Carga de Datos a Destinos - Día 4

Change Status

in-progress 40 min

Learning Objectives

1. Implementar estrategias eficientes de carga a bases de datos SQL usando to_sql()
2. Comprender diferencias entre carga completa vs incremental
3. Gestionar integridad referencial y conflictos durante la carga

Actividades y Aprendizajes

Aprende todo sobre funciones y módulos en Python con ejemplos prácticos.

Task 1: Carga Básica a Bases de Datos SQL (10 minutos)

La carga representa el **momento culminante** del pipeline ETL, donde datos transformados se integran al sistema destino de manera eficiente y segura.

to_sql(): El Puente entre Pandas y SQL

Parámetros esenciales: Conexión, nombre de tabla, estrategia de conflicto.

```
# Carga básica
df.to_sql('clientes', conn, index=False, if_exists='replace')

# Parámetros importantes:
# - index=False: No guardar índice de DataFrame como columna
# - if_exists='replace': Reemplazar tabla si existe
```

```
# - if_exists='append': Agregar a tabla existente
# - if_exists='fail': Error si tabla existe
```

Tipos de datos automáticos: Pandas infiere tipos SQL apropiados:

`int64` → `INTEGER`
`float64` → `REAL/FLOAT`
`object` → `TEXT/VARCHAR`
`datetime64` → `TIMESTAMP`
Estrategias de Carga: Replace vs Append

Replace completa: Ideal para datasets pequeños o cuando se necesita historial limpio.

```
# Reemplazar tabla completamente
df_clientes.to_sql('clientes', conn, index=False, if_exists='replace')
df_pedidos.to_sql('pedidos', conn, index=False, if_exists='replace')
```

Append incremental: Para agregar nuevos datos sin perder existentes.

```
# Agregar datos nuevos
df_nuevos_clientes.to_sql('clientes', conn, index=False, if_exists='append')
```

Consideraciones de performance: Replace puede ser más rápido para tablas pequeñas, append es esencial para datos históricos.

Task 2: Estrategias de Carga Avanzadas (10 minutos)

Las estrategias avanzadas manejan **escenarios complejos** donde la carga simple no es suficiente, requiriendo lógica adicional para integridad y performance.

Upsert: Actualizar o Insertar

Problema resuelto: ¿Qué hacer cuando un registro puede existir o no?

```
# Implementación manual de upsert (SQLite)
def upsert_dataframe(df, table_name, conn, key_columns):
```

```
        for _, row in df.iterrows():
            # Verificar si existe
            key_values = tuple(row[col] for col in key_columns)
            placeholders = ','.join(['?'] * len(key_columns))
            where_clause = ' AND '.join([f'{col} = ?' for col in key_columns])

            cursor = conn.cursor()
            cursor.execute(f'SELECT COUNT(*) FROM {table_name} WHERE {where_clause}', key_values)

            if cursor.fetchone()[0] > 0:
                # UPDATE
                set_clause = ', '.join([f'{col} = ?' for col in df.columns if col not in key_columns])
                values = tuple(row[col] for col in df.columns if col not in key_columns) + key_values
                cursor.execute(f'UPDATE {table_name} SET {set_clause} WHERE {where_clause}', values)
            else:
                # INSERT
                columns = ','.join(df.columns)
                placeholders = ','.join(['?'] * len(df.columns))
                values = tuple(row)
                cursor.execute(f'INSERT INTO {table_name} ({columns}) VALUES ({placeholders})', values)

        conn.commit()
```

Alternativas por motor:

PostgreSQL: `ON CONFLICT ... DO UPDATE`
MySQL: `INSERT ... ON DUPLICATE KEY UPDATE`
SQL Server: `MERGE` statement
Carga por Chunks para Grandes Volúmenes

Problema de memoria: DataFrames grandes pueden agotar RAM durante carga.

```
# Carga por chunks
chunk_size = 10000
for i in range(0, len(df), chunk_size):
    chunk = df.iloc[i:i + chunk_size]
    chunk.to_sql('ventas', conn, index=False, if_exists='append', method='multi')
```

Parámetro method:

`'multi'` para múltiples INSERT statements
None para usar executemany (más eficiente)

Optimización de Índices Durante Carga

Desactivar índices durante carga masiva:

```
# Para PostgreSQL/MySQL
conn.execute('ALTER TABLE ventas DISABLE KEYS;')  # MySQL
# o
conn.execute('ALTER TABLE ventas DROP CONSTRAINT pk_ventas CASCADE;')  # PostgreSQL

# Cargar datos
df.to_sql('ventas', conn, if_exists='append')

# Reactivar índices
conