YETL

Yet Another ETL Framework

Table Configuration

The table configuration defines that is loaded in a data pipeline.

The table metadata is the most difficult and time consuming metadata to curate. Therefore yetl provides a command line tool to convert an excel curated definition of metadata into the required yaml format. Curating this kind of detail for large numbers of tables is much easier to do in an Excel document due to it's excellent features.

Example

A solution lands 3 files:

- `customer_details_1`
- `customer_details_2`
- `customer_preferences`

The files are loaded into raw from landing with a deltalake table for each file.

Those tables are then loaded into base tables. `customer_details_1` and `customer_details_2` are unioned together and loaded into a customer table. So the base tables are:

- `customers`
- `customer_perferences`

Each file has a header and footer with some audit data we load this with some other etl audit data into deltalake audit tables:

- `header_footer`
- `raw_audit`
- `base_audit`

Here is the `tables.yaml` metadata that describes the stages, databases and tables:

```yaml
# yaml-language-server: $schema=./json_schema/sibytes_yetl_tables_schema.json

version: 3.0.0

audit_control:
  delta_lake:
    yetl_control_header_footer_uc:
      catalog: development
      base_audit:
        depends_on:
        - raw.yetl_raw_header_footer_uc.*
        sql: ../sql/{{database}}/{{table}}.sql
        vacuum: 168
      header_footer:
        depends_on:
        - raw.yetl_raw_header_footer_uc.*
        sql: ../sql/{{database}}/{{table}}.sql
        vacuum: 168
      raw_audit:
        depends_on:
        - raw.yetl_raw_header_footer_uc.*
        - audit_control.yetl_control_header_footer_uc.header_footer
        sql: ../sql/{{database}}/{{table}}.sql
        vacuum: 168

landing:
  read:
    yetl_landing_header_footer_uc:
      catalog: development
      customer_details_1: null
      customer_details_2: null
      customer_preferences: null

raw:
  delta_lake:
    yetl_raw_header_footer_uc:
      catalog: development
      customer_details_1:
        custom_properties:
          process_group: 1
          rentention_days: 365
        depends_on:
        - landing.yetl_landing_header_footer_uc.customer_details_1
        exception_thresholds:
          invalid_rows: 2
          min_rows: 1
        id: id
        vacuum: 168
        z_order_by: _load_date
      customer_details_2:
        custom_properties:
          process_group: 1
          rentention_days: 365
        depends_on:
        - landing.yetl_landing_header_footer_uc.customer_details_2
        exception_thresholds:
          invalid_rows: 2
          min_rows: 1
        id: id
        vacuum: 168
        z_order_by: _load_date
      customer_preferences:
        custom_properties:
```

```
          process_group: 1          rentention_days: 365        depends_on:        -
landing.yetl_landing_header_footer_uc.customer_preferences        exception_thresholds:
invalid_rows: 2          min_rows: 1        id: id        vacuum: 168        z_order_by:
_load_datebase:  delta_lake:    yetl_base_header_footer_uc:      catalog: development
customer_details_1:        custom_properties:          process_group: 1          rentention_days:
365        depends_on:        - raw.yetl_raw_header_footer_uc.customer_details_1
exception_thresholds:          invalid_rows: 0          min_rows: 1        id: id        vacuum:
168        z_order_by: _load_date      customer_details_2:        custom_properties:
process_group: 1          rentention_days: 365        depends_on:        -
raw.yetl_raw_header_footer_uc.customer_details_2        exception_thresholds:
invalid_rows: 0          min_rows: 1        id: id        vacuum: 168        z_order_by:
_load_date      customer_preferences:        custom_properties:          process_group: 1
rentention_days: 365        depends_on:        -
raw.yetl_raw_header_footer_uc.customer_preferences        exception_thresholds:
invalid_rows: 0          min_rows: 1        id: id        vacuum: 168        z_order_by:
_load_date      customers:        custom_properties:          process_group: 1
rentention_days: 365        depends_on:        - raw.yetl_raw_header_footer_uc.*
exception_thresholds:          invalid_rows: 0          min_rows: 1        id: id        vacuum:
168        z_order_by: _load_date
```

Specification

If you use the `yetl` cli to create a project using `python -m yetl init <my_project>` then the json validation schema for the config files including table the table config will be created at `./<my_project>/piplines/json-schemas/sibytes_yetl_tables_schema.json`. Using vscode and the RedHat yaml extension you can add the following json schema reference to `./<my_project>/piplines/tables.yml` to provide live validation and intellisense:

```
# yaml-language-server: $schema=./json_schema/sibytes_yetl_tables_schema.json
```

This reference describes the required format of the `tables.yaml` configuration.

```
version: major.minor.patch

<stage:Stage>:
    <table_type:TableType>:
        delta_properties:
          <property_name>: str
        <database_name:string>:
            catalog: str|null
            <table_name:str>:
                id: str|list[str]|null
                depends_on: index|list[index]|null
                delta_properties:
                  <property_name>: str
                delta_constraints:
                  <constraint_name>: str
                custom_properties:
                  <property_names>: str
                z_order_by: str|list[str]|null
                partition_by: str|list[str]|null
                cluster_by: str|list[str]|null
                vacuum: int|null
                sql: path|null
                warning_thresholds:
                    invalid_ratio: float
                    invalid_rows: int
                    max_rows: int
                    min_rows: int
                exception_thresholds:
                    invalid_ratio: float
                    invalid_rows: int
                    max_rows: int
                    min_rows: int

            <table_name:str>:
              # table details
              ...
            <table_name:str>:
              # table details
              ...
```

version

Version is the version number of yetl that the metadata is compatible with. If the major and minor version are not the same as the yetl python libary that you're using to load the metadata then an error will be raised. This is to ensure the metadata is compatible with the version of yetl that you're using.

