Documentation of Databricks ETL Jobs

Dependencies:

- `pyspark.sql` - Spark SQL module for working with structured data.
- `simple_salesforce` - A Python library for interacting with Salesforce APIs.
- `pyspark.sql.functions` - Functions for working with Spark DataFrame columns.
- `pyspark` - The core Spark module.
- `pyspark.sql.types` - Provides the data types available in Spark.
- `json` - JSON encoder and decoder for Python.
- `snowflake.connector` - Python connector for Snowflake.
- `os` - Provides a way of using operating system dependent functionality.
- `datetime` - Supplies classes for working with dates and times.
- `hmac` - Implements the HMAC algorithm as described in RFC 2104.
- `hashlib` - Contains secure hash algorithms.
- `urllib.parse` - Provides functions for parsing URLs.

▼ **Using Databricks Secrets**

To set up secrets you:

1. Create a secret scope. Secret scope names are case-insensitive.
2. Add secrets to the scope. Secret names are case-insensitive.
3. If you have the Premium plan or above, assign access control to the secret scope.

Concrete Example below →

Installing and Initialising Databricks CLI

```
pip3 install databricks-cli
databricks --version
```

Create Databricks scope

```
databricks secrets create-scope --scope my_scope
```

List all secret copes

```
databricks secrets list-scopes
```

Adding secrets in existing scope

```
databricks secrets put --scope my_scope --key some_key --string-value some_value
```

List secrets in a scope

```
databricks secrets list --scope my_scope
```

▼ Main Migration Job

This technical documentation provides an overview of the various functions of the jobs and their parameters. It aims to help understand the purpose and functionality of each function, making it easier to use and maintain the code.

Function: `anonymise(tab: str, pg_df: 'DataFrame') -> 'DataFrame'`

This function is responsible for anonymizing sensitive data in the given DataFrame based on the specified table name.

- `tab`: A string representing the table name to determine the anonymization logic.
- `pg_df`: The input DataFrame containing the data to be anonymized.

The function applies different anonymization techniques based on the provided `tab` value. It modifies the DataFrame by adding new columns or replacing existing columns with anonymized values and returns the updated DataFrame.

Function: `read_src_pg_table(url: str, query: str, properties: dict) -> 'DataFrame'`

This function reads data from a PostgreSQL database table and returns a DataFrame.

- `url`: The URL of the PostgreSQL database.
- `query`: The SQL query to select the data from the table.
- `properties`: A dictionary containing the connection properties.

The function establishes a connection to the PostgreSQL database, reads the specified table using the provided query and properties, and returns the data as a DataFrame.

Function: `read_tgt_snw_table(query: str, snow_options: dict) -> 'DataFrame'`

This function reads data from a Snowflake database table and returns a DataFrame.

- `query`: The SQL query to select the data from the table.
- `snow_options`: A dictionary containing the Snowflake connection options.

The function establishes a connection to the Snowflake database, reads the specified table using the provided query and options, and returns the data as a DataFrame.

Function: `get_all_tabs_to_migrate(url: str, get_all_tables_pg: str, pg_options: dict) -> list`

This function retrieves all the database tables to migrate from a PostgreSQL database.

- `url`: The URL of the PostgreSQL database.
- `get_all_tables_pg`: The SQL query to select all the tables from the database.
- `pg_options`: A dictionary containing the connection options.

The function connects to the PostgreSQL database, executes the query to retrieve all table names, and returns a list of table names.

Function: `check_schema_difference(table_to_check: str) -> bool`

This function checks for schema differences between the source (Postgres) table and the target (Snowflake) table.

- `table_to_check`: The name of the table to compare schemas.

The function retrieves the column metadata from the source and target tables, applies a data type mapping, and compares the schemas. If any differences are found, the function generates a string describing the differences and appends it to `msg_to_send` list. It returns a boolean value indicating whether there are schema differences or not.

Function: `handle_schema_change(data_df: 'DataFrame', table_name: str, versioning_schema: str) -> None`

This function handles schema changes in the target (Snowflake) table.

- `data_df`: The DataFrame containing the updated data.
- `table_name`: The name of the target table.
- `versioning_schema`: The schema name for versioning.

