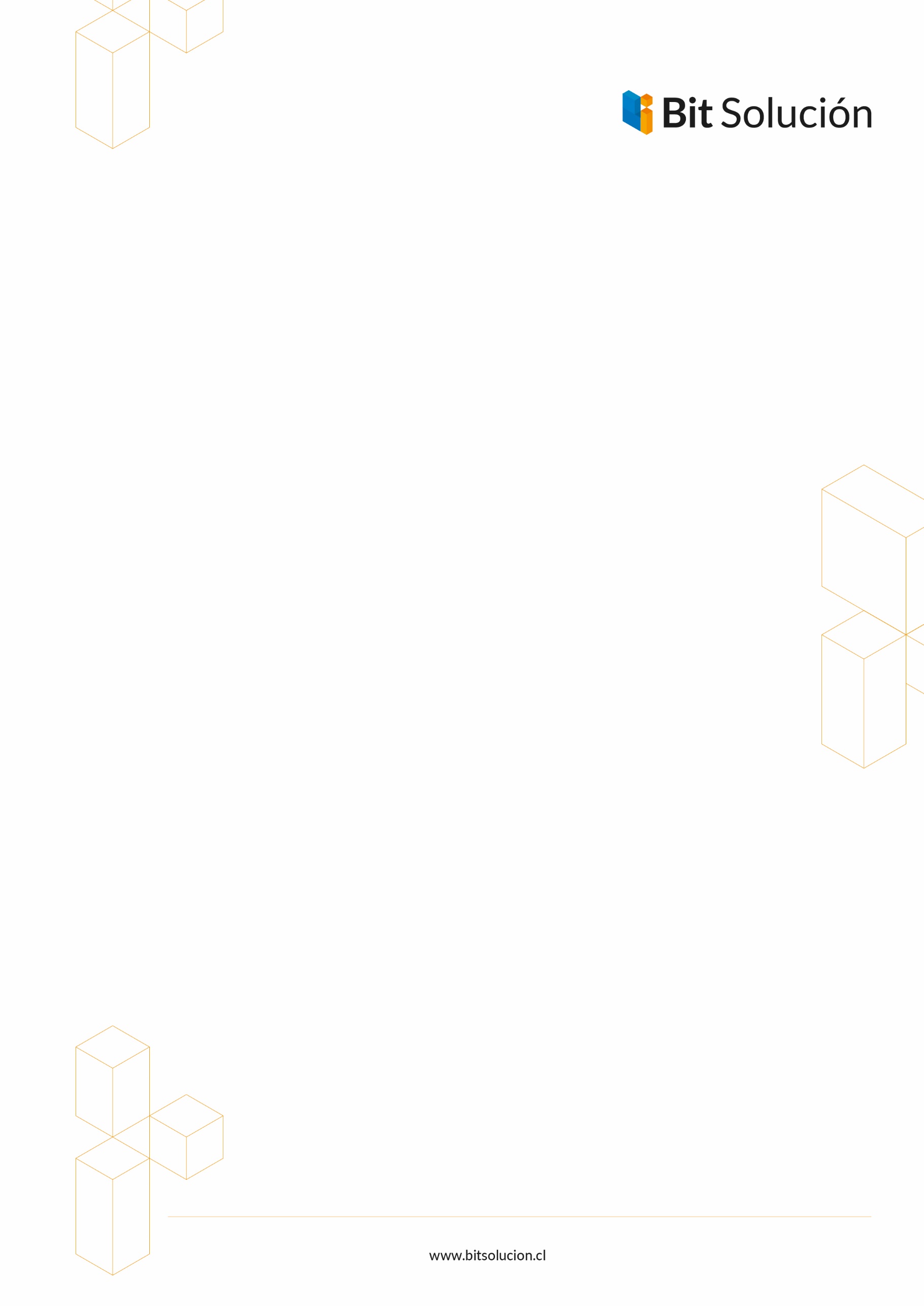


Estándar y Buenas Prácticas de Desarrollo en Databricks – Bit Solución



[Introducción 2](#_Toc206081278)

[Creación de Tablas en Databricks 3](#_Toc206081279)

[1. Estructura y Documentación del Notebook 3](#_Toc206081280)