Example:

```
version: 3.0.0
```

Stage

The stage of the datalake house architecture. Yetl supports the following `stage`s:

- `audit-control` - define tables for holding etl data and audit logs

- `landing` - define landing object store where files are copied into you your cloud storage before they uploaded into the delta lakehouse
- `raw` - define databases and tables for the bronze layer of the datalake. These will typically be deltalake tables loading with landing data with nothing more than schema validation applied
- `base` - define databases and deltalake tables for the silver layer of the datalake. These tables will hold data loaded from raw with data quality and cleansing applied.
- `curated` - define databases and deltalake tables for the gold layer of the datalake. These tables will hold the results of heavy transforms that integrate and aggregate data using complex business transformations specifically for business requirements.

At least 2 stages must defined:

- `landing`
- `raw`

These stages are optional:

- `audit_control`
- `base`
- `curated`

Example:

```
audit_control:
  delta_lake:
...
landing:
  read:
...
raw:
  delta_lake:
```

delta_properties

Deltalake properties is an object of key-value pairs that describes the deltalake properties. They can be defined at the table type level or the table level. The lowest level of granularity takes precedence over the higher levels. So you can define properties at a high level but override them at the table level if a table has specific properties that need to be defined.

Example:

```
delta_properties:
  delta.appendOnly: true
  delta.autoOptimize.autoCompact: true
  delta.autoOptimize.optimizeWrite: true
  delta.enableChangeDataFeed: false
```

delta_constraints

Deltalake properties is an object of key-value pairs that describes the deltalake constraints. The key is the constraint name and the value is the sql constraint.

The constraints are added when yetl creates the tables.

```
delta_properties:
  dateWithinRange: "(birthDate > '1900-01-01')"
  validIds: "(id > 1 and id < 99999999)"
```

TableType

Table type is the type of table that is used. Yetl supports the following `table_type`s:

- `read` - These are tables that are read using the spark read data api. Typically these are files with various formats. These types of tables are typically defined on the `landing` stage of the datalake.
- `delta_lake` - These are deltalake tables that written to and read from during a pipeline load.

Example:

```
audit_control:
  delta_lake:
...
landing:
  read:
...
raw:
  delta_lake:
```

index

Index is a string formatted specifically to describe a table index. In the Yetl api the tables are index and the index can be used to quickly find and define dependencies.

The index takes the following form:

```
stage.database.table
```

It supports a wild card form for defining or finding a collection of tables e.g.

- `stage.*.*` - return/configure all the tables in a stage
- `stage.database.*` - return/configure all the tables in a database

Example:

```
audit_control:
  delta_lake:
    yetl_control_header_footer:
      base_audit:
        # this table depends on all the tables in the raw database called yetl_raw_header_footer
        depends_on:
        - raw.yetl_raw_header_footer.*
```

id

`id` is a string or list of strings that is the columns name or names of the table uniqie identifier.

z_order_by

`z_order_by` is a string or list of strings that is the columns name or names to z_order the table by.

partition_by

`partition_by` is a string or list of strings that is the columns name or names to partition the table by.

sql

`sql` is relative path to the directory that holds a file container the explicit SQL to create the table. Note that jinja varaiable can be used for database and table thus defining that the sql directory is structured by database and table.

Example:

```
sql: ../sql/{{database}}/{{table}}.sql
```

thresholds

Thresholds allow to define ETL audit metrics for each table. There are 2 properties for this:

- warning_thresholds - used to define metrics that if exceeded raises a warning
- exception_thresholds - usde to define metrics that if exceeded raises an exception

This is just metadata so how you use it and handle this metadata is entirely down to the developmer however. The pipeline code it self is used to calculate what these values are and compare them to the these thresholds and take appropriate action.

Each threshold type supports the following metrics:

- `invalid_ratio` - number of invalid records divided by the total number of records
- `invalid_rows` - number of invalid records
- `min_rows` - minimum number of rows
- `max_rows` - maximum number of rows

Example:

```
warning_thresholds:
    # if more than 10% of the rows are invalid then raise a warning.
    invalid_ratio: 0
    invalid_rows: 0
    max_rows: null
    # if there's less than 1000 records raise a warning
    min_rows: 1000
exception_thresholds:
    # if more than 50% of the rows are invalid then raise an exception
    invalid_ratio: 0.5
    invalid_rows: null
    max_rows: null
    # if there's less than 1 record raise an exception
    min_rows: 1
```

vacuum

`vacuum` is the day threshold over which to apply the vacuum statement.

custom_properties

`custom_properties` is object of key value pairs for anything that you want to define that's not in the specification. This feature allows yetl to be very flexible for any additional requirement that you may have.

Example:

```
custom_properties:
    # define an affinity group to process tables on the same job clusters
    process_group: 1
    # define the days to retain the data for after which it is archived or deleted
    rentention_days: 365
```

Yet Another ETL Framework