🧊 DQX Utilities: El Estándar de Calidad de Datos en Pacífico

Bienvenido. Si estás leyendo esto, es porque te importa que los datos que mueves por nuestro Lakehouse (RDV, UDV, DDV) sean confiables.

DQX Utilities es el módulo centralizado dentro de `pacifico_utils`[^1] que democratiza el uso de Databricks Labs DQX. Su objetivo es eliminar la fricción de escribir validaciones manuales, delegando esa tarea a dos agentes especializados potenciados por IA.

🤔 ¿Qué es y por qué lo usamos?

Esta librería divide la gestión de calidad en dos roles claros para cubrir todo el ciclo de vida del dato:

- **El Arquitecto (DQXProfiling):** Se usa en Desarrollo. Te ayuda a definir las reglas. Puede analizar tus datos automáticamente o traducir tus requerimientos de negocio a código técnico.
- **El Guardián (DQXQuality):** Se usa en Producción. Vive dentro de tus Workflows de Databricks[^2] y se encarga de ejecutar las reglas, frenar datos malos y reportar métricas.

🏗️# Integración con el Framework Pacífico

Esta utilidad ha sido diseñada para encajar nativamente en nuestra arquitectura moderna:

- **Modularidad:** Vive dentro de `pacifico_utils`, lista para ser importada en cualquier Notebook de transformación (`nb_*.py`)[^3].
- **Eficiencia (Smart Sampling):** Al perfilar DataFrames gigantes, la herramienta aplica un muestreo inteligente automático para obtener estadísticas representativas sin disparar los costos de cómputo.
- **Seguridad:** Al usar la generación con IA, el sistema envía solo metadatos y estadísticas, minimizando la exposición de datos sensibles.

Importación

En cualquier notebook de tu flujo de trabajo:

```python
from pacifico_utils.dqx_utils import DQXProfiling, DQXQuality
```

🛠️# Fase 1: Desarrollo (Diseñando las Reglas)

Estás creando un nuevo proceso o explorando una fuente. Necesitas definir qué es "buena calidad" para estos datos antes de automatizar nada.

Opción A: "El Negocio me dio las reglas" (Generación con IA)

Si tienes los criterios de aceptación claros (ej: ticket en Jira), no pierdas tiempo escribiendo YAML a mano.

Usa `generate_rules_from_prompt`. La herramienta es **híbrida**:

- Si le pasas un **Nombre de Tabla**, consulta el esquema en Unity Catalog.
- Si le pasas un **DataFrame** (ej. lectura de un Parquet en RDV), calcula estadísticas rápidas para darle contexto a la IA.

```python
profiler = DQXProfiling()

# Escribe las reglas tal como te las dieron (Lenguaje Natural)
PROMPT = """
1. El ID de cliente debe ser único y obligatorio.
2. El correo electrónico debe tener formato válido.
3. El monto de transacción no puede ser negativo.
4. Si el estado es 'ACTIVO', la fecha de baja debe ser nula.
"""

# Genera el archivo YAML de configuración en tu Repo
profiler.generate_rules_from_prompt(
    user_prompt=PROMPT,
    input_data="hive_metastore.udv_clientes.maestra",  # O un objeto DataFrame
    quality_check_filename="reglas_udv_clientes"
)
```

¿Qué ocurre aquí? La IA analiza la estructura de tus datos + tu pedido y crea un archivo técnico estándar listo para ser versionado.

Opción B: "No conozco la data" (Perfilamiento Automático)

Ideal para fuentes nuevas, ya sean archivos sueltos en RDV/Landing o tablas ya registradas en UDV. Deja que la herramienta escanee los datos y te diga qué patrones encuentra.

```python
# CASO 1: Archivos en Data Lake (Landing / RDV)
# Leemos el archivo Parquet (recordar que en RDV todo es texto/parquet)
df_raw = spark.read.parquet("abfss://landing@pacificolake.../archivos/")

profiler.profile_dataframe(
    df=df_raw,
    quality_check_filename="reglas_base_landing"
)

# CASO 2: Tablas Registradas (UDV / Unity Catalog / Hive Metastore)
# Perfilamos directamente usando el nombre de la tabla
profiler.profile_table(
```

```
    table_name="hive_metastore.udv_ventas.fact_transacciones",
    quality_check_filename="reglas_base_ventas"
)

# CASO 3: Tablas Delta por Ruta (UDV / Time Travel)
# Útil si accedes a la tabla Delta por su ruta física o una versión específica
df_delta = spark.read.format("delta").load("abfss://udv@pacificolake.../fact_transacciones")

profiler.profile_dataframe(
    df=df_delta,
    quality_check_filename="reglas_base_delta_path"
)
```

🏭 Fase 2: Producción (Ejecutando el Control)

Ya tienes tus reglas (el archivo YAML generado). Ahora es momento de operacionalizarlas en nuestro entorno moderno.