[2. Parametrización y Configuración 3](#_Toc206081281)

[3. Modularidad y Reutilización de Código 4](#_Toc206081282)

[4. Validaciones Previas (Pre-flight Checks) 4](#_Toc206081283)

[5. Gestión de Metadatos: El Corazón del Notebook 4](#_Toc206081284)

[Creación de Notebook “Logica” 7](#_Toc206081285)

[2. Convenciones de Nomenclatura 8](#_Toc206081286)

[3. Estandarización de Parámetros (Widgets) 8](#_Toc206081287)

[4. Buenas Prácticas de Codificación 8](#_Toc206081288)

[4.1. Modularidad y Reutilización 8](#_Toc206081289)

[4.2. Idempotencia 9](#_Toc206081290)

[4.3. Legibilidad y Documentación 9](#_Toc206081291)

[4.4. Prácticas de SQL 10](#_Toc206081292)

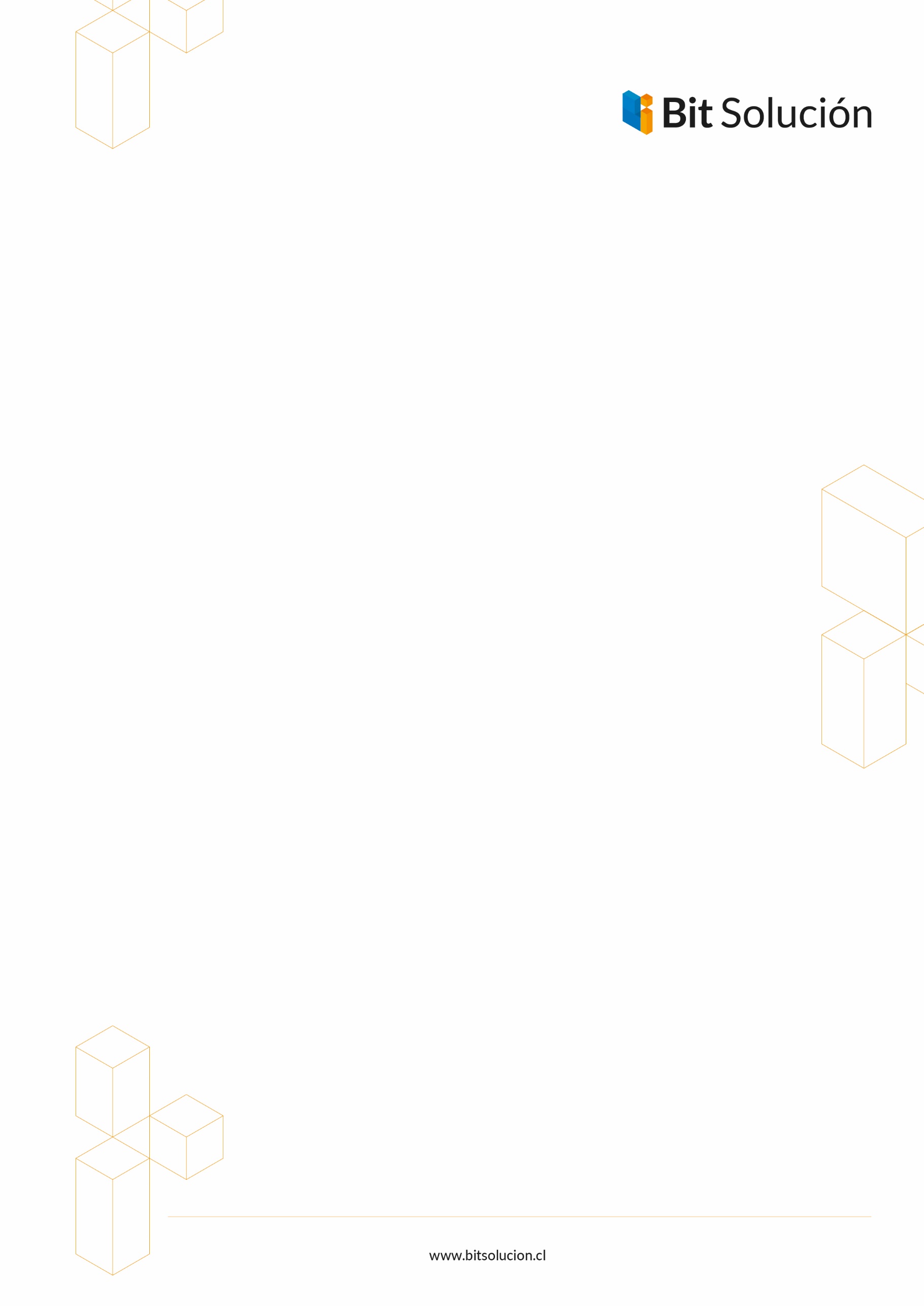
[5. Ciclo de Vida de los Datos y Procesos 10](#_Toc206081293)

