```python
from pyspark.sql.functions import col, count, lower
from typing import Dict, Any
from pyspark.sql import DataFrame
from great_expectations.dataset import SparkDFDataset
from great_expectations.core.expectation_validation_result import
ExpectationValidationResult
from abc import abstractmethod
from functools import reduce
import boto3
import os
import sys
import inspect
import logging
import json


logging.basicConfig(
    level=logging.INFO,
    format='%(asctime)s [%(levelname)s]: %(message)s',
    datefmt='%Y-%m-%d %H:%M:%S',
    handlers=[logging.StreamHandler()]
)

logger = logging.getLogger(__name__)


class DQChecker():

    def __init__(self, spdf, dq_json, obj, pipeline_name) -> None:
        self.totalFailsDict = {}
        self.obj = obj
        self.pipeline_name = pipeline_name
        self.spdf = spdf
        self.dq_json = dq_json
        self.function_name = self.__class__.__name__
        self.columns_to_remove_rows = dq_json.get("ActionRemoveRows")
        self.columns_to_stop_load = dq_json.get("ActionStopLoad")

    @abstractmethod
    def run(self):
        """Run the check on the spdf."""
        raise NotImplementedError

    def checkNotNeeded(self, columns: dict) -> bool:
        """Return True if check is not specified in dq_json, else False."""
        if columns is None:
            return True
        return False

    def checkPassed(self, passed: bool, column: str = None) -> bool:
        """Print pass- or fail-message and return True if check was a success, else
print how many rows failed and in perent."""
        if not column:
            if passed:
                passMessage = f"Check PASSED!\n"
                logger.info(passMessage)
                return True
            else:
                failedMessage = f"Check FAILED\n"
                logger.warning(failedMessage)
                return False
        else:
            if passed:
                passMessage = f"Column {column} PASSED!\n"
                logger.info(passMessage)
                return True
            else:
                failedMessage = f"Column {column} FAILED\n"
                logger.warning(failedMessage)
                return False


class DQActions():

    @staticmethod
    def remove_failing_rows_TestNotNull(checker, column):
        num_removed_rows = checker.spdf.filter(col(column).isNull()).count()
        checker.spdf = checker.spdf.filter(col(column).isNotNull())
        return checker, num_removed_rows

    @staticmethod
    def remove_failing_rows_TestUnique(checker, res, column):
        num_removed_rows = res.result.get('unexpected_count') -
checker.spdf.groupby(
            col(column)).count().filter(col("count") >
1).agg(count(column)).collect()[0][0]
        checker.spdf = checker.spdf.drop_duplicates([column])
        return checker, num_removed_rows

    @staticmethod
    def remove_failing_rows_TestNotNullIfCondition(checker, column):
        num_removed_rows = 0
        rows_to_remove_df = checker.spdf.filter("1=0")
        cond_cols =
checker.dq_json.get(checker.function_name).get(column).get("cond_col")
        cond_types =
checker.dq_json.get(checker.function_name).get(column).get("cond_type")
        cond_values =
checker.dq_json.get(checker.function_name).get(column).get("value")

        # If the values are not lists, convert them to lists
        if not isinstance(cond_cols, list):
            cond_cols = [cond_cols]
        if not isinstance(cond_types, list):
            cond_types = [cond_types]
        if isinstance(cond_cols, list) and not isinstance(cond_values[0], list):
            cond_values = [cond_values]

        for cond_col, cond_type, cond_value in zip(cond_cols, cond_types,
cond_values):
            cond_type = cond_type.lower()
            for value in cond_value:
                if cond_type == "string":
                    rows_to_remove = checker.spdf.filter(
                        col(cond_col) == value).filter(col(column).isNull())
