```
from typing import Union, Dict, Optional, Any, List, Tuple
import json
import requests
import uuid
from string import Template

from pyspark.sql import DataFrame, SparkSession
from pyspark.sql import functions as F, types as T # noqa

# --- DQX Imports ---
from databricks.labs.dqx.profiler.profiler import DQProfiler
from databricks.labs.dqx.profiler.generator import DQGenerator
from databricks.labs.dqx.metrics_observer import DQMetricsObserver
from databricks.labs.dqx.config import InputConfig, WorkspaceFileChecksStorageConfig, LLMModelConfig
from databricks.labs.dqx.engine import DQEngine
from databricks.sdk import WorkspaceClient

# --- Local Config ---
from dqx_config import (
init_logging,
get_rules_path,
extrayendo_summary_stats,
create_dataframe,
filter_to_return_final_dfs,
DEFAULT_LLM_MODEL,
DQX_NOTIFY_TEMPLATE_STR
)

logger = init_logging("dqx_quality")

#
==========================================================================
# CLASE DQXAlert (Efectos Secundarios: Guardar y Notificar)
#
==========================================================================
```

```
class DQXAlert:
"""
Gestiona las acciones post-validación:
1. Guardado condicional de rechazados (Flag).
2. Notificación condicional (Template JSON) basada en métricas.
"""
def __init__(self, webhook_url: Optional[str] = None):
self.webhook_url = webhook_url

def save_rejected(self, df_rejected: DataFrame, target_table: str, save_flag: int) -> None:
"""
Inserta el df_rejected a una tabla Delta.
Flag: 1 guarda, 0 no guarda.
"""
if save_flag == 1:
try:
logger.info(f"💾 [DQXAlert] Guardando rechazados en: {target_table}")
(df_rejected.write
.format("delta")
.mode("append")
.option("mergeSchema", "true")
.saveAsTable(target_table)
)
logger.info("☑ [DQXAlert] Guardado exitoso.")
except Exception as e:
logger.error(f"❌ [DQXAlert] Error guardando rechazados: {e}")
# No lanzamos excepción para permitir que el pipeline continúe si solo falla el log
else:
logger.info("⏭ [DQXAlert] Flag=0. Se omite guardado de rechazados.")

def evaluate_and_notify(
self,
metrics_dict: Dict[str, Any],
table_name: str,
```

```
pipeline_run_id: str,
dqx_run_id: str,
audit_table: Optional[str] = None
) -> None:
"""
Engatilla la alerta SOLO si hay errors o warnings en el metrics_dict.
"""
# 1. Evaluar Métricas (Gatekeeper)
errors = int(metrics_dict.get("error_row_count", 0))
warnings = int(metrics_dict.get("warning_row_count", 0))

if errors == 0 and warnings == 0:
logger.info(f"☑ [DQXAlert] Calidad OK ({table_name}). No se envía alerta.")
return

if not self.webhook_url:
logger.warning("⚠# [DQXAlert] Hay errores pero no se configuró Webhook URL.")
return

try:
audit_url = "https://adb-tu-workspace.azuredatabricks.net" # ⚠# Configurar Base URL
if audit_table:
audit_url += f"/explore/data/{audit_table.replace('.', '/')}"

template_data = {
"table_name": table_name,
"pipeline_run_id": pipeline_run_id,
"dqx_run_id": dqx_run_id,
"input_row_count": str(metrics_dict.get("input_row_count", 0)),
"error_row_count": str(errors),
"warning_row_count": str(warnings),
"audit_url": audit_url
}
```

```
json_template = Template(DQX_NOTIFY_TEMPLATE_STR)
data_json = json_template.safe_substitute(template_data)

logger.info(f" 🚨 [DQXAlert] Enviando notificación a Teams...")
response = requests.post(
self.webhook_url,
data=data_json.encode("utf-8"),
headers={'Content-Type': 'application/json'},
timeout=10
)

if response.status_code in [200, 202]:
logger.info(f"☑ [DQXAlert] Notificación enviada.")
else:
logger.error(f"❌ [DQXAlert] Error Webhook: {response.status_code} - {response.text}")

except Exception as e:
logger.error(f"❌ [DQXAlert] Fallo crítico en notificación: {e}")



#
# ============================================================================
# CLASE DQXQuality (Generador de Data y Esquemas)
#
# ============================================================================
class DQXQuality:
def __init__(self, observer_name: str = "dq_monitoring_observer") -> None:
self.ws = WorkspaceClient()
self.observer = DQMetricsObserver(name=observer_name)
self.engine = DQEngine(self.ws, observer=self.observer)

# --- MÉTODOS PÚBLICOS ---

def apply_checks_table(
self,
table_name: str,
```

```
pipeline_run_id: str = "-1",
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[Dict, DataFrame, DataFrame, DataFrame]:
"""Aplica reglas a una tabla Delta/Hive existente."""
spark = SparkSession.builder.getOrCreate()
return self._execute_check_logic(
df_target=spark.table(table_name),
input_ref=table_name,
dataset_name=table_name,
pipeline_run_id=pipeline_run_id,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

def apply_checks_dataframe(
self,
df: DataFrame,
dataset_name: str = "dataframe_memory",
pipeline_run_id: str = "-1",
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[Dict, DataFrame, DataFrame, DataFrame]:
"""Aplica reglas a un DataFrame en memoria."""
