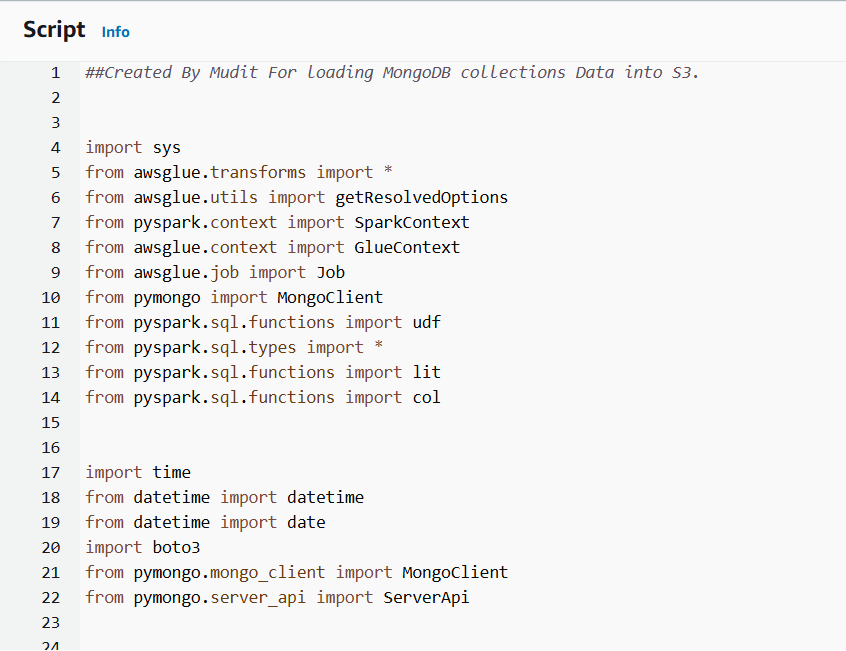
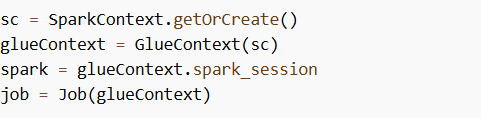
This AWS Glue job is designed to load MongoDB collections into Amazon S3 in JSON format. It connects to MongoDB using a specified URI, retrieves collections, and filters them based on predefined rules. The job fetches the latest update times from Snowflake, filters records from MongoDB based on these timestamps, and writes the filtered data to S3 in a folder structure. Additionally, it manages error handling and job status logging, ensuring that the process is well-documented, and failures are tracked in Snowflake logs.  
  
This imports necessary libraries for working with AWS Glue (a fully managed ETL service), PySpark (a Python API for Apache Spark), and MongoDB (pymongo).

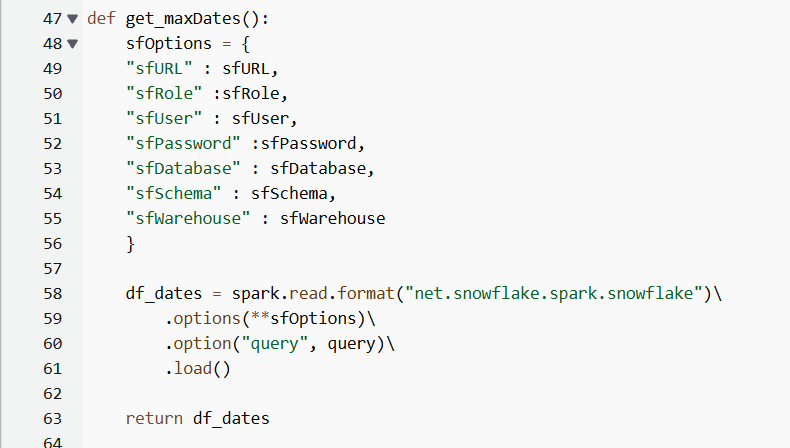
* awsglue: AWS Glue-specific modules for working with Spark.
* pymongo: For connecting to MongoDB and retrieving collections.
* boto3: AWS SDK for Python, used to interact with S3 and Glue jobs.

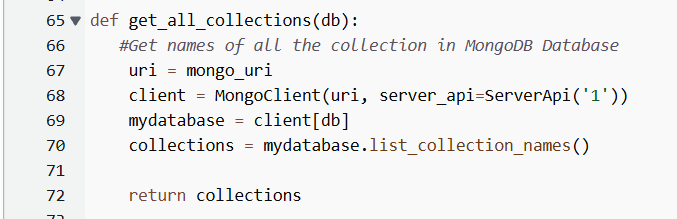


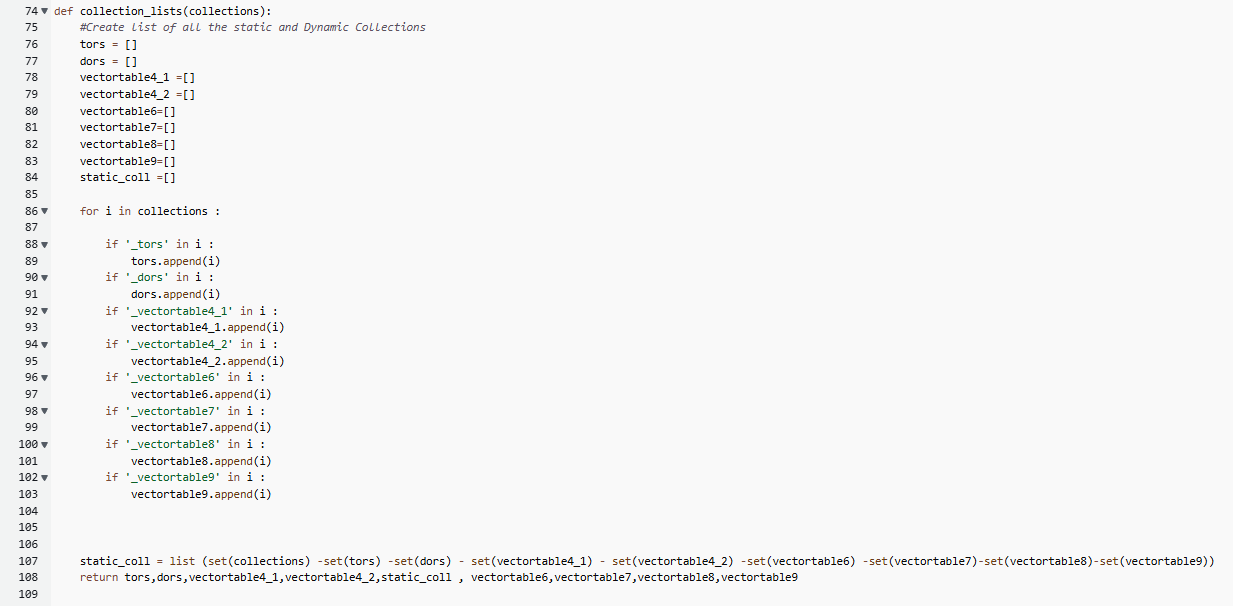
This sets up the Spark context and Glue context to work with AWS Glue's managed Spark environment. It also initializes a new Glue job.

This pulls command-line arguments passed to the Glue job. These include:

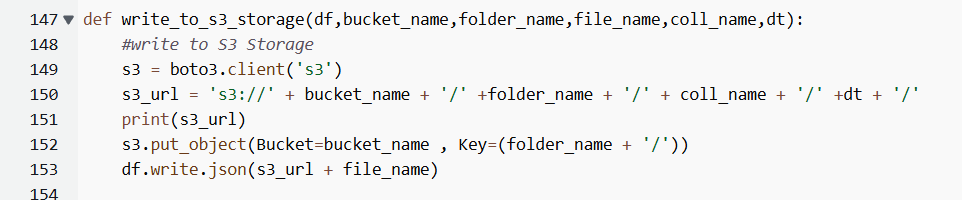
* db: The MongoDB database name.
* bucket: The S3 bucket where data will be stored.
* mongo\_uri: URI for connecting to MongoDB.
* excluded\_clients: Collections to be excluded from the operation.
* Snowflake parameters for logging job status and extracting metadata.

