthdate␣
↪defined", nullable=True),
                Column("blood_group", String(3), comment="Blood group of the customer
↪", nullable=True),
            ],
            comment="The customer table",
            timestamp_column_name="etl_timestamp",
            batch_params={"report_date": report_date},
            dq_info_column_names=["etl_timestamp"],
        ))
```

TBC

CHAPTER

THREE

CLASSES

class sqltask.base.engine.**EngineContext**(*name*, *url*, *engine_params=None*, *metadata_params=None*)

create_new(*database*, *schema*)

Create a new EngineContext based on the current instance, but with a different schema.

Parameters

- **database** (Optional[str]) – Database to use. If left unspecified, falls back to the database provided by the original engine context
- **schema** (Optional[str]) – Schema to use. If left unspecified, falls back to the schema provided by the original engine context

Return type *EngineContext*

Returns a new instance of EngineContext with different url

class sqltask.base.table.**BaseTableContext**(*name*, *engine_context*, *columns*, *comment=None*, *database=None*, *schema=None*, *batch_params=None*, *timestamp_column_name=None*, *table_params=None*)

The BaseTableContext class contains everything necessary for creating/modifying a target table/schema and inserting/removing rows.

delete_rows()

Delete old rows from target table that match batch parameters.

Return type None

get_new_row()

Get a new row intended to be added to the table.

Return type *BaseOutputRow*

insert_rows()

Insert rows into target tables.

Return type None

map_all(*row_source*, *mappings=None*, *funcs=None*)

Convenience method for mapping all rows and columns from the input row source to the output table in a one-to-one fashion. The optional arguments *mappings* and *funcs* can be used to specify alternative column name mappings and conversion functions.

Parameters

- **row_source** (*BaseRowSource*) – Input row source to map to the outout table.

- **mappings** (Optional[Dict[str, str]]) – mapping from target column name to source column name. If the source and target names differ for one or several columns, these can be specified here. Example: {"customer_name": "cust_n"} would map the values in the source column "cust_n" to the target column "customer_name".
- **funcs** (Optional[Dict[str, Callable[[Any], Any]]]) – mapping from target column name to callable function. If the source and target types differ for one or several columns, a callable can be specified here. Typically this is needed when ingesting data from a CSV file where the source data types are always strings, but might need to be cast to int, float or Decimal. Example: {"customer_age": int} would call *int()* on the source value.

Return type None

migrate_schema()
Migrate table schema to correspond to table definition.

Return type None

class sqltask.base.table.**BaseOutputRow**(*table_context*)
A class for storing cell values for a single row in a TableContext table. When the object is created, all batch parameters are prepopulated.

append()
Append the row to the target table. append() should only be called once all cell values for the row have been fully populated, as any changes.

Return type None

map_all(*input_row*, *mappings=None*, *funcs=None*, *columns=None*, *auto_append=False*)
Convenience method for mapping column values one-to-one from an input row to the output row. Will only map any unmapped columns, i.e. if the target row has columns "customer_id" and "customer_name", and "customer_name" has already been populated, only "customer_id" will be mapped.

Parameters

- **input_row** (Mapping[str, Any]) – the input row to map values from.
- **mappings** (Optional[Dict[str, str]]) – mapping from target column name to source column name. If the source and target names differ for one or several columns, these can be specified here. Example: {"customer_name": "cust_n"} would map the values in the source column "cust_n" to the target column "customer_name".
- **funcs** (Optional[Dict[str, Callable[[Any], Any]]]) – mapping from target column name to callable function. If the source and target types differ for one or several columns, a callable can be specified here. Typically this is needed when ingesting data from a CSV file where the source data types are always strings, but might need to be cast to int, float or Decimal. Example: {"customer_age": int} would call *int()* on the source value.
- **columns** (Optional[Sequence[str]]) – A list of column names to map. If undefined, tries to map all unmapped columns in target row.
- **auto_append** (bool) – Call append after mapping rows if the mapping operation is successful.

Return type None

class sqltask.base.table.**DqTableContext**(*name*, *engine_context*, *columns*, *comment=None*, *schema=None*, *batch_params=None*, *timestamp_column_name=None*, *table_params=None*, *dq_table_name=None*, *dq_engine_context=None*, *dq_schema=None*, *dq_info_column_names=None*, *dq_table_params=None*)

A `TableContext` child class with support for logging data quality issues to a separate data quality table. A with the ability to log data quality issues

delete_rows()

Delete old rows from target table that match batch parameters.

Return type `None`

get_new_row()

Get a new row intended to be added to the table.

Return type *DqOutputRow*

insert_rows()

Insert rows into target tables.

Return type `None`

migrate_schema()

Migrate table schema to correspond to table definition.

Return type `None`

class sqltask.base.table.**DqOutputRow**(*table_context*)

log_dq(*column_name*, *category*, *priority*, *source*, *message=None*)

Log data quality issue to be recorded in data quality table. If logging has been disabled by calling *set_logging_enabled(False)*, data quality issues will be ignored.

Parameters

- **column_name** (`Optional[str]`) – Name of affected column in target table.
- **category** (`Category`) – The type of data quality issue.
- **source** (`Source`) – To what phase the data quality issue relates.
- **priority** (`Priority`) – What the priority of the data quality issue is. Should be None for aggregate data quality issues.
- **message** (`Optional[str]`) – Verbose description of observed issue.

Return type `None`

set_logging_enabled(*enabled*)

If logging is set to false, data quality issues will not be passed to the log table. This is useful for rows with lower priority data, e.g. inactive users, whose data quality may be of poorer quality due to being stale.

Parameters **enabled** (`bool`) – set to True to log issues; False to ignore calls to *log_dq*`

Return type `None`

class sqltask.base.lookup_source.**BaseLookupSource**(*name*, *row_source*, *keys*)

get(**unnamed_keys*, ***named_keys*)

Get a value from the lookup. Assuming the key for a Lookup is key1, key2, key3, the following are valid calls:

```
>>> # only unnamed keys
>>> lookup.get("val1", "val2", "val3")
>>> # only named keys in non-original order
>>> lookup.get(key3="val3", key1="val1", key2="val2")
>>> # both named and unnamed keys
>>> lookup.get("val1", key3="val3", key2="val2")
```

If a row is not found in the lookup table, the method returns an empty dict.

Parameters

- **unnamed_keys** – unnamed key values to be used as keys
- **named_keys** – named key values to be used as keys

Return type Dict[str, Any]

Returns A dict with keys as the column name and values as the cell values. If key undefined in internal dict return an empty dict.

class sqltask.base.row_source.**BaseRowSource**(*name=None*)

Base class for data sources that return iterable rows. A row from a BaseRowSource can be any Mapping from a key (=column name) to a value (=cell value) that can be referenced as follows: >>> for row in rows: >>> column_value = row[“column_name”]

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search

INDEX