The function creates a new table with a timestamp in its name and saves the updated data. It then renames the old table and renames the new table to replace the old table. The function performs these operations within a transaction and handles any errors that occur.

Function: `upsert_target_table(tab: str, merge_key: str = None) -> bool`

This function performs an upsert (merge) operation on the target table.

- `tab`: The name of the target table.
- `merge_key`: The column used as the merge key. Defaults to 'id' if not specified.

The function retrieves the column names of the target table and generates a MERGE query to perform the upsert operation. It executes the query using a Snowflake cursor and returns a boolean value indicating the success of the operation.

Function: `load_tgt_snw_table(data_df: 'DataFrame', db_table: str, snow_options: dict, write_mode: str, update_flag: bool) -> None`

This function saves data to a Snowflake database table.

- `data_df`: The DataFrame containing the data to be saved.
- `db_table`: The name of the target database table.
- `snow_options`: A dictionary containing the Snowflake connection options.
- `write_mode`: The write mode to determine how the data is saved ('overwrite' or 'append').
- `update_flag`: A flag indicating whether the target table should be updated.

The function establishes a connection to the Snowflake database and saves the data to a temporary table or directly to the target table based on the update flag and write mode. If the update flag is set, it performs an UPSERT operation on the target table.

Function: `send_mail(subject: str, message: str, link: str = None) -> None`

This function sends an email using the SMTP protocol.

- `subject`: The subject line of the email.
- `message`: The body content of the email.
- `link`: An optional link to include in the email.

The function uses the provided email credentials to authenticate and sends the email with the specified subject, message, and link (if provided).

Table Migration Process

This code is designed to migrate tables from a source PostgreSQL database to a target Snowflake database. It follows a specific process to migrate the tables and handles various scenarios such as table existence checks, schema differences, and incremental data updates.

Overview

The migration process can be summarized as follows:

1. Check if the tables specified in the `add_to_created_at` list exist in the `all_pg_tabs` list, which contains all the source tables.
2. If a table exists in both lists, append it to the `all_created_at` list.
3. Iterate through each table in the `all_created_at` list and migrate them to the target Snowflake database.
4. During the migration, handle different scenarios based on table conditions, such as new tables, schema differences, and incremental data updates.

Migration Steps

1. **Table Existence Check**
    - The code checks if the tables specified in the `add_to_created_at` list exist in the `all_pg_tabs` list using list comprehension and conditional check.

- Tables that exist in both lists are appended to the `all_created_at` list.

2. **Date Column Mapping**
   - A dictionary named `date_col` is defined, which maps each table name to its corresponding date column.
   - The date columns are used later for determining the maximum date value during incremental data updates.
3. **Table Migration Loop**
   - Iterate through each table in the `all_created_at` list.
   - Print a message indicating the migration of the current table.
   - Set the initial maximum column to `'created_at'`.
   - If the current table is also present in the `add_to_created_at` list, update the maximum column using the corresponding date column from the `date_col` dictionary.
4. **Table Migration Process**
   - Check if the current table is not present in the `all_snw_tabs` list, which represents the already migrated tables in Snowflake.
   - If the table is not present, perform the following steps:
     - Construct a PostgreSQL query to fetch the table data using the table name and maximum column.
     - Read the source PostgreSQL table using the `read_src_pg_table` method and cache the resulting DataFrame in memory.
     - If the DataFrame is empty (no rows), continue to the next table.
     - Load the DataFrame into the target Snowflake table using the `load_tgt_snw_table` method with the "overwrite" write mode.
     - Unpersist (remove from memory) the DataFrame.
     - Append a migration success message for the current table to the `msg_to_send` list.
5. **Handling Schema Differences**

- If the current table is present in the `all_snw_tabs` list, indicating it has already been migrated, check for schema differences using the `check_schema_difference` method.
- If there are schema differences, perform the following steps:
    - Construct a PostgreSQL query to fetch the table data using the table name.
    - Read the source PostgreSQL table using the `read_src_pg_table` method and cache the resulting DataFrame in memory.
    - If the DataFrame is empty (no rows), continue to the next table.
    - Handle the schema change using the `handle_schema_change` method, passing the DataFrame, table name, and a temporary schema name.
    - Unpersist the DataFrame.