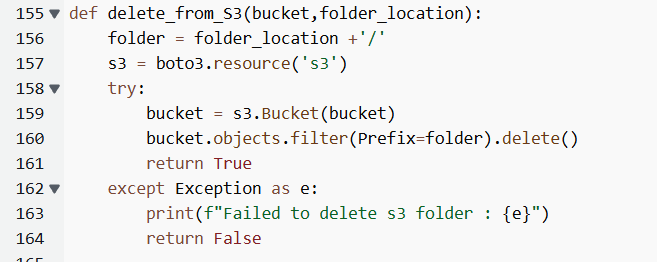
This function retrieves metadata from Snowflake using a query (to get the last update times for collections). This will help in filtering MongoDB records that were updated after the last fetch.  
This will be used to load data incrementally.

Connects to MongoDB and retrieves all collection names from the specified database. Returns a list of all the collections in the given database.

This function categorizes collections based on specific naming conventions like \_tors, \_dors, \_vectortable, etc. Anything that doesn't match these categories is considered a static collection.  
Returns the lists that contain all the types of collections.

This function loads data from a specified MongoDB collection into a Spark DataFrame using the MongoDB connector for Spark. And returns the Dataframe.

This function writes a Spark DataFrame as JSON to a specified S3 bucket in the given path.

  
Deletes the existing data from S3 to ensure that new data can be written afresh. The folder prefix is passed as a parameter to target specific paths.



save\_collections\_to\_S3(coll\_list, folder\_name, db, bucket\_name, mongo\_uri, df\_dates)

This function saves MongoDB collections to AWS S3 based on a specified filter condition and several inputs such as a list of collections, folder name, and other configurations.

Parameters:

coll\_list: A list of MongoDB collection names to be processed.

folder\_name: The folder name in S3 where the data will be stored.

db: The name of the MongoDB database.

bucket\_name: The S3 bucket name where the data will be uploaded.

mongo\_uri: The MongoDB URI used to connect to the MongoDB server.

df\_dates: A DataFrame from Snowflake containing the maximum updated timestamp for each table (used to filter records in the MongoDB collection).

Process:

Importing Required Modules:  
The function imports time, date, and additional PySpark SQL functions like expr and upper for handling date-based filtering and string manipulation.

Getting the Current Date:  
The current date is stored using date.today() to create folder structures or file names with date-based patterns. Additionally, time.time() gets the current epoch time in seconds.

Iterating Through Collections:  
The function loops through each collection in coll\_list, performing the following actions:

Filtering Data from Snowflake:  
For each MongoDB collection, the function filters the df\_dates DataFrame (from Snowflake) based on the table name. It uses a case-insensitive match to get the corresponding maximum updatedAt timestamp for the current collection using:

filtered\_df\_dates = df\_dates.filter(expr(f"upper('{i}') like concat('%', TABLE\_NAME, '%')"))

Reading the MongoDB Collection into a DataFrame:  
The collection is read from MongoDB into a Spark DataFrame using the spark.read.format("com.mongodb.spark.sql.DefaultSource"). This retrieves a maximum of 10,000 records, as specified by option("sampleSize", 10000).

Filtering Records Based on the updatedAt Field:  
The function filters the data where the updatedAt field is greater than the max\_updatedat\_value (retrieved from Snowflake). This ensures only records updated after the last Snowflake update are processed.

Saving Data to S3:  
If the filtered DataFrame contains any records (df.count() > 0), the collection data is written to an S3 bucket using the write\_to\_s3\_storage function. The data is saved in JSON format, and the file name is constructed with a timestamp.