                elif cond_type == "string-not":
                    rows_to_remove = checker.spdf.filter((col(cond_col) != value) |
(
                        col(cond_col).isNull())).filter(col(column).isNull())
                rows_to_remove_df = rows_to_remove_df.union(rows_to_remove)
                num_removed_rows += rows_to_remove.count()
        checker.spdf =
checker.spdf.select('*').exceptAll(rows_to_remove_df.select('*'))
        return checker, num_removed_rows


    @staticmethod
    def remove_failing_rows_TestCompoundColumnsUnique(checker, res):
        column_list = list(res.result["unexpected_list"][0].keys())
        num_removed_rows = res.result.get('unexpected_count')/2
        checker.spdf = checker.spdf.drop_duplicates(column_list)
        return checker, num_removed_rows

    @staticmethod
    def remove_failing_rows_TestRowHasOneOfColumns(checker):
        columns = checker.dq_json.get("TestRowHasOneOfColumns")
        rows_to_remove_df = checker.testLogic(columns)
        checker.spdf =
checker.spdf.select('*').exceptAll(rows_to_remove_df.select('*'))
        return
    @staticmethod
    def remove_failing_rows_Others(checker, res, column):
        num_removed_rows = res.result.get('unexpected_count')
        rows_to_remove = list(set(res.result["unexpected_list"]))
        checker.spdf = checker.spdf.filter(~col(column).isin(rows_to_remove))
        return checker, num_removed_rows

    @staticmethod
    def remove_failing_rows(checker, res, column):
        """Remove rows that cause the test for the specified column to fail."""
        logger.info(f"Removing rows for column {column}..")
        if checker.function_name == "TestNotNull":
            checker, num_removed_rows =
DQActions.remove_failing_rows_TestNotNull(checker, column)

        elif checker.function_name == "TestNotNullIfCondition":
            checker, num_removed_rows =
DQActions.remove_failing_rows_TestNotNullIfCondition(
                checker, column)

        elif checker.function_name == "TestUnique":
            checker, num_removed_rows = DQActions.remove_failing_rows_TestUnique(
                checker, res, column)

        elif checker.function_name == "TestCompoundColumnsUnique":
            checker, num_removed_rows =
DQActions.remove_failing_rows_TestCompoundColumnsUnique(
                checker, res)
        else:
            checker, num_removed_rows =
DQActions.remove_failing_rows_Others(checker, res, column)

        checker.ge_df = SparkDFDataset(checker.spdf)

        logger.info(f"{num_removed_rows} rows removed in
{checker.function_name}.\n")
        logger.info(f"{checker.spdf.count()} rows left.")

        return checker, num_removed_rows

    @staticmethod
    def logFailedColumns(checker: DQChecker, column: str, res=None) -> Dict[str,
Any]:
        """Ensuring we keep track of checks and columns that fails"""
        if checker.function_name not in checker.totalFailsDict:
            checker.totalFailsDict[checker.function_name] = {}
        num_failing_rows = DQActions.getFailedRows(checker, column, res)
        logger.info(
            f"Number of rows that failed {checker.function_name} for column
{column}: {num_failing_rows}")
        if num_failing_rows > 0:
            checker.totalFailsDict[checker.function_name][column] =
num_failing_rows
        return checker.totalFailsDict

    @staticmethod
    def getFailedRows(checker: DQChecker, column: str, res=None):
        """Return the number of rows that failed the check"""
        if checker.__class__.__bases__[0].__name__ == "DQCheckGE":
            if checker.function_name == "TestUnique":
                # res.result["unexpected_count"] is the number of rows that failed
the check but includes all duplicates. We need to subtract the number of unique
duplicates.
                # This means that if we have "value1" in 3 rows, we will only count
2 rows as failing the check.