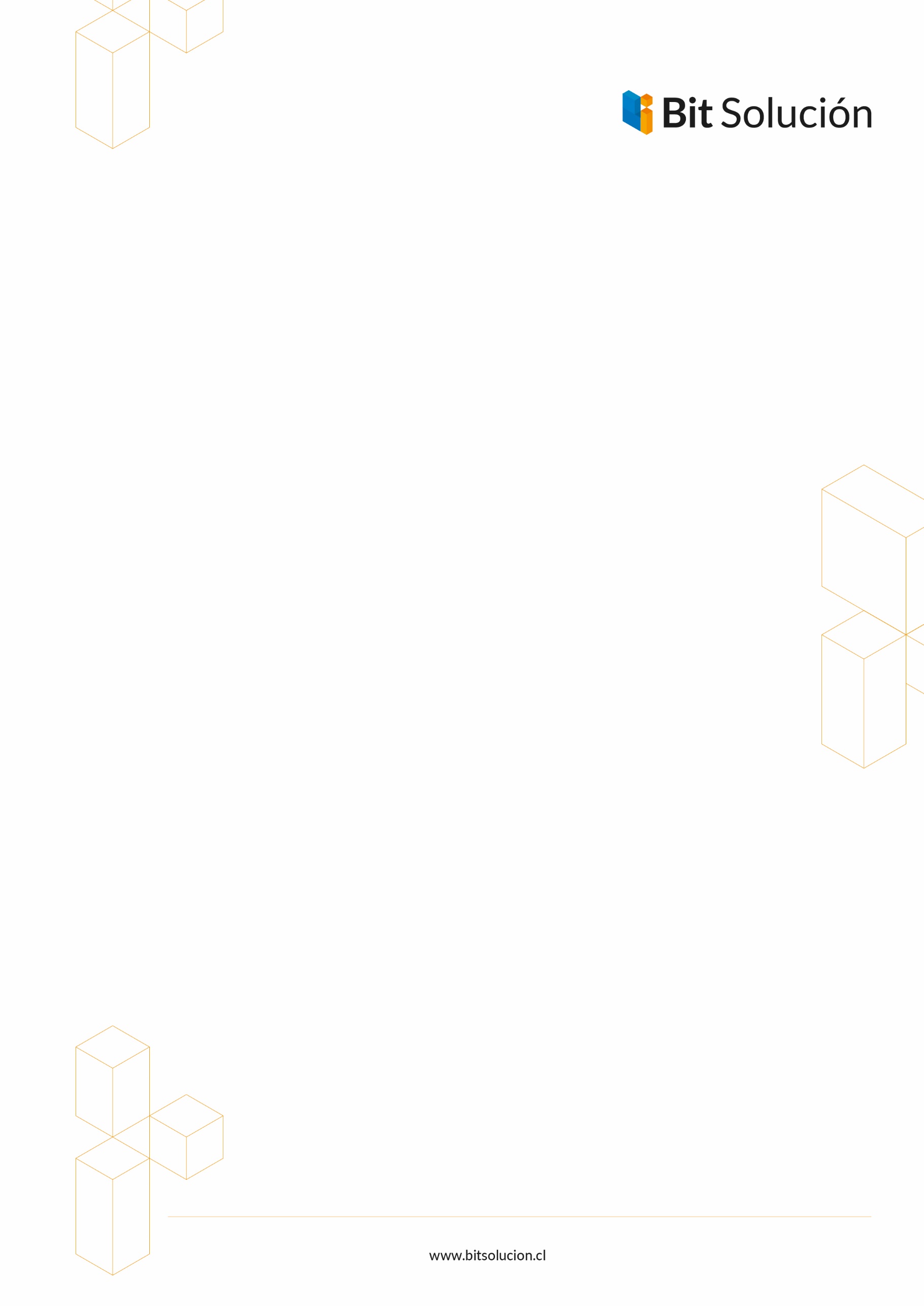
[5.1. Manejo de Tablas Temporales 10](#_Toc206081294)