En nuestra arquitectura, la lógica de transformación vive exclusivamente en Azure Databricks. Data Factory actúa únicamente como disparador a través de sus Mallas (carpeta 4-SCH)[^4], ejecutando Workflows de Databricks[^5].

Tu notebook productivo (`nb_*.py`) es una Task dentro de ese Workflow[^6]. Ahí es donde debes invocar al motor de calidad.

Implementación en el Notebook (nb_*.py)

El control de calidad (DQXQuality) actúa como un **Quality Gate**: valida los datos transformados en memoria antes de que se escriban en la siguiente capa.

```python
# ... [Bloque previo: Lectura de parámetros y Transformación con scripts .py] ...
# df_final = transform_logic(...)

# ============================================================
# 🛡️ QUALITY GATE (DQX)
# ============================================================
from pacifico_utils.dqx_utils import DQXQuality

logger.info("Iniciando validación de calidad DQX...")

quality_engine = DQXQuality()

# Ejecuta las reglas sobre el DataFrame transformado (en memoria)
# Esto asegura que NUNCA escribamos basura en UDV/DDV
df_resumen, df_validado = quality_engine.apply_checks_table(
```

```
    table_name="hive_metastore.udv_finanzas.reporte_final",  # O pasar df_final
    quality_check_filename="reglas_udv_finanzas",  # Archivo YAML creado en Dev
    return_enriched=True
)

# 1. Observabilidad:
# Muestra los KPIs en los logs del Driver del Workflow para revisión rápida.
display(df_resumen)

# 2. Circuit Breaker (Cortacircuitos):
# Si detectamos errores críticos, fallamos la tarea para que ADF detenga la malla.
cnt_errores = df_resumen.filter("error_filas > 0").count()

if cnt_errores > 0:
    logger.error(f"🚫 CALIDAD FALLIDA: Se detectaron {cnt_errores} reglas rotas.")

    # Detener el Workflow protegiendo la integridad de la capa siguiente
    # raise Exception("DQX Validation Failed: Data Integrity Issues Detected")

logger.info("☑️ Calidad validada exitosamente. Procediendo a escritura.")
```

🌐 Guía de Uso por Fuente de Datos

Nuestro framework se adapta a la capa donde estés trabajando:

Escenario	Tipo de Input	Método Recomendado	Ejemplo
Tablas Unity Catalog (Capas UDV / DDV)	String (Nombre de la tabla)	Pasa el nombre directo. La herramienta consulta el esquema en el metastore.	`input_data="hive_metastore.schema.tabla"`
Archivos en Data Lake (Capas Landing / RDV)	DataFrame (Spark Object)	Lee el archivo con Spark primero y pasa el objeto DataFrame.	`df = spark.read.parquet(...)` `input_data=df`
DataFrames en Memoria (Transformaciones In-flight)	DataFrame (Spark Object)	Ideal para validar pasos intermedios dentro del Workflow.	`input_data=df_transformado`

🧠 Capacidades "Power User"

1. Detección de Llaves Primarias (Auto-PK)

¿Recibiste un dataset en RDV sin documentación y no sabes cuál es el ID único? DQXProfiling usa algoritmos inteligentes para sugerirte la llave primaria.

```python
# Funciona leyendo el parquet directo
df_rdv = spark.read.parquet("abfss://landing@...")
pk_sugerida = profiler.detect_primary_keys(df_rdv)

print(f"La PK probable es: {pk_sugerida}")
```

2. Modelos de IA a la Carta

Por defecto, usamos modelos optimizados de Databricks. Si necesitas probar otro modelo, puedes parametrizarlo usando Widgets en tu notebook de desarrollo:

```python
# Usar Llama 3 para este análisis específico
model_name = dbutils.widgets.get("llm_model")

profiler.generate_rules_from_prompt(
    user_prompt=...,
    model_name=model_name
)
```

💡 Mejores Prácticas

1. **Versionamiento:** Guarda los archivos YAML de reglas generados en tu repositorio Git (carpetas `framework_*/quality_rules`)[^7]. Son parte de tu código.

2. **Iteración:** La IA es una asistente, no magia. Revisa siempre el YAML generado (`reglas_ia_*.yml`) y ajústalo si es necesario.

3. **Validación en Memoria:** Siempre prefiere validar el DataFrame antes de escribir (`apply_checks_dataframe`). Es más eficiente y seguro que escribir y luego validar.

Equipo de Ingeniería de Datos - Pacífico Seguros

[^1]: Módulo de utilidades compartidas del equipo
[^2]: Orquestación nativa de Databricks
[^3]: Convención de nomenclatura para notebooks productivos
[^4]: Estructura de carpetas en Azure Data Factory
[^5]: Jobs programados en Databricks
[^6]: Unidad de ejecución dentro de un Workflow
[^7]: Convención de versionamiento de reglas de calidad