6. **Incremental Data Updates**
    - If there are no schema differences, fetch the maximum date value for the current table from the source PostgreSQL database using a cursor.
    - If a maximum date value exists, construct an extension query based on the table name and inclusion/exclusion conditions.
    - Read the source PostgreSQL table using the `read_src_pg_table` method, applying the maximum date filter and extension query, if applicable.
    - Cache the resulting DataFrame in memory.
    - Anonymize the data in the DataFrame using the `anonymise` method.
    - Load the DataFrame into the target Snowflake table using the `load_tgt_snw_table` method with the "overwrite" write mode and the "update" flag.
    - Unpersist the DataFrame.
7. **Error Handling and Reporting**
    - If a maximum date value is None, indicating no specific date to filter source rows, append an error message for the current table to the `msg_to_send` list.

- After processing all tables, print the total number of tables to migrate and the actual number of tables migrated.
- If the total and actual counts do not match, append an error message to the `msg_to_send` list.

8. **Message Notification**
    - At the end of the code, if there are any messages in the `msg_to_send` list, they can be sent as notifications.

Conclusion

This code provides a systematic approach to migrating tables from a source PostgreSQL database to a target Snowflake database. It handles table existence checks, schema differences, and incremental data updates based on date columns. The code also includes error handling and message notifications to report any issues encountered during the migration process.

▼ **Salesforce Migration Job**

Technical Documentation: Code for Salesforce to Snowflake ETL

This technical documentation explains the code provided, which performs an ETL (Extract, Transform, Load) process from Salesforce to Snowflake using PySpark and the Simple Salesforce and Snowflake connectors.

Function: read_src_sf_table(table: str, query_extension: str = None) -> 'DataFrame'

Description: This function reads data from a Salesforce object (table) using the Salesforce Python library. It retrieves the object description, reads the schema, and queries the data using SOQL. It then extracts the rows from the retrieved records and converts the data to a PySpark DataFrame.

Components:

- `table` (str): The name of the Salesforce object (table) to read data from.

- `query_extension` (str, optional): An optional extension to the SOQL query. Default value is `None`.

Returns: PySpark DataFrame containing the data read from the Salesforce object.

Function: read_tgt_snw_table(query: str, snow_options: dict) -> 'DataFrame'

Description: This function reads data from a Snowflake database table using the Snowflake Spark connector. It takes a SQL query and Snowflake connection options as input and returns a PySpark DataFrame containing the queried data.

Components:

- `query` (str): The SQL query to retrieve data from the Snowflake table.
- `snow_options` (dict): A dictionary containing the Snowflake connection options, including URL, user, password, database, schema, warehouse, and role.

Returns: PySpark DataFrame containing the data read from the Snowflake table.

Function: get_all_tabs_to_migrate() -> list

Description: This function returns a list of all database tables to migrate. The list contains the names of the tables that are hard-coded within the function.

Components: None

Returns: List of table names to migrate.

Function: get_all_tgt_tabs_to_migrate(snow_options: dict) -> list

Description: This function retrieves all target database tables from the Snowflake database using the provided Snowflake connection options. It queries the Snowflake metadata tables to obtain the list of table names in the target database schema.

Components:

- `snow_options` (dict): A dictionary containing the Snowflake connection options, including URL, user, password, database, schema, warehouse, and role.

Returns: List of target table names to migrate.

Function: check_schema_difference(table_to_check: str) -> bool

Description: This function checks for schema differences between a source Salesforce table and a target Snowflake table. It reads the metadata of both tables, compares the column names, data types, and nullability, and detects any differences. If a schema difference is found, the function appends the details to a global list `Schema_change_vals`.

Components:

- `table_to_check` (str): The name of the table to check for schema differences.

Returns: `True` if a schema difference is found, `False` otherwise.

Function: handle_schema_change(data_df: 'DataFrame', table_name: str, versioning_schema: str) -> None

Description: This function handles schema changes detected in the source Salesforce table. It creates a new table with a timestamped name to store the new data, renames the existing table to an alternate schema, and renames the new table to replace the existing table. The function performs these operations within a transaction to ensure data integrity.

Components:

- `data_df` ('DataFrame): The DataFrame containing the new data.
- `table_name` (str): The name of the target table to handle schema changes for.
- `versioning_schema` (str): The name of the alternate schema to store the existing table.