                num_failing_rows = res.result["unexpected_count"] -
checker.spdf.groupby(
                    col(column)).count().filter(col("count") >
1).agg(count(column)).collect()[0][0]
            elif checker.function_name == "TestCompoundColumnsUnique":
                num_failing_rows = res.result.get("unexpected_count")/2
            else:
                num_failing_rows = res.result["unexpected_count"]
        elif checker.__class__.__bases__[0].__name__ == "DQChecker":
            num_failing_rows = checker.num_failing_rows
        return num_failing_rows

    @staticmethod
    def stopLoad(checker):
        failingTest = {k: v for k, v in checker.totalFailsDict.items() if k in [
            checker.function_name]}
        send_email_on_data_quality_failure(
            checker.obj, checker.pipeline_name, failingTest,
checker.dq_json.get("ActionSendEmail"))
        logger.critical("Exiting script - Not loading data")
        sys.exit(1)


class DQCheckGE(DQChecker):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.ge_df = SparkDFDataset(self.spdf)
        self.expectation_name = self.get_expectation_name()
        self.result_format = "COMPLETE"

    def get_expectation(self, column):
        """Return the appropriate Great Expectations expectation for the check."""
        expectation_func = getattr(self.ge_df, self.expectation_name)
        return expectation_func(**self.get_expectation_kwargs(column),
result_format=self.result_format)

    @abstractmethod
    def get_expectation_name(self):
        """Return the name of the Great Expectations expectation for the check."""
        raise NotImplementedError

    @abstractmethod
    def get_expectation_kwargs(self, column):
        """Return a dictionary of keyword arguments for the Great Expectations
expectation."""
        raise NotImplementedError


class RunnerCheck():
    def __init__(self, checker) -> None:
        self.checker: DQChecker = checker
        self.num_removed_rows: int = 0

    def run(self, return_removed_rows: bool = False):
        """Run check on specified columns if needed and handle if check fails."""
        columns = self.checker.dq_json.get(self.checker.function_name)
        if self.checker.checkNotNeeded(columns):

            return self.checker.spdf, self.num_removed_rows
        logger.info(f"Starting check {self.checker.function_name}..")
        self.checker.check(columns)
        if return_removed_rows:
            return self.checker.spdf, self.num_removed_rows
        else:
            return self.checker.spdf


class RunnerCheckSpark(RunnerCheck):
    def __init__(self, checker, spark) -> None:
        super().__init__(checker)
        self.spark = spark

    def run(self, return_removed_rows: bool = False):
        """Run check on specified columns if needed and handle if check fails."""
        columns = self.checker.dq_json.get(self.checker.function_name)
        if self.checker.checkNotNeeded(columns):
            return self.checker.spdf, self.num_removed_rows
        logger.info(f"Starting check {self.checker.function_name}..")
        if return_removed_rows:
            return self.checker.check(columns, self.spark), self.num_removed_rows
        else:
            return self.checker.check(columns, self.spark)


class RunnerCheckGE():
    def __init__(self, checker: bool = None) -> None:
        self.checker: DQCheckGE = checker
        self.num_removed_rows: int = 0

    def run(self, return_removed_rows: bool = False):
        """Run check on specified columns if needed and handle if check fails."""
        columns = self.checker.dq_json.get(self.checker.function_name)

        if self.checker.checkNotNeeded(columns):
            return self.checker.spdf, self.num_removed_rows
        res_dict = {}
        logger.info(f"Starting check {self.checker.function_name}..")
        for column in columns:
            # Run check
            res = self.checker.get_expectation(column)
            res_dict[column] = res
            if not self.checkPassed(res, column):
                # Log failed check and columns
                self.checker.totalFailsDict = DQActions.logFailedColumns(
                    self.checker, column=column, res=res)
                if self.checker.columns_to_stop_load and column in
self.checker.columns_to_stop_load:
                    DQActions.stopLoad(checker=self.checker)
        for column in columns:
            if self.checker.columns_to_remove_rows and column in
self.checker.columns_to_remove_rows and res_dict[column].success == False:
                self.checker, num_removed_rows = DQActions.remove_failing_rows(
                    checker=self.checker, res=res_dict[column], column=column)
                self.num_removed_rows += num_removed_rows
        if return_removed_rows:
            return self.checker.spdf, self.num_removed_rows
        return self.checker.spdf

    def checkPassed(self, res: dict, column: str):
        """Print pass- or fail-message and return True if check was a success, else
print how many rows failed and in perent."""
        if res.success:
```

```python
            passMessage = f"Column {column} PASSED!\n"
            logger.info(passMessage)
            return True
        else:
            if self.checker.function_name == "TestCompoundColumnsUnique":
                failedMessage = f"{int(res.result.get('unexpected_count'))/2} of
{res.result.get('element_count')} items or
{res.result.get('unexpected_percent'):.2f}% Column {column} FAILED\n"
            else:
                failedMessage = f"{res.result.get('unexpected_count')} of
{res.result.get('element_count')} items or
{res.result.get('unexpected_percent'):.2f}% Column {column} FAILED\n"
            logger.warning(failedMessage)
            return False


class TestUnique(DQCheckGE):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def get_expectation_name(self):
        return "expect_column_values_to_be_unique"

    def get_expectation_kwargs(self, column):
        return {"column": column}


class TestRange(DQCheckGE):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def get_expectation_name(self):
        return "expect_column_values_to_be_between"

    def get_expectation_kwargs(self, column):
        return {
            "column": column,
            "min_value":
self.dq_json.get(self.function_name).get(column).get("min_value"),
            "max_value":
self.dq_json.get(self.function_name).get(column).get("max_value"),
            "mostly":
self.dq_json.get(self.function_name).get(column).get("mostly")
        }


class TestSet(DQCheckGE):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def get_expectation_name(self):
        return "expect_column_values_to_be_in_set"

    def get_expectation_kwargs(self, column):
        return {
            "column": column,
            "value_set": self.dq_json.get(self.function_name).get(column)
        }


class TestNotNull(DQCheckGE):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def get_expectation_name(self):
        return "expect_column_values_to_not_be_null"

    def get_expectation_kwargs(self, column):
        return {
            "column": column}


class TestDateformat(DQCheckGE):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def get_expectation_name(self):
        return "expect_column_values_to_match_strftime_format"

    def get_expectation_kwargs(self, column):
        return {
            "column": column,
            "strftime_format": self.dq_json.get(self.function_name).get(column)
        }


class TestNotNullIfCondition(DQCheckGE):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.cond_col = None
        self.cond_type = None
        self.cond_value = None
        self.dict_column_cond_values = {}
    def get_expectation_name(self):
        return "expect_column_values_to_not_be_null"

    def get_expectation_kwargs(self, column):
        cond_cols =
self.dq_json.get(self.function_name).get(column).get("cond_col")
        cond_types =
self.dq_json.get(self.function_name).get(column).get("cond_type")
        cond_values = self.dq_json.get(self.function_name).get(column).get("value")

        # If the values are not lists, convert them to lists
        if not isinstance(cond_cols, list):
            cond_cols = [cond_cols]
        if not isinstance(cond_types, list):
            cond_types = [cond_types]
        # Ensure that cond_values is always a nested list when dealing with
multiple conditions
        if isinstance(cond_cols, list) and not isinstance(cond_values[0], list):
            cond_values = [cond_values]

        # Assert that the lengths of the lists are the same
        assert len(cond_cols) == len(cond_types) == len(cond_values), "The lengths
of the condition lists are not the same"

        sub_spdf = self.spdf
        for cond_col, cond_type, cond_value in zip(cond_cols, cond_types,
cond_values):
            cond_type = cond_type.lower()
            if cond_type == "boolean" and len(cond_value) == 1:
                cond_value = [eval(cond_value[0].capitalize())]
            if cond_type == "string":
                sub_spdf = sub_spdf.filter(col(cond_col).isin(cond_value))
            elif cond_type == "string-not":
                sub_spdf = sub_spdf.filter(~col(cond_col).isin(cond_value) |
(col(cond_col).isNull()))
            if sub_spdf.first() == None:
                logger.info(
                    f"Couldn't find any rows where column: '{cond_col}' has value:
{cond_value}")
                return None
        return sub_spdf

    def get_expectation(self, column):
        sub_spdf = self.get_expectation_kwargs(column)
        if sub_spdf is None:
            return ExpectationValidationResult(
                success=True,
```

```python
                result={
                    "unexpected_count": 0,
                    "element_count": 0,
                    "unexpected_percent": 0.0,
                }
            )

        ge_sub_df = SparkDFDataset(sub_spdf)
        expectation_func = getattr(ge_sub_df, self.expectation_name)
        return expectation_func(column=column, result_format="COMPLETE")


class TestCompoundColumnsUnique(DQCheckGE):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)

    def get_expectation_name(self):
        return "expect_compound_columns_to_be_unique"

    def get_expectation_kwargs(self, column):
        return {"column_list": self.dq_json.get(self.function_name).get(column)}


class TestOneRowHasValue(DQChecker):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.function_name = self.__class__.__name__
        self.num_failing_rows = 0

    def check(self, columns: dict):
        for column in columns:
            for value in columns.get(column):
                column_has_value = self.testLogic(column, value)
                self.checkPassed(column_has_value, column)
                self.totalFailsDict = DQActions.logFailedColumns(self, column)
        return self.spdf

    def testLogic(self, column, value):
        return self.spdf.filter(col(column) == value).count() > 0


class TestRowHasOneOfColumns(DQChecker):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.function_name = self.__class__.__name__

    def check(self, columns: dict):
        # Filter DataFrame to only include rows where none of the specified columns
have a value
        filtered_df = self.testLogic(columns)
        self.num_failing_rows = filtered_df.count()
        self.checkPassed(filtered_df.count() > 0)
        # Logging the colunmns and the number of rows that failed the test
        # as the generic logFailedColumns takes a single column and does not work
for this test
        self.totalFailsDict[self.function_name] = {}
        self.totalFailsDict[self.function_name][(", ").join(
            self.dq_json.get(self.function_name))] = self.num_failing_rows
        DQActions.remove_failing_rows_TestRowHasOneOfColumns(self)
        return self.spdf

    def testLogic(self, columns: dict) -> DataFrame:
        return self.spdf.where(
            ~(reduce(lambda x, y: x | y, [col(c).isNotNull() for c in columns])))


class TestExistsInDifferentTable(DQChecker):
    def __init__(self, *args, **kwargs):
        super().__init__(*args, **kwargs)
        self.function_name = self.__class__.__name__

    @staticmethod
    def load_parquet(path, spark):
        try:
            return spark.read.option("mergeSchema", "true").parquet(path)
        except Exception as e:
            raise e

    def join_cols(self, reference_spdf_column, column, reference_column):
        # Join the two dataframes on the columns of interest using leftsemi join
        col_to_check = self.spdf.select(column).dropDuplicates()
        joined = col_to_check.join(reference_spdf_column, col(column)
                                   == col(reference_column), "leftsemi")

        # Perform a left anti join to find the rows in colA that are not present in
colB
        not_in_colB = col_to_check.join(reference_spdf_column, col(
            column) == col(reference_column), "leftanti")
        self.num_failing_rows = not_in_colB.dropna().count()

        return col_to_check.dropna(), joined.dropna(), not_in_colB.dropna()

    @staticmethod
    def evaluate_join(joined, col_to_check, column, not_in_colB):
```

```python
        # If the number of rows in the joined dataframe is the same as the number
of rows in colA, then all values in colA exist in colB
        if joined.count() == col_to_check.count():
            logger.info(f"All values in column: {column} exist in reference table")
        else:
            logger.warning(f"Not all values in column: {column} exist in reference
table")

        # Show the elements in colA that are not present in colB
        if not_in_colB.count() > 0:
            in_colA = col_to_check.count()
            logger.warning(f"{not_in_colB.count()} out of {in_colA}
({not_in_colB.count()/in_colA*100:.1f}%) was not found in reference column. The
following elements in column: {column} are not present in reference table:")
            not_in_colB.show()

    def check(self, columns: dict, spark):