[5.2. Generación de Archivos de Salida 11](#_Toc206081295)

[6. Orquestación y Finalización del Notebook 11](#_Toc206081296)

[Conclusión 12](#_Toc206081297)





Introducción

Este documento establece los estándares, convenciones y buenas prácticas que deben seguirse para el desarrollo de notebooks de Pyspark y SQL en el entorno de Databricks. El objetivo es asegurar la consistencia, legibilidad, mantenibilidad y robustez de todos los procesos desarrollados. Los ejemplos y reglas aquí descritos se basan en el análisis de los procesos existentes y desarrollados por la empresa.

Creación de Tablas en Databricks

El propósito de esta guía es estandarizar y mejorar la calidad de los notebooks de Databricks cuyo objetivo es la creación y gestión de metadatos de tablas, especialmente en un entorno donde los datos ya residen en un Data Lake (ADLS). El notebook provisto como ejemplo (BCI00\_Excesos\_Hip\_Creacion\_Modelo.py) sirve como una excelente base para estas prácticas.

A continuación, se detallan las mejores prácticas divididas por sección.

1. Estructura y Documentación del Notebook

Un notebook bien documentado es la base de la mantenibilidad. Cualquier desarrollador debería poder entender el propósito, el autor y el historial de cambios rápidamente.

* **Encabezado Detallado:** Incluir siempre un bloque de Markdown (%md) al inicio con información clave.
  + **Contenido Esencial:** Nombre del notebook, ruta, autor original, contacto del responsable de negocio o de sistemas (Ing. SW), fecha de creación y una descripción clara de su propósito.
  + **Ejemplo de Encabezado:**

Markdown

# MAGIC %md

# MAGIC ### Encabezado

# MAGIC \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

# MAGIC \* Nombre: BCI00\_Excesos\_Hip\_Creacion\_Modelo.ipynb

# MAGIC \* Autor: Sebastian Acuña

# MAGIC \* Ing. SW BCI: Rafael Montecino Toloza - rafael.montecinost@bci.cl

# MAGIC \* Fecha: 23/05/2025

# MAGIC \* Descripción: Proceso encargado de la creación de metadatos para tablas de entrada y la generación de tablas de salida para el proceso Exceso Hipotecario.

# MAGIC \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

* **Registro de Mantenciones:** Mantener un historial de cambios directamente en el notebook. Esto es crucial para el seguimiento de versiones y la depuración.
  + **Formato Sugerido:** Número de mantención, autor, fecha y una descripción concisa del cambio realizado.

2. Parametrización y Configuración

Los notebooks no deben contener valores "hardcodeados" (escritos directamente en el código). La parametrización es clave para la reutilización y la automatización en distintos entornos (desarrollo, QA, producción).

* **Uso de Widgets:** Utilizar dbutils.widgets para todas las variables externas, como nombres de bases de datos, rutas (locations) y parámetros de ejecución. Esto permite que los jobs de orquestación (como Databricks Workflows o Azure Data Factory) pasen valores dinámicamente.
  + **Ejemplo:**

Python

dbutils.widgets.text("db\_platinumW","","01 platinum DB:")

dbutils.widgets.text("db\_location\_platinumW","","05 Location DB Platinum:")

* **Asignación y Log de Parámetros:** Asignar los valores de los widgets a variables y mostrarlos en la salida. Esto es fundamental para la depuración, ya que el log de ejecución del job mostrará con qué parámetros se ejecutó.
  + **Ejemplo:**

Python

db\_platinumX = dbutils.widgets.get("db\_platinumW")

print("Parámetros de Ejecución:")

print(f"db\_platinumX: {db\_platinumX}")

3. Modularidad y Reutilización de Código

Evita duplicar código. Las funciones comunes deben centralizarse para facilitar su mantenimiento y asegurar la consistencia.