Returns: None

Function: upsert_target_table(tab: str, merge_key: str = None) -> None

Description: This function performs an upsert operation on the target Snowflake table. It takes the name of the target table and an optional merge key as input. The function uses the Snowflake Spark connector to write the data from the provided DataFrame to the target table. If a merge key is specified, the function performs an upsert operation based on that key. Otherwise, it performs a regular insert operation.

Components:

- `tab` (str): The name of the target table to perform the upsert operation on.
- `merge_key` (str, optional): An optional merge key to perform an upsert operation. Default value is `None`.

Returns: None

Core Migration Method

This code segment represents the core migration method responsible for migrating tables from a source Salesforce database to a target Snowflake database. The code follows a specific process that handles various scenarios such as table existence checks, schema differences, and incremental data updates.

Overview

The migration process can be summarized as follows:

1. Retrieve the list of Salesforce tables to migrate using the `get_all_tabs_to_migrate()` function and assign it to the `salesforce_tables` variable.
2. Retrieve the list of target Snowflake tables to migrate using the `get_all_tgt_tabs_to_migrate()` function, passing the `snow_options` dictionary, and assign it to the `all_snw_tabs` variable.
3. Define a list of Salesforce tables to be excluded from the migration process and assign it to the `deleted_salesforce_objects` variable.

4. Print the number of Salesforce tables to be migrated.
5. Remove the tables specified in the `deleted_salesforce_objects` list from the `all_snw_tabs` list.
6. Initialize an empty list called `no_created_at`.
7. Print the `all_snw_tabs` list, which represents the final list of Snowflake tables to migrate.

Table Migration Loop

The code then enters a loop that iterates over each table in the `all_snw_tabs` list. Within the loop:

1. Set the Snowflake warehouse to be used for the migration process as `'COMPUTE_WH'`.
2. Print a message indicating the migration of the current table.
3. Set the initial maximum column name to `'CreatedDate'`.
4. If the current table is `'userrole'`, update the maximum column name to `'LASTMODIFIEDDATE'`.

Table Migration Process

Within the table migration loop, the code performs the following steps:

1. Check if the current table is not present in the `salesforce_tables` list, indicating a new table that needs to be migrated.
    - If it is a new table, read the source Salesforce table using the `read_src_sf_table()` function, passing the table name, and cache the resulting DataFrame in memory.
    - Set the Snowflake warehouse to be used for the migration process as `'COMPUTE_WH'`.
    - Load the DataFrame into the target Snowflake table using the `load_tgt_snw_table()` function with the options "overwrite".
    - Continue to the next table in the loop.

2. If the current table is present in the `salesforce_tables` list, indicating a table that has already been migrated, check for schema differences using the `check_schema_difference()` function.
    - If there are schema differences, read the source Salesforce table using the `read_src_sf_table()` function, passing the table name, and cache the resulting DataFrame in memory.
    - Handle the schema change using the `handle_schema_change()` function, passing the DataFrame, table name, and a temporary schema name.
3. If there are no schema differences, proceed with incremental data updates.
    - Fetch the maximum date value for the current table from the target Snowflake table using a cursor.
    - Convert the maximum date value to the format `'YYYY-MM-DDTHH:MM:SSZ'` to be used in the Salesforce query.
    - Construct a Salesforce query to retrieve records where the date column is greater than or equal to the maximum date value.
    - Read the source Salesforce table using the `read_src_sf_table()` function, passing the table name and the Salesforce query.
    - Cache the resulting DataFrame in memory.
    - Load the DataFrame into the target Snowflake table using the `load_tgt_snw_table()` function with the options "overwrite" and "update".

Error Handling and Reporting

After processing all tables, the code performs error handling and reporting:

- If a maximum date value is `None`, indicating no specific date to filter source rows, an error message for the current table is appended to the `msg_to_send` list.
- Print the total number of tables to be migrated and the actual number of tables migrated.
- If the total and actual counts do not match, an error message is appended to the `msg_to_send` list.

Message Notification

At the end of the code, if there are any messages in the `msg_to_send` list, they can be sent as notifications via email. The email subject is set as `'Salesforce Leader Board - job from Databrick!'`, and the email message includes the messages from the `msg_to_send` list concatenated together.