        # Get sparkdataframe for reference table (Table B) from s3.
        # Reference table must be stored as parquet file.
        logger.debug(f"os.environ.get('STAGE'): {os.environ.get('STAGE')}")
        for column in columns:
            if os.environ.get('STAGE') == 'dev':
                reference_table_path = self.dq_json.get(self.function_name).get(
                    column).get("reference_table_path_dev")
            elif os.environ.get('STAGE') == 'test':
                reference_table_path = self.dq_json.get(self.function_name).get(
                    column).get("reference_table_path_test")
            elif os.environ.get('STAGE') == 'val':
                reference_table_path = self.dq_json.get(self.function_name).get(
                    column).get("reference_table_path_val")
            else:
                reference_table_path = self.dq_json.get(self.function_name).get(
                    column).get("reference_table_path_prod")
            logger.info(f"reference_table_path: {reference_table_path}")

            reference_column = self.dq_json.get(self.function_name).get(
                column).get("reference_column_name")
            reference_spdf_column = self.load_parquet(
                reference_table_path, spark).select(reference_column)
            col_to_check, joined, not_in_colB = self.join_cols(
                reference_spdf_column, column, reference_column)
            self.evaluate_join(joined, col_to_check, column, not_in_colB)
            self.totalFailsDict = DQActions.logFailedColumns(self, column)

        return self.spdf
```

```python
def create_email_message(fails_dict, obj):
    # Create list of failed checks and columns
    failed_check_and_columns_list = [
        f"{key} failed for {value}" for key, value in fails_dict.items()]

    # Create message string
    newline = "\t\n"
    message = inspect.cleandoc(f"""
        Data Quality checks for the table: {obj} failed!
        These Data Quality checks failed for following columns:
        {f"{newline}".join(failed_check_and_columns_list)}
        Please ensure that this is looked into"""
                               )
    return message


def send_email(receivers, message, pipeline_name):
    # Create SNS client and get account ID
    client = boto3.client("sns")
    account_id = boto3.client("sts").get_caller_identity()["Account"]
    arns = []
    # Send email to each recipient
    for receiver in receivers:
        arn =
f"arn:aws:sns:{os.environ.get('AWS_DEFAULT_REGION')}:{account_id}:{receiver}-
{pipeline_name}"
        arns.append(arn)
        logger.info(f"Sending email to arn: {arn}")
        try:
            response = client.publish(
                TopicArn=arn,
                Message=message,
                Subject=f"DQ checks failing in {pipeline_name}"
            )
        except Exception as e:
            logger.error(f"Error sending email to {arn}: {e}")

    return arns


def get_receivers(fails_dict, columns_send_mail):
    # Make list of all columns that fail
    all_failed_columns = list(set([col for cols in fails_dict.values() for col in
cols]))

    # Make a list of all columns that requires an email to be sent if they fail
    columns_that_require_email_if_fails = [column for column in columns_send_mail]
    # The only lists that needs to be considered in email sending are the emails
that are both
    # required to act on and has actually failed
    failed_columns_to_send_email = list(
        set(all_failed_columns).intersection(columns_that_require_email_if_fails))

    # Creating a list of receivers of the email based on the responsibility inputed
in
    # columns_send_mail and if the columns actually failed. Flattening the list.
    nested_list_of_receivers = [columns_send_mail.get(
        x) for x in columns_send_mail if x in failed_columns_to_send_email]
    receivers = list(set([topic for topics in nested_list_of_receivers for topic in
topics]))

    return receivers


def send_email_on_data_quality_failure(obj: str, pipeline_name: str, fails_dict:
dict, columns_send_mail: dict):
    # Get list of receivers
    receivers = get_receivers(fails_dict, columns_send_mail)
    # Create email message
    message = create_email_message(fails_dict, obj)
    # Send email
    send_email(receivers, message, pipeline_name)


def send_teams_notification_to_sns(topic_name: str, fails_dict: dict, obj: str):
    """Send notification to SNS topic."""