* **Notebook de Funciones:** Agrupar funciones de ayuda (helpers) en un notebook separado (ej. Funciones) y llamarlo usando el comando mágico %run.
* **Funciones Comunes:** Este notebook debería contener utilidades como:
  + valida\_param\_vacio(): Para asegurar que los parámetros obligatorios no estén vacíos.
  + valida\_ruta(): Para verificar que las rutas en ADLS existen.
  + sqlsafe(): Una función wrapper para ejecutar spark.sql() que puede incluir manejo de errores, logging o reintentos.

4. Validaciones Previas (Pre-flight Checks)

Antes de ejecutar la lógica principal (crear tablas), es fundamental validar que todas las condiciones y entradas son correctas. Esto previene fallos a mitad de proceso y arroja errores más claros.

* **Validar Parámetros:** Verificar que ningún widget esencial se haya quedado sin valor.
* **Validar Rutas:** Comprobar la existencia de las rutas (location) de las tablas de entrada para evitar errores de "Archivo no encontrado" más adelante.

5. Gestión de Metadatos: El Corazón del Notebook

Esta es la sección más importante. Es crucial diferenciar cómo se manejan las tablas de entrada (lectura de datos existentes) y las de salida (generación de nuevos datos).

* **Uso de CREATE DATABASE IF NOT EXISTS:** Para la creación de bases de datos, usar siempre IF NOT EXISTS para que el script sea idempotente (pueda ejecutarse varias veces sin fallar). Especificar siempre la LOCATION.
* **Distinción entre Tablas de Entrada y Salida:**
  + **Tablas de Entrada (Ej: RiesgoCred, RiesgoCred\_RiesgoCredEmp)**
    - **Objetivo:** Crear únicamente el **metadato** (la definición en el metastore de Hive) que apunta a los datos que **ya existen** en el Data Lake.
    - **Práctica:**
      1. **(Opcional pero recomendado) DROP TABLE IF EXISTS db.tabla:** Esto asegura que si la estructura de la tabla (columnas, tipos de datos) ha cambiado en el notebook, se aplicará la nueva definición. Sin el DROP, un CREATE IF NOT EXISTS no haría nada si la tabla ya existe, ignorando los cambios.
      2. **CREATE TABLE IF NOT EXISTS db.tabla (...) USING DELTA LOCATION 'path/to/data':**
         * IF NOT EXISTS: Aporta idempotencia a la operación.
         * USING DELTA: Define el formato de la tabla.
         * LOCATION: Es la cláusula más importante. Le dice a Spark dónde encontrar los datos físicos en ADLS. **No se mueven ni se copian datos.**
         * PARTITIONED BY: Definir las particiones es crítico para el rendimiento de las consultas.
         * COMMENT: **Documentar cada tabla y cada columna.** Esto es invaluable para el gobierno de datos y para que otros usuarios entiendan el propósito de cada campo.
  + **Tablas de Salida (Ej: platinum)**
    - **Objetivo:** Crear una tabla que contendrá el resultado de un proceso. Se debe asegurar que cada ejecución genere un resultado limpio y completo, sin datos de ejecuciones anteriores.
    - **Práctica:**
      1. **DROP TABLE IF EXISTS db.tabla\_salida:** Elimina el metadato de la tabla del metastore.
      2. **dbutils.fs.rm('path/to/output/data', recurse=True):** **Este paso es fundamental.** Como la tabla es externa (LOCATION), el DROP TABLE no borra los datos en ADLS. Este comando elimina explícitamente los archivos de la ejecución anterior, asegurando un inicio limpio.
      3. **CREATE TABLE db.tabla\_salida (...) USING DELTA LOCATION 'path/to/output/data':** Se crea la nueva definición de la tabla vacía, apuntando al location que acaba de ser limpiado. El proceso posterior se encargará de poblar esta tabla.