return self._execute_check_logic(
df_target=df,
input_ref=dataset_name,
dataset_name=dataset_name,
pipeline_run_id=pipeline_run_id,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

# --- LÓGICA INTERNA ---
```

```
def _execute_check_logic(self, df_target: Any, input_ref: Any, dataset_name: str, pipeline_run_id: str, quality_check_filename: Optional[str], rules_path: Optional[str]):

try:

# 1. Cargar Reglas

ref_for_path = input_ref if isinstance(input_ref, str) else dataset_name

final_rules_path = get_rules_path(ref_for_path, dataset_name, quality_check_filename, rules_path)

dq_rules_raw = self.engine.load_checks(WorkspaceFileChecksStorageConfig(final_rules_path))

# 2. Ejecutar DQX

df_enriched, observation = self.engine.apply_checks_by_metadata(df_target, dq_rules_raw)

df_enriched.cache()

# 3. Obtener Métricas y RunID nativo

metrics_dict = observation.get

dqx_run_id = metrics_dict.get("run_id", str(uuid.uuid4()))

# 4. Construir DF SUMMARY

df_summary = self._build_summary_df(metrics_dict, dataset_name, pipeline_run_id, dqx_run_id)

# 5. Construir DF REJECTED

df_valid, df_rejected_raw = filter_to_return_final_dfs(df_enriched)

df_rejected_final = df_rejected_raw.select(

F.col("row_values"),

F.col("_errors").alias("errors"),

F.col("_warnings").alias("warnings"),

F.lit(dataset_name).alias("nombre_tabla"),

F.lit(pipeline_run_id).alias("pipeline_run_id"),

F.lit(dqx_run_id).alias("dqx_run_id")

)

return metrics_dict, df_summary, df_valid, df_rejected_final

except Exception as e:

logger.error(f"Error DQX Quality: {e}")
```

```
if 'df_enriched' in locals(): df_enriched.unpersist()
raise e

def _build_summary_df(self, metrics: Dict, dataset_name: str, pipeline_run_id: str, dqx_run_id: str) -> DataFrame:
rows = [(
int(metrics.get("input_row_count", 0)),
int(metrics.get("valid_row_count", 0)),
int(metrics.get("error_row_count", 0)),
int(metrics.get("warning_row_count", 0))
)]
schema = T.StructType([
T.StructField("input_row_count", T.LongType(), True),
T.StructField("valid_row_count", T.LongType(), True),
T.StructField("error_row_count", T.LongType(), True),
T.StructField("warning_row_count", T.LongType(), True)
])
extra_cols = [
F.current_timestamp().alias("fecha_actual"),
F.lit(dataset_name).alias("nombre_tabla"),
F.lit(pipeline_run_id).alias("pipeline_run_id"),
F.lit(dqx_run_id).alias("dqx_run_id")
]
return create_dataframe(schema, rows, extra_cols)

#
==============================================================================
# CLASE DQXProfiling (Profiling para Tablas y DataFrames)
#
==============================================================================
class DQXProfiling:
def __init__(self, workspace_client: Optional[WorkspaceClient] = None, model_name: str = DEFAULT_LLM_MODEL) -> None:
self.ws = workspace_client or WorkspaceClient()
self.llm_config = LLMModelConfig(model_name=model_name)
```

```
self.profiler = DQProfiler(self.ws, llm_model_config=self.llm_config)

self.generator = DQGenerator(self.ws, llm_model_config=self.llm_config)

self.engine = DQEngine(self.ws)

logger.info(f"DQXProfiling inicializado (Modelo: {model_name})")


def profile_table(self, table_name: str, columns: Optional[List[str]] = None, options:
Optional[Dict[str, Any]] = None, quality_check_filename: Optional[str] = None, save_path:
Optional[str] = None) -> DataFrame:

"""Profiling sobre una Tabla Delta/Hive."""

logger.info(f"Profiling tabla: {table_name}")

stats, profiles = self.profiler.profile_table(input_config=InputConfig(location=table_name),
columns=columns, options=options)

return self._finalize_profiling(table_name, stats, profiles, quality_check_filename, save_path)


def profile_dataframe(self, df: DataFrame, dataset_name: str = "dataframe_memory", columns:
Optional[List[str]] = None, options: Optional[Dict[str, Any]] = None, quality_check_filename:
Optional[str] = None, save_path: Optional[str] = None) -> DataFrame:

"""Profiling sobre un DataFrame en memoria."""

logger.info(f"Profiling dataframe: {dataset_name}")

stats, profiles = self.profiler.profile(df, columns=columns, options=options)

return self._finalize_profiling(dataset_name, stats, profiles, quality_check_filename, save_path)


def _finalize_profiling(self, identifier: str, stats: Dict[str, Any], profiles: Any,
quality_check_filename: Optional[str], save_path: Optional[str]) -> DataFrame:

rules = self.generator.generate_dq_rules(profiles)

final_path = get_rules_path(identifier, identifier, quality_check_filename, save_path)

self.engine.save_checks(rules, WorkspaceFileChecksStorageConfig(final_path))

return self._summary_stats_to_df(stats)


def _summary_stats_to_df(self, summary_stats: Dict[str, Any]) -> DataFrame:

columns_data = summary_stats.get("columns", summary_stats)

rows = extrayendo_summary_stats(columns_data)

schema = T.StructType([

T.StructField("columna", T.StringType(), True), T.StructField("total", T.LongType(), True),

T.StructField("nulos", T.LongType(), True), T.StructField("cardinalidad", T.LongType(), True),

T.StructField("min", T.StringType(), True), T.StructField("max", T.StringType(), True),
```

```
T.StructField("media", T.StringType(), True), T.StructField("desviacion_estandar", T.StringType(), True)

])

extra_cols = [F.when(F.col("total") > 0, F.round((F.col("nulos") / F.col("total")) * 100, 2)).otherwise(F.lit(0.0)).alias("porcentaje_nulos")]

return create_dataframe(schema, rows, extra_cols)
```