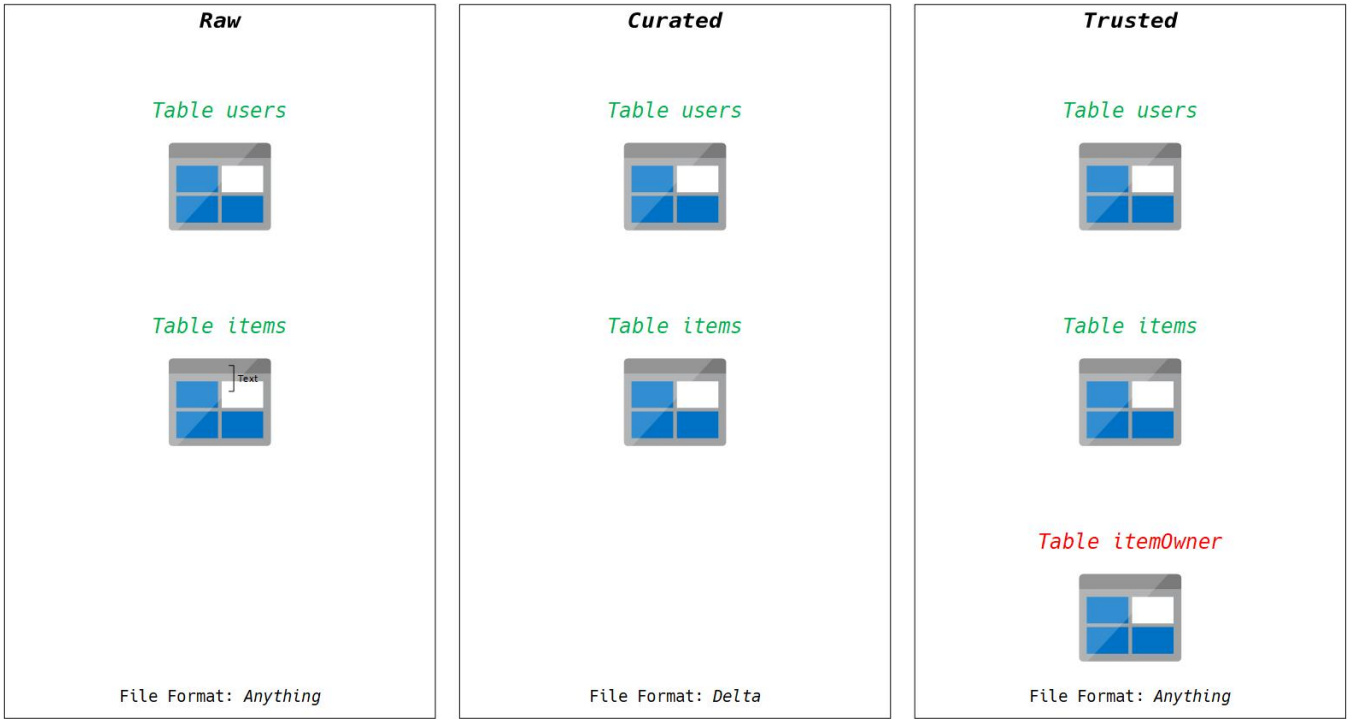
Concepts

Standards

Standards are simply a set of, agreed on, rules for the ETL pipelines and files/folders naming in the Data Lake. This library's*standard functions (**ETL.raw**,**ETL.curated**and**ETL.trusted**)*enforces those rules while functions from other modules are as unopinionated as possible and only help with things like mounting, configuration or writing/reading spark dataframes.

Standard tables

Generally, a "**standard**" table is simply a way to call tables that are present in all zones of the data lake storage (*raw*,*curated*and*trusted*). They do not have to follow each and every standard rules.



The tables *users* and *items*have good chances of being "standard" but not the table *itemOwner* becauseits made from a mix of different tables and data sources.

Functions

Many functions wrap spark'sreadand/orwriteAPIs and so they accept all options that spark supports.

Standard functions might set different options than the default spark ones so that they can work as intended. For example, spark's default save mode *"appe nd"* and *"overwrite"*areignored since the library's implements its own incrementalmode.