**6. Finalización del Notebook**

* **Salida Controlada:** Utilizar dbutils.notebook.exit() al final del script. Esto permite devolver un mensaje de estado (éxito o error) al orquestador. Un formato JSON es ideal para pasar múltiples valores, como un código de error y un mensaje descriptivo.
  + **Ejemplo de Éxito:**

Python

dbutils.notebook.exit("{\"coderror\":\"0\", \"msgerror\":\"Notebook termina ejecucion satisfactoriamente\"}")

Al seguir estas prácticas, los notebooks de creación de tablas no solo serán funcionales, sino también robustos, fáciles de automatizar, sencillos de depurar y, lo más importante, se integrarán de manera efectiva en una estrategia de gobierno de datos gracias a su auto-documentación y claridad estructural.

Creación de Notebook “Logica”

Todo notebook debe seguir una estructura estandarizada y documentada mediante celdas Markdown. Las secciones obligatorias son:

1. **Título Principal:** El nombre del notebook como título principal (# BCIXX\_Nombre\_Proceso).
2. **Información del Notebook:**
   * **Encabezado:** Metadatos del notebook (Nombre, Ruta, Autor, Contacto, Fecha, Descripción).
   * **Mantenciones:** Un historial de cambios para registrar modificaciones.
   * **Tablas Entrada y Salida:** Lista explícita de las tablas o archivos principales que consume y produce el notebook.
3. **Captura de Variables:**
   * **Crear Widgets para Captura de Variables:** Celda donde se definen todos los dbutils.widgets.
   * **Asignar Objeto a Lectura de Widgets y Variables:** Celda donde se leen los widgets y se asignan a variables de Python y configuraciones de Spark.
4. **Carga de Funciones:**
   * Celda que invoca al notebook de Funciones centralizado mediante %run "../Funciones".
5. **Validación de Ingreso de Parámetros:**
   * Bloques dedicados a validar los parámetros de entrada (vacíos, existencia de BD, formato de fecha, existencia de rutas).
6. **Asignación de Variables de Fecha:**
   * Celda donde se generan todas las variantes de formato de fecha que se usarán en el proceso (anomes, fechanormativo, etc.).
7. **Inicio de Lógica:**
   * **Creación/Eliminación de Tablas Temporales:** Bloque que prepara el entorno, eliminando y/o creando las tablas temporales que se usarán.
   * **Desarrollo Lógica:** Sección principal dividida en Pasos (#### Paso X: Descripción de la tarea). Cada paso debe tener una celda Markdown descriptiva seguida de la celda de código que la ejecuta.
8. **Mensaje Final:**
   * La última celda del notebook, que debe invocar a dbutils.notebook.exit con un mensaje de estado en formato JSON.

2. Convenciones de Nomenclatura

Una nomenclatura consistente es clave para la legibilidad.

* **Notebooks:** Deben ser numerados secuencialmente y tener un nombre descriptivo.
  + [PrefijoProceso]XX\_[Etapa]\_[Componente].py
  + Ejemplo: BCI01\_Ruta\_Exceso\_Hip.py, BCI06\_Ruta\_Exceso\_Hip\_Eliminacion\_Tablas\_Temp.py.
* **Variables:**
  + **Widgets:** Sufijo W. Ejemplo: fecha\_procesoW, platinum\_temp\_dbW.
  + **Variables de Python leídas de Widgets:** Sufijo X. Ejemplo: fecha\_procesoX, platinum\_temp\_dbX.
  + **Variables de Configuración de Spark:** Prefijo del proyecto. Ejemplo: spark.conf.set("bci.fecha\_procesoX", fecha\_procesoX).
* **Tablas Temporales:** Deben seguir una estructura que identifique su origen y propósito.
  + [prefijo]\_[num\_notebook]\_[proyecto]\_tmp\_[proposito]
  + Ejemplo: bci01\_exchip\_tmp\_erc\_gtia\_hip\_b1, bci03\_exchip\_tmp\_exceso\_cli.
* **Variables de Consultas SQL:** Nombrar la variable de forma que describa la acción y la tabla.
  + Ejemplo: drop\_erc\_gtias\_hip\_b1, vista\_exceso\_cli, insert\_gtia\_exceso\_hip.

3. Estandarización de Parámetros (Widgets)

* Todo valor de configuración externo (fechas, rutas, nombres de BD, nombres de archivos) **debe** ser un widget.
* Los widgets se deben definir en la sección "Crear Widgets para Capturar de Variables".
* La etiqueta del widget debe ser descriptiva e incluir un número de orden.

