**LoadData Function (NEW)**

Components:

A Config file which lists the bucket path

Parameters:

S3Path: The S3 bucket path where the dataset is stored

FileFormat: The format of the file (e.g., CSV, JSON, Parquet)

Output:

DataFrame

Code Body:

1. Initialize DataFrame Structure:

- Try:

- Create an empty DataFrame or initialize based on the expected schema

- Except Exception as e:

- Log error: "Error initializing DataFrame structure: {e}"

- Raise an appropriate exception or handle the error gracefully

2. Load Data from S3:

- Try:

- Connect to the S3 bucket using appropriate credentials

- Retrieve the file from the specified S3Path

- Except boto3.exceptions.S3UploadFailedError as e:

- Log error: "Error connecting to S3 or retrieving file: {e}"

- Raise an appropriate exception or handle the error gracefully

3. Read the file based on the specified FileFormat:

- Try:

- If FileFormat is CSV:

- Read the file using pandas' read\_csv method

- Else If FileFormat is JSON:

- Read the file using pandas' read\_json method

- Else If FileFormat is Parquet:

- Read the file using pandas' read\_parquet method

- Else:

- Log error: "Unsupported file format: {FileFormat}"

- Raise an appropriate exception or handle the error gracefully

- Except Exception as e:

- Log error: "Error reading file: {e}"

- Raise an appropriate exception or handle the error gracefully

4. Return DataFrame:

- Return the prepared DataFrame for validation functions

**Completeness check**

Completeness Check Function

Parameters:

DataFrame: The dataset to be validated

RequiredFields: A list of fields that must be present and non-null in each record

Output:

Flattened Results file in S3

Code Body:

1. Initialize CompletenessResults structure:

- Try:

- Create a structure to hold the results of the completeness check for each required field.

- CompletenessResults = []

- Read the RequiredFields from the configuration file.

- Except Exception as e:

- Log error: "Error initializing CompletenessResults structure or retrieving fields: {e}"

- Raise an appropriate exception or handle the error gracefully

2. Loop through RequiredFields:

- For each field in RequiredFields:

- Try:

a. Check for null values in the field:

- NullCount = DataFrame[field].isnull().sum()

b. Record the number of missing values:

- TotalRecords = len(DataFrame)

c. Calculate completeness percentage for the field:

- CompletenessPercentage = ((TotalRecords - NullCount) / TotalRecords) \* 100

d. Append the results to CompletenessResults:

- CompletenessResults.append({

"Field": field,

"TotalRecords": TotalRecords,

"NullCount": NullCount,

"CompletenessPercentage": CompletenessPercentage

})

- Except KeyError:

- Log error: "Field {field} not found in DataFrame"

- Continue to the next field

- Except ZeroDivisionError:

- Log error: "Division by zero error for field {field}"

- CompletenessResults.append({

"Field": field,

"TotalRecords": 0,

"NullCount": 0,

"CompletenessPercentage": 0

})

- Continue to the next field

- Except Exception as e:

- Log error: "Error processing field {field}: {e}"

- Continue to the next field

3. Flatten & store results in S3:

- Try:

a. Convert CompletenessResults to a DataFrame:

- ResultsDataFrame = pd.DataFrame(CompletenessResults)

b. Get current datetime for partitioning:

- CurrentDateTime = getCurrentDateTime()

c. Define S3 output path with datetime partition:

- S3OutputPath = f"s3://output/completeness\_results/{CurrentDateTime}/results.csv"

d. Save the results DataFrame to the S3 output path:

- ResultsDataFrame.to\_csv('/tmp/results.csv', index=False)

- UploadFileToS3('/tmp/results.csv', S3OutputPath)

- RemoveLocalFile('/tmp/results.csv') # Optional but recommended to clean up

- Except Exception as e:

- Log error: "Error flattening and storing results in S3: {e}"

- Raise an appropriate exception or handle the error gracefully

4. Return:

- Return a success message or status indicating that the results have been saved to S3.

**Uniqueness Check**

UniquenessCheck Function Pseudocode

Parameters:

DataFrame: The dataset to be validated

Config: The configuration object loaded from the YAML file

Output:

Flattened Results file in S3

Code Body:

1. Load Configuration:

- Read the uniqueness check configuration from the Config object.

2. Initialize UniquenessResults structure:

- Create a structure to hold the results of the uniqueness check for each primary key field.

