```
from typing import Union, Dict, Optional, Any, List, Tuple

# --- Spark Imports ---
from pyspark.sql import DataFrame, SparkSession
import pyspark.sql.functions as F
from pyspark.sql.types import (
StructType,
StructField,
StringType,
LongType,
NumericType
)

# --- 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
)

logger = init_logging("dqx_quality")
```

```
#
================================================================================
# CLASE DQXProfiling
#
================================================================================
class DQXProfiling:
"""
Capa central de profiling y generación de reglas de calidad (DQX).
"""

def __init__(self, workspace_client: Optional[WorkspaceClient] = None) -> None:
self.ws = workspace_client or WorkspaceClient()
self.profiler = DQProfiler(self.ws)
self.generator = DQGenerator(self.ws)
self.engine = DQEngine(self.ws)
logger.info("DQXProfiling inicializado correctamente.")

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,
):
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(
identifier=table_name,
stats=stats,
profiles=profiles,
quality_check_filename=quality_check_filename,
save_path=save_path,
)

def profile_dataframe(
self,
df: Any,
columns: Optional[List[str]] = None,
options: Optional[Dict[str, Any]] = None,
quality_check_filename: Optional[str] = None,
save_path: Optional[str] = None,
):
logger.info("Profiling DataFrame en memoria")

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

return self._finalize_profiling(
identifier="dataframe_memory",
stats=stats,
profiles=profiles,
quality_check_filename=quality_check_filename,
save_path=save_path,
)

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

logger.info(f"Reglas de profiling guardadas en: {final_path}")
return self._summary_stats_to_df(stats)

def _summary_stats_to_df(self, summary_stats: Dict[str, Any]):
try:
columns_data = summary_stats.get("columns", summary_stats)
rows = extrayendo_summary_stats(columns_data)

schema = StructType([
StructField("columna", StringType(), True),
StructField("total", LongType(), True),
StructField("nulos", LongType(), True),
StructField("cardinalidad", LongType(), True),
StructField("min", StringType(), True),
StructField("max", StringType(), True),
StructField("media", StringType(), True),
StructField("desviacion_estandar", 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)

except Exception as e:
raise e

def generate_rules_from_prompt(
self,
user_prompt: str,
input_data: Union[str, Any],
model_name: str = "databricks/databricks-claude-sonnet-4-5",
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"
logger.info(f"IA Rules Generation | Target={target_name} | Model={model_name}")

try:
llm_config = LLMModelConfig(model_name=model_name)
ai_generator = DQGenerator(self.ws, llm_model_config=llm_config)

if isinstance(input_data, str):
checks = ai_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 = ai_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))
logger.info(f"Reglas IA guardadas en: {final_path}")

except Exception as e:
logger.error(f"Error generando reglas con IA: {e}")
raise e



#
============================================================================
# CLASE DQXQuality (Lógica Estricta: Error OR Warning = Fallo)
#
============================================================================
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)
logger.info(f"DQXQuality (DQX Native) inicializado.")

def apply_checks_table(
self,
table_name: str,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[DataFrame, DataFrame, DataFrame]:

spark = SparkSession.builder.getOrCreate()
logger.info(f"Applying checks on Table: {table_name}")

return self._execute_check_logic(
spark.table(table_name),
table_name,
table_name,
quality_check_filename,
rules_path
)

def apply_checks_dataframe(
self,
df: Any,
quality_check_filename: Optional[str] = None,
rules_path: Optional[str] = None
) -> Tuple[DataFrame, DataFrame, DataFrame]:

logger.info(f"Applying checks on DataFrame en Memoria")

return self._execute_check_logic(
df,
"dataframe_memory",
"dataframe_memory",
quality_check_filename,
rules_path
)

def _execute_check_logic(
self,
df_target: Any,
input_ref: Any,
dataset_name: str,
quality_check_filename: Optional[str],
rules_path: Optional[str]
) -> Tuple[DataFrame, DataFrame, DataFrame]:

try:
spark = SparkSession.builder.getOrCreate()
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)
)

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

df_enriched.cache()
count_res = df_enriched.count()
logger.info(f"Checks procesados. Total filas: {count_res}")
```

```
# 2. SUMMARY (KPIs)
metrics_dict = observation.get
df_summary = self._build_summary_df(spark, metrics_dict, dataset_name)

# 3. SPLIT INTELIGENTE (MODO ESTRICTO: Error OR Warning = Rechazo)

dqx_cols_to_drop = ["_errors", "_warnings"]

# --- DEFINICIÓN DE LÓGICA (Strict Null-based) ---

# RECHAZADO: Tiene errores O tiene warnings.
# Usamos | (OR Bitwise) y paréntesis obligatorios.
condicion_fallido = (F.col("_errors").isNotNull()) | (F.col("_warnings").isNotNull())

# VÁLIDO: No tiene errores Y no tiene warnings.
# Usamos & (AND Bitwise) y paréntesis obligatorios.
condicion_valido = (F.col("_errors").isNull()) & (F.col("_warnings").isNull())

# --- BIFURCACIÓN ---

# A) Datos Válidos: Completamente limpios
df_valid = df_enriched.filter(condicion_valido).drop(*dqx_cols_to_drop)

# B) Datos Rechazados: Tienen algún defecto (Error o Warning)
df_failed_raw = df_enriched.filter(condicion_fallido)

# Identificamos columnas de negocio
business_cols = [c for c in df_enriched.columns if c not in dqx_cols_to_drop]

# Transformación VARIANT (DBR 15.3+)
df_rejected = df_failed_raw.select(
F.parse_json(
F.to_json(F.struct(*[F.col(c) for c in business_cols]))
).alias("row_values"),
F.col("_errors"),
F.col("_warnings"),
F.current_timestamp().alias("fecha_auditoria")
)

return df_summary, df_valid, df_rejected
```

```
except Exception as e:
logger.error(f"Error aplicando checks: {e}")
if 'df_enriched' in locals(): df_enriched.unpersist()
raise e

def _build_summary_df(self, spark: SparkSession, metrics: Dict, dataset_name: Optional[str] = None):
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 = StructType([
StructField("total_filas", LongType(), True),
StructField("filas_validas", LongType(), True),
StructField("error_filas", LongType(), True),
StructField("advertencias_filas", LongType(), True)
])

extra_cols = [
F.current_timestamp().alias("fecha_actual"),
F.lit(dataset_name).alias("nombre_tabla"),
F.lit(-1).alias("pipeline_run_id")
]

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