Python

dbutils.widgets.text("fecha\_procesoW", "", "01 Fecha Proceso:")

dbutils.widgets.text("platinum\_temp\_dbW", "", "02 platinum temp db:")

* Al inicio de la ejecución, todos los widgets deben ser eliminados (dbutils.widgets.removeAll()) para evitar la persistencia de valores entre ejecuciones manuales.

4. Buenas Prácticas de Codificación

4.1. Modularidad y Reutilización

* Toda lógica común de validación, formateo o ejecución debe residir en el notebook Funciones.
* Se debe implementar una función sqlsafe() o similar para encapsular la ejecución de consultas SQL, permitiendo un manejo de errores centralizado si fuese necesario.

4.2. Idempotencia

La idempotencia es crucial: un proceso debe poder ejecutarse múltiples veces con los mismos parámetros y producir siempre el mismo resultado, sin efectos secundarios no deseados.

* **Para Tablas Temporales:** Siempre realizar una limpieza en dos pasos antes de crear o poblar una tabla.
  1. Eliminar la tabla del metastore: DROP TABLE IF EXISTS mi\_db.mi\_tabla\_tmp.
  2. Eliminar los datos físicos subyacentes: dbutils.fs.rm("ruta/a/los/datos", True).

Python

drop\_tabla = """DROP TABLE IF EXISTS """ + db\_tempX + """.mi\_tabla\_tmp"""

sqlsafe(drop\_tabla)

dbutils.fs.rm(ruta\_tempX + "mi\_tabla\_tmp", True)

* **Para Tablas Finales (Cargas Incrementales):** Antes de insertar los datos de la partición actual, se debe eliminar la partición existente para evitar duplicados.

SQL

delete\_query = """DELETE FROM """ + db\_finalX + """.tabla\_final WHERE fec\_proceso = '""" + fecha\_procesoX + """' """

sqlsafe(delete\_query)

insert\_query = """INSERT INTO """ + db\_finalX + """.tabla\_final SELECT col1, col2 FROM ..."""

sqlsafe(insert\_query)

4.3. Legibilidad y Documentación

* **Comentar es Obligatorio:** Cada paso lógico (Paso X) debe estar precedido por una celda Markdown que explique claramente el propósito de la siguiente celda de código.
* **SQL Legible:** Las consultas SQL deben estar formateadas con indentación para facilitar su lectura. Utilizar alias descriptivos para las tablas.
* **Comentarios en Lógica Compleja:** En lógicas de negocio complejas dentro de un CASE WHEN, añadir comentarios en línea para explicar reglas específicas.

SQL

-- Ejemplo de comentario en línea en SQL

case when cod\_stp\_cart = 'ACT' then 0 --DEUDA ACTIVA

when ind\_cdet != 'N' then 100 --DEUDA CTG DET

else 100 --OTRA DEUDA CONTINGENTE

end as fcc

4.4. Prácticas de SQL

* **Evitar el uso de SELECT \*:** Nunca se debe utilizar SELECT \* para consultar tablas, ya sean de origen o temporales. Siempre se deben nombrar explícitamente todas las columnas requeridas en la consulta.
  + **Justificación:**
    - **Robustez:** El uso de SELECT \* hace que el código sea frágil. Si la estructura de la tabla de origen cambia (se añade, elimina o reordena una columna), el código puede fallar o, peor aún, procesar datos incorrectamente sin arrojar un error.
    - **Legibilidad y Mantenimiento:** Nombrar las columnas hace que el código sea autodocumentado, ya que queda claro qué datos se están utilizando en cada paso del proceso.
    - **Rendimiento:** Seleccionar solo las columnas necesarias puede reducir la cantidad de datos que se mueven a través de la red y que se procesan en memoria, mejorando el rendimiento.
  + **Práctica Incorrecta:**

SQL

-- MAL: Frágil y poco claro

INSERT INTO db\_temp.tabla\_destino

SELECT \* FROM db\_origen.tabla\_fuente;

