```
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,
get_rejected_df,
DEFAULT_LLM_MODEL,
DQX_NOTIFY_TEMPLATE_STR
)

logger = init_logging("dqx_quality")

#
============================================================================
# CLASE INTERNA: _DQXPersistence (I/O Simplificado)
#
============================================================================
class _DQXPersistence:
"""
Maneja el guardado de tablas y envío de alertas.
Asume que el Catálogo y Esquema ya existen en el ambiente (ctl_desa/prod).
"""
def __init__(self, webhook_url: Optional[str] = None):
self.webhook_url = webhook_url

def save_summary(self, df_summary: DataFrame, target_table: str) -> None:
"""
Guarda el historial de ejecución.
Spark creará la tabla automáticamente si no existe (siempre que el esquema exista).
"""
try:
logger.info(f"📊 Guardando métricas en: {target_table}")
(df_summary.write
.format("delta")
.mode("append")
.option("mergeSchema", "true")
.saveAsTable(target_table)
)
except Exception as e:
logger.error(f"❌ Error guardando summary en {target_table}: {e}")

def save_rejected(self, df_rejected: DataFrame, target_table: str) -> None:
"""
Guarda registros rechazados.
"""
try:
logger.info(f"💾 Guardando rechazados en: {target_table}")
(df_rejected.write
.format("delta")
.mode("append")
.option("mergeSchema", "true")
.saveAsTable(target_table)
)
logger.info("✅ Guardado de rechazados exitoso.")
except Exception as e:
logger.error(f"❌ Error guardando rechazados en {target_table}: {e}")

def evaluate_and_notify(self, metrics: Dict, table_name: str, pipeline_run_id: str, dqx_run_id:
str, audit_table: Optional[str]) -> None:
"""Envía alerta a Teams si hay errores."""
errors = int(metrics.get("error_row_count", 0))
warnings = int(metrics.get("warning_row_count", 0))

if errors == 0 and warnings == 0:
logger.info(f"✅ Calidad OK. No se envía alerta.")
return

if not self.webhook_url:
logger.warning("⚠# Hay errores pero no hay Webhook URL configurado.")
return

try:
# Construcción dinámica del enlace al Data Explorer
audit_url = "https://adb-6151277180747240.0.azuredatabricks.net" # URL Base Fija
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.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"🚨 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 not in [200, 202]:
logger.error(f"❌ Error Webhook: {response.status_code} - {response.text}")
else:
logger.info("✅ Notificación enviada.")

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


#
=============================================================================
# CLASE PRINCIPAL: DQXQuality (Facade)
#
=============================================================================
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,
summary_table: str, # OBLIGATORIO
audit_table: Optional[str] = None, # Opcional (si flag=1)
save_rejected_flag: int = 1, # 1=Guardar, 0=No
webhook_url: Optional[str] = None,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> DataFrame:
"""
Aplica reglas a una tabla Delta/Hive existente.
Guarda métricas y alertas internamente.
"""
spark = SparkSession.builder.getOrCreate()
return self._run_internal_flow(
df_target=spark.table(table_name),
input_ref=table_name,
dataset_name=table_name,
pipeline_run_id=pipeline_run_id,
summary_table=summary_table,
audit_table=audit_table,
save_rejected_flag=save_rejected_flag,
webhook_url=webhook_url,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

def apply_checks_dataframe(
self,
df: DataFrame,
dataset_name: str,
pipeline_run_id: str,
summary_table: str,
audit_table: Optional[str] = None,
save_rejected_flag: int = 1,
webhook_url: Optional[str] = None,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> DataFrame:
"""
Aplica reglas a un DataFrame en memoria.
"""
return self._run_internal_flow(
df_target=df,
input_ref=dataset_name,
dataset_name=dataset_name,
pipeline_run_id=pipeline_run_id,
summary_table=summary_table,
audit_table=audit_table,
save_rejected_flag=save_rejected_flag,
webhook_url=webhook_url,
quality_check_filename=quality_check_filename,
rules_path=rules_path
)

# --- LÓGICA INTERNA ---


def _run_internal_flow(self, df_target: DataFrame, input_ref: Any, dataset_name:
str, pipeline_run_id: str, summary_table: str, audit_table: str, save_rejected_flag: int,
webhook_url: str, quality_check_filename: str, rules_path: str) -> DataFrame:

# 1. Ejecutar Motor (Metrics + Rejected DF)
metrics, df_summary, df_rejected = self._execute_engine_logic(
df_target, input_ref, dataset_name, pipeline_run_id, quality_check_filename, rules_path
)

# 2. Persistencia
persistor = _DQXPersistence(webhook_url=webhook_url)
```

```
# A. Guardar Summary (Obligatorio)
persistor.save_summary(df_summary, summary_table)


# B. Guardar Rechazados (Condicional)
if save_rejected_flag == 1 and audit_table:
persistor.save_rejected(df_rejected, audit_table)

# C. Notificar
dqx_run_id = metrics.get("run_id", "unknown")
persistor.evaluate_and_notify(metrics, dataset_name, pipeline_run_id, dqx_run_id,
audit_table)


return df_rejected


def _execute_engine_logic(self, df_target: DataFrame, input_ref: Any, dataset_name: str,
pipeline_run_id: str, quality_check_filename: Optional[str], rules_path: Optional[str]):
try:
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))

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

# Materialización Obligatoria (Stall prevention)
df_enriched.cache()
_ = df_enriched.count()

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

# Construcción DFs
df_summary = self._build_summary_df(metrics_dict, dataset_name, pipeline_run_id,
dqx_run_id)
df_rejected_raw = get_rejected_df(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_rejected_final

except Exception as e:
logger.error(f"Error Engine Interno: {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 (Se mantiene igual)
#
==============================================================================
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)


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:
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:
stats, profiles = self.profiler.profile(df, columns=columns, options=options)
return self._finalize_profiling(dataset_name, stats, profiles, quality_check_filename,
save_path)


def generate_rules_from_prompt(self, user_prompt: str, input_data: Union[str, Any], options:
Optional[Dict[str, Any]] = None, quality_check_filename: Optional[str] = None, save_path:
Optional[str] = None) -> None:
target_name = input_data if isinstance(input_data, str) else "dataframe_memory"
if isinstance(input_data, str):
checks = self.generator.generate_dq_rules_ai_assisted(user_input=user_prompt,
input_config=InputConfig(location=input_data))
else:
stats, _ = self.profiler.profile(input_data, options=options)
checks = self.generator.generate_dq_rules_ai_assisted(user_input=user_prompt,
summary_stats=stats)
final_path = get_rules_path(target_name, target_name if isinstance(input_data, str) else None,
quality_check_filename, save_path)
self.engine.save_checks(checks, WorkspaceFileChecksStorageConfig(final_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)
```