    topic_arn = generate_sns_arn(topic_name)
    json_message = generate_sns_message(fails_dict, obj)
    sns_client = boto3.client("sns")
    response = sns_client.publish(
        TopicArn=topic_arn,
        Message=json.dumps(json_message),
        Subject="Data Quality Check Failed"
    )
    if response["ResponseMetadata"]["HTTPStatusCode"] != 200:
        logger.error("SNS notification failed")
        raise Exception("SNS notification failed")
    logger.info(f"Notification sent succesfully to SNS topic: {topic_name}")
    logger.info(f"Notification message: {json.dumps(json_message)}")
    return response


def generate_sns_arn(topic_name: str):
```

```python
    """Generate SNS topic ARN."""
    logger.info("Generating SNS topic ARN")
    account_id = boto3.client("sts").get_caller_identity()["Account"]
    arn =
f"arn:aws:sns:{os.environ.get('AWS_DEFAULT_REGION')}:{account_id}:{topic_name}"
    logger.info(f"SNS topic ARN generated: {arn}")
    return arn


def generate_sns_message(fails_dict: dict, obj: str):
    """Generate SNS message."""
    logger.info("Generating SNS message")
    sns_message_dict = {}
    sns_message_dict["DQM"] = True
    sns_message_dict["obj"] = obj
    sns_message_dict["fails_dict"] = fails_dict
    return sns_message_dict


def run_checks(spdf, dq_json: dict, obj: str = None, pipeline_name: str = None,
return_num_removed_rows: bool = False, spark=None):
    """Run all checks on the specified dataframe."""
    checks = [
        TestUnique,
        TestRange,
        TestSet,
        TestNotNull,
        TestDateformat,
        TestNotNullIfCondition,
        TestOneRowHasValue,
        TestRowHasOneOfColumns,
        TestCompoundColumnsUnique,
        TestExistsInDifferentTable
    ]

    checker = DQChecker(spdf, dq_json, obj, pipeline_name)
    total_num_removed_rows: int = 0

    for check_class in checks:
        check = check_class(checker.spdf, dq_json, obj, pipeline_name)

        if check.function_name == "TestExistsInDifferentTable":
            checkRunner = RunnerCheckSpark(check, spark=spark)
        elif check.__class__.__bases__[0].__name__ == "DQCheckGE":
            checkRunner = RunnerCheckGE(check)
        elif check.__class__.__bases__[0].__name__ == "DQChecker":
            checkRunner = RunnerCheck(check)
        logger.debug(
            f"Running check: {check.__class__.__name__}, Runner:
{checkRunner.__class__.__name__}")
        if return_num_removed_rows:
            checker.spdf, num_removed_rows = checkRunner.run(
                return_removed_rows=return_num_removed_rows)
            total_num_removed_rows += num_removed_rows

        else:
            checker.spdf = checkRunner.run()
        checker.totalFailsDict.update(check.totalFailsDict)

    if dq_json.get("ActionSendEmail"):
        send_email_on_data_quality_failure(
            obj, pipeline_name, fails_dict=checker.totalFailsDict,
columns_send_mail=dq_json.get("ActionSendEmail"))
    if dq_json.get("ActionSendTeamsNotification"):
        columns_to_send_notification_on_failure = dq_json.get(
            "ActionSendTeamsNotification").get("columns")
        if columns_to_send_notification_on_failure and checker.totalFailsDict !=
{}:
            filtered_totalFailsDict = {test: {column_to_include:
result[column_to_include]
                                              for column_to_include in
columns_to_send_notification_on_failure if column_to_include in result} for test,
result in checker.totalFailsDict.items()}
            if filtered_totalFailsDict != {}:
                logger.info(
                    f"Found columns that failed which requires a notification to be
sent: {filtered_totalFailsDict}")
                topics =
dq_json.get("ActionSendTeamsNotification").get("sns_topics")
                for topic in topics:
                    send_teams_notification_to_sns(topic, filtered_totalFailsDict,
obj)

    if return_num_removed_rows:
        return checker.spdf, total_num_removed_rows
    else:
        return checker.spdf
```