Handling Exceptions:  
Nested try-except blocks ensure that errors encountered during reading the collections or saving to S3 are caught and printed, and the function continues processing the remaining collections.

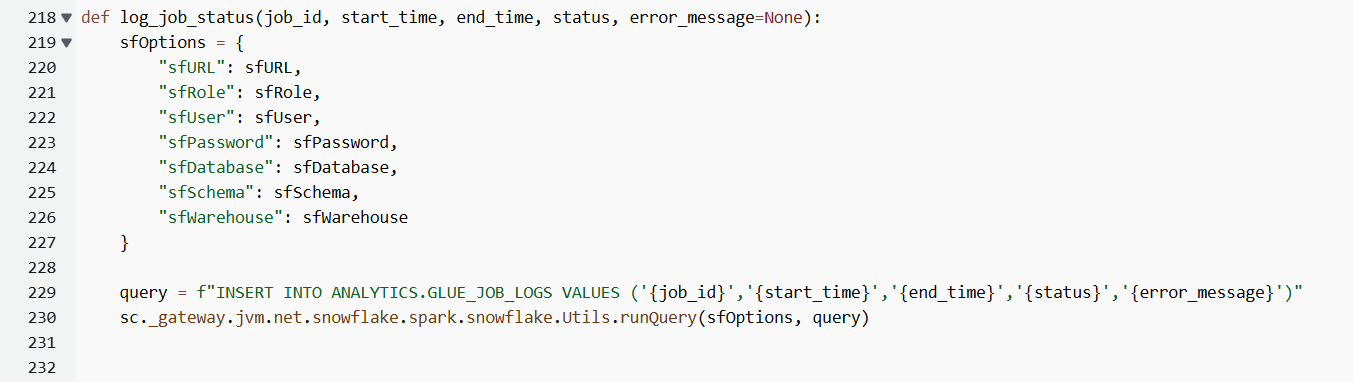
Key Points:

The data is filtered based on an updatedAt field in the MongoDB collections.

Data is only written to S3 if there are new records since the last recorded update.

Collection names are used to dynamically create file names and folder paths in S3.

The function uses a time-based filter for incremental data loading to S3, improving efficiency.



log\_job\_status(job\_id, start\_time, end\_time, status, error\_message=None)

This function records the job execution status in Snowflake by inserting relevant information into the ANALYTICS.GLUE\_JOB\_LOGS table.

Parameters:

* job\_id: A unique identifier for the job being logged.
* start\_time: The time the job started.
* end\_time: The time the job finished.
* status: The final status of the job (e.g., "success" or "failure").
* error\_message (optional): If the job fails, this field captures the error message related to the failure. It is optional and can be None if no error occurred.

Process:

Snowflake Connection Configuration (sfOptions):  
A dictionary called sfOptions is defined to store Snowflake connection details. These details include:

* sfURL: The Snowflake URL used to connect to the database.
* sfRole: The Snowflake role to be used for executing the query.
* sfUser: The Snowflake username.
* sfPassword: The Snowflake password.
* sfDatabase: The name of the Snowflake database where the logs will be inserted.
* sfSchema: The schema inside the Snowflake database where the table resides.
* sfWarehouse: The Snowflake virtual warehouse that will execute the query.

SQL Query Construction (query):  
The function constructs an SQL INSERT query as a string using Python’s f-string formatting. This query inserts the provided job details (job\_id, start\_time, end\_time, status, and error\_message) into the ANALYTICS.GLUE\_JOB\_LOGS table. If error\_message is not provided, it defaults to None in the query.