Lower-level functions aim at keeping default options and behaviors as similar to original spark APIs as possible.

Standard functions

Recognizing when a table could be standard have the benefit of allowing the use of functions from the **ETL.raw**, **ETL.curated** and **ETL.trusted** modules, some of the library's most useful function to save development time by handling common operations and by providing*incremental loading*features all while following standard rules such as files location.

- Handles where the data is saved, paths always follow the same standards to keep things organized and so that it is as easy as possible for humans to find what they are looking for.
- The library checks if the volumes are mounted and usable only when they are really needed to avoid errors and time loss due to unnecessary mounting.
- Incrementaland differential loads are supported with the*incremental*parameter, metadata of past ingestions are always saved and handled automatically.
- Uses the Delta lake APIs for the curated zone.
- Validates paths, zone names and table names.
- ...

ETL.raw module	
def	read(config, table, incremental=False, **dataframe_reader_options)
def	get_table_file_paths(config, table, incremental=False)
def	get_table_folder_paths(config, table, incremental=False)

ETL.curated module	
def	read(config, table, incremental=False, **raw_dataframe_reader_options)
def	write(config, table, df=None, transformation=None, incremental=False, raw_dataframe_reader_options=None, **dataframe_writer_options)
def	merge(config, table, df=None, transformation=None, unique_key=None, incremental=False, raw_dataframe_reader_options=None)
def	delete(config, table, drop=False)

ETL.trusted module	
def	read(config, table, **dataframe_reader_options)
def	write(config, table, num_files=10, df=None, transformation=None, incremental=False, **dataframe_writer_options)

- Note that all standard functions must take a table argument, the table must be specified inconfig's params.
- The *unique_key* and *incremental* parameters can be specified in theconfig's params.
- The *transformation* parameter is a Transform object.

Example of a standard table operation: Merging raw ingestion data into the curated zone.

Merging / upsertingneeds a *unique key* so it can update any existing rows or insert new ones based on that column value.

config and imports

```
from ETL import Config, raw, curated, trusted


params = {
  'data_source': 'yammer',
  'tables': {
    'MessagesLikes': {
      'unique_key': ['id', 'message_id'],
      'trusted_incremental_mode': True
    },
    'Messages': {
      'unique_key': 'id',
      'trusted_incremental_mode': False
    },
    'Users': {
      'unique_key': 'id',
      'trusted_incremental_mode': True
    },
    'Groups': {
      'unique_key': 'id',
      'trusted_incremental_mode': False
    }
  }
}

config = Config(params)
```

In our use case, the table users will be in all 3 zones so it is illegible for standard functions. As such, we can use the *ETL.curated.merge*function to handle this common pattern of transferring data from the raw zone to the curated zone.
With the assumption that the table exists in the raw zone, we can then use the *ETL.curated* module without worrying about how to read the raw ingestion data.

Simply specify which table to load (Users), that the raw data files are in the CSV format and that the separator is the "pipe" character to atomically update the curated table with that extract.

```
curated.merge(
        config=config,
        table='Users',
    incremental=True,
        raw_dataframe_reader_options={'format': 'csv', 'sep': '|'}
)
```

The *incremental*parameter always affects what is loaded source-side.

- True: Merge the curated zone with all the raw data.
- False: Merge the curated zone with only new raw data.

Other Functions

Use these functions to handle other use cases or when higher-level, standard functions cannot be used.

What the previous standard functions do is they simply use those, more general-purpose, functions by prefilling all the necessary parameters. This is possible because it is assumed that a set of *standard rules apply*.

ETL.dataframe_utils	
def	write(config, df, zone, path, file_format, num_files=None, check_df=True, **dataframe_writer_options)
def	read(config, zone, path, **dataframe_reader_options)

def	groupby_and_to_list(df, groupby_col_name, new_col_name='x')

ETL.delta_utils

def	read(config, zone, path, **dataframe_reader_options)
def	write(config, df, zone, path, table=None, **dataframe_writer_options)
def	merge(config, zone, table, df, path, unique_key)
def	delete(config, zone, table, path, database_name=None, drop=False)
def	create_if_not_exists(config, table, schema, location=None)