- UniquenessResults = []

3. Identify Primary Key Fields:

- If 'composite\_key\_fields' exists in Config:

- KeyFields = Config['uniqueness\_check']['composite\_key\_fields']

- Else:

- KeyFields = Config['uniqueness\_check']['primary\_key\_fields']

4. Identify Duplicates:

- Try:

- If KeyFields is a list (composite key):

- DuplicateRecords = DataFrame[DataFrame.duplicated(subset=KeyFields)]

- Else (single key):

- DuplicateRecords = DataFrame[DataFrame.duplicated(subset=[KeyFields])]

- Catch and log any errors during duplicate identification.

5. Calculate Duplicate Statistics:

- Try:

- For each field in KeyFields:

a. TotalRecords = len(DataFrame)

b. DuplicateCount = len(DuplicateRecords)

c. DuplicatePercentage = (DuplicateCount / TotalRecords) \* 100

d. Append the results to UniquenessResults:

- UniquenessResults.append({

"Field": field,

"TotalRecords": TotalRecords,

"DuplicateRecords": DuplicateCount,

"DuplicatePercentage": DuplicatePercentage

})

- Catch and log any errors during calculation of duplicate statistics.

6. Flatten & store results in S3:

- Try:

a. Convert UniquenessResults to a DataFrame.

b. Get current datetime for partitioning:

- CurrentDateTime = getCurrentDateTime()

c. Define S3 output path with datetime partition:

- S3OutputPath = f"s3://output/uniqueness\_results/{CurrentDateTime}/results.csv"

d. Save the results DataFrame to the S3 output path:

- Save DataFrame to CSV format.

- Upload CSV file to S3OutputPath.

- Catch and log any errors during the flattening and storing process.

7. Return:

- Try:

- Return a success message or status indicating that the results have been saved to S3.

- Catch and log any errors during the return process.or status indicating that the results have been saved to S3.

**Validity**

ValidityCheck Function Pseudocode

Parameters:

DataFrame: The dataset to be validated

Config: The configuration object loaded from the YAML file

Output:

Flattened Results file in S3

Code Body:

1. Load Configuration:

- Read the validity check configuration from the Config object.

- Try:

- ValidationRules = Config['validity\_check']['rules']

- Catch and log any errors during configuration loading.

2. Initialize ValidityResults structure:

- Create a structure to hold the results of the validity check for each field.

- ValidityResults = []

3. Loop through ValidationRules:

- Try:

- For each field and its rule in ValidationRules:

a. Apply the validation rule for each field:

- InvalidRecords = DataFrame[~DataFrame[field].apply(rule)]

b. Record invalid entries:

- InvalidCount = len(InvalidRecords)

- TotalRecords = len(DataFrame)

- InvalidPercentage = (InvalidCount / TotalRecords) \* 100

- InvalidExamples = InvalidRecords[field].tolist()

c. Append the results to ValidityResults:

- ValidityResults.append({

"Field": field,

"TotalRecords": TotalRecords,

"InvalidRecords": InvalidCount,

"InvalidPercentage": InvalidPercentage,

"InvalidExamples": InvalidExamples[:5] # Limiting examples for readability

})

- Catch and log any errors during the validation process.

4. Flatten & store results in S3:

- Try:

a. Convert ValidityResults to a DataFrame.

b. Get current datetime for partitioning:

- CurrentDateTime = getCurrentDateTime()

c. Define S3 output path with datetime partition:

- S3OutputPath = f"s3://output/validity\_results/{CurrentDateTime}/results.csv"

d. Save the results DataFrame to the S3 output path:

- Save DataFrame to CSV format.

- Upload CSV file to S3OutputPath.

- Catch and log any errors during the flattening and storing process.

5. Return:

- Try:

- Return a success message or status indicating that the results have been saved to S3.

- Catch and log any errors during the return process.

**Explanation of Exception Handling**

1. **Load Configuration**:
   * Try-catch block ensures that any errors during the loading of configuration from the YAML file are caught and logged, preventing the process from failing silently.
2. **Initialize ValidityResults Structure**:
   * This step is straightforward and does not typically require exception handling.
3. **Loop through ValidationRules**:
   * A try-catch block is used to handle any errors that occur during the application of validation rules. This includes checking if each record meets the rule criteria and recording invalid entries.
   * If an error occurs, it is logged, and the process continues for the next field, ensuring robustness.
4. **Flatten & Store Results in S3**:
   * Another try-catch block is used here to handle any errors during the conversion of results to a DataFrame, partitioning by datetime, and saving to S3.
   * This ensures that any issues during file saving are logged, and the process does not fail without notification.
5. **Return**:
   * The final try-catch block ensures that any errors during the return process are caught and logged, ensuring that the function always returns a status, even if an error occurs.

By grouping related steps into try-catch blocks, we minimize the number of exception handling points and ensure that each critical section is protected against potential errors. This approach makes the pseudocode production-grade and robust for real-life projects.

===========================================================