* + **Práctica Correcta:**

SQL

-- BIEN: Robusto, legible y eficiente

INSERT INTO db\_temp.tabla\_destino

SELECT

id\_cliente,

monto\_compra,

fecha\_transaccion,

id\_vendedor as id\_vendedor\_fuente -- Renombrar para mayor claridad

FROM

db\_origen.tabla\_fuente

WHERE

fecha\_transaccion >= '2025-01-01';

5. Ciclo de Vida de los Datos y Procesos

5.1. Manejo de Tablas Temporales

* Las tablas temporales se crean al inicio de cada notebook relevante.
* Su propósito es materializar resultados intermedios para mejorar el rendimiento y la claridad del proceso.
* **Es obligatorio que exista un notebook de limpieza** (como BCI06\_...\_Eliminacion\_Tablas\_Temp) que elimine todas las tablas temporales generadas en el flujo para liberar recursos.

5.2. Generación de Archivos de Salida

El proceso para generar y entregar archivos planos debe seguir un patrón robusto y explícito:

1. **Formateo en Spark:** Usar SQL para concatenar los campos y unir la cabecera, creando un DataFrame de una sola columna.
2. **Escritura en Staging (DBFS):** Escribir el DataFrame como un archivo de texto único (.coalesce(1)) en una ruta temporal en DBFS.
3. **Renombrado:** Listar el directorio de staging, encontrar el archivo part-00000... y renombrarlo al nombre final requerido (dbutils.fs.mv).
4. **Limpieza del Staging:** Eliminar los archivos residuales del directorio de staging.
5. **Limpieza del Destino Final (ADLS):** **Siempre** eliminar los archivos existentes en la carpeta de destino final en ADLS para asegurar una entrega limpia.
6. **Copia a ADLS:** Copiar el archivo renombrado desde DBFS a la ubicación final en ADLS (dbutils.fs.cp).

6. Orquestación y Finalización del Notebook

* Cada notebook debe finalizar con una llamada a dbutils.notebook.exit().
* El argumento de salida debe ser un string en formato JSON que contenga, como mínimo, un código de error y un mensaje descriptivo. Esto es fundamental para que las herramientas de orquestación (como Databricks Workflows o ADF) puedan determinar el éxito o fracaso de la ejecución.

Python

dbutils.notebook.exit("{\"coderror\":\"0\", \"msgerror\":\"Notebook termina ejecucion satisfactoriamente\"}")

Conclusión

En conclusión, el documento "Estándar y Buenas Prácticas de Desarrollo en Databricks" establece un marco de trabajo integral y riguroso diseñado para garantizar la consistencia, robustez y mantenibilidad de los procesos de datos. La guía enfatiza la importancia de una estructura y documentación disciplinada desde el inicio de cualquier desarrollo, exigiendo encabezados detallados y registros de mantenimiento en cada notebook.

Los pilares fundamentales de este estándar son la automatización y la fiabilidad, promovidas a través de prácticas clave como:

* **Parametrización total:** Se prohíben los valores "hardcodeados", exigiendo el uso de widgets para toda configuración externa, lo que facilita la orquestación y la ejecución en diferentes entornos.
* **Modularidad:** Se centraliza la lógica común, como validaciones y ejecución de SQL, en notebooks de funciones reutilizables para evitar la duplicación de código.
* **Idempotencia:** Se recalca como un principio crucial que los procesos deben poder re-ejecutarse sin generar resultados inconsistentes o errores, detallando los procedimientos de limpieza para tablas temporales y finales.
* **Calidad de Código:** Se imponen reglas estrictas como la prohibición del uso de SELECT \* para asegurar la robustez y legibilidad del código SQL.

Además, el documento define un ciclo de vida claro para los datos, diferenciando el manejo de metadatos para tablas de entrada existentes de la creación y limpieza de tablas de salida y temporales. Finalmente, se estandariza la comunicación con las herramientas de orquestación mediante una salida controlada con

dbutils.notebook.exit(), asegurando que el estado de cada ejecución sea explícito y manejable.

Al seguir estas directrices, los equipos de desarrollo pueden construir soluciones de datos en Databricks que no solo son eficientes y escalables, sino también fáciles de depurar, mantener y colaborar, sentando las bases para un gobierno de datos efectivo.