Example query:

INSERT INTO ANALYTICS.GLUE\_JOB\_LOGS   
VALUES ('job\_123', '2024-09-10 08:00:00', '2024-09-10 08:10:00', 'success', None)

Executing the Query in Snowflake:  
The sc.\_gateway.jvm.net.snowflake.spark.snowflake.Utils.runQuery() function is called to execute the query in Snowflake. The sfOptions dictionary provides the connection parameters, and the query string contains the SQL query that inserts the job log.

sc.\_gateway.jvm: This is the entry point to the Java Virtual Machine (JVM) from the PySpark context (sc). It allows you to access Snowflake’s utilities to run queries directly.

net.snowflake.spark.snowflake.Utils.runQuery(): This is a Snowflake Spark Connector utility that runs a custom SQL query in Snowflake using the connection options provided.

Key Points:

* The function is designed to log job information, including any potential errors, to Snowflake.
* The Snowflake table ANALYTICS.GLUE\_JOB\_LOGS is assumed to have fields for job\_id, start\_time, end\_time, status, and error\_message.
* The function uses PySpark to interact with the Snowflake Spark Connector and execute the query.
* If no error\_message is passed, the default value None is inserted into the table.



The main() function coordinates the process of retrieving data from MongoDB, saving the filtered collections to an S3 bucket, and logging the job status in Snowflake. The function handles errors by logging failures and ensures that the job status is updated based on the success or failure of the operations.

#### **Explanation of Each Step:**

1. **Initialize Timestamp (current\_timestamp)**: The current timestamp is captured in the format YYYY-MM-DD HH:MM:SS. This timestamp is used to track when the job starts and is later logged for status updates.
2. **Initialize job\_id**: The job\_id is initialized as None at the start of the function. This value will later be updated with the actual Glue job ID that is fetched from AWS Glue.
3. **try Block**: The core logic of the function is enclosed within a try block to handle exceptions in a controlled manner. If an error occurs, it will be caught and handled in the except block.
4. **Retrieve Glue Job ID (job\_id)**: The function connects to AWS Glue using the boto3 client and retrieves the Glue job ID by calling the get\_job\_run() method. The job name (JOB\_NAME) and run ID (JOB\_RUN\_ID) are required parameters, and the response provides the job ID, which is stored in job\_id.
5. **Retrieve All Collections from MongoDB**: The function then retrieves all collections from the specified MongoDB database (db\_name) by calling the get\_all\_collections() function. The result is a list of all collections stored in the database.
6. **Filter Excluded Collections**: A list of collections to be excluded (excluded\_clients) is processed by splitting it into individual items. The function filters out the unwanted collections from the initial list by checking for any matches between the excluded list and the collections in the database. The filtered list of collections is stored for further processing.
7. **Divide Collections into Categories**: The filtered collection list is then categorized into nine specific groups, such as tors, dors, vectortable4\_1, and others. These categories represent different subsets of collections that are handled separately later in the function.
8. **Prepare Collection Folders and List**: Two lists are prepared: one for collection folder names (coll\_folders) and another for the categorized collections (coll\_list). These lists are used to structure the data-saving process and link the appropriate collections to their respective folder names.
9. **Delete Previous Data from S3**: The function calls delete\_from\_S3() to remove any existing data from the S3 bucket for the specified folder location. This ensures that the new data will not overwrite or conflict with any previous uploads.
10. **Save Collections to S3**: Using a while loop, the function iterates through each categorized collection in coll\_list. For each collection, it calls the save\_collections\_to\_S3() function, passing the relevant collection, folder name, bucket, and MongoDB URI. This function filters and saves the data from the MongoDB collections to the S3 bucket.
11. **Log Job Status (Success)**: After processing all collections and saving them to S3, the function records the end timestamp. It logs the job status as "successful" by calling the log\_job\_status() function, passing the job ID, start time, end time, and the success status.
12. **except Block (Error Handling)**: If any error occurs during the execution of the function, the except block is triggered. The function records the end timestamp and logs the job status as "failed" along with the error message. If the job ID was not successfully retrieved, it logs "N/A" for the job ID.
13. **Completion**: Upon successful execution or failure, the function completes its process and ensures that the job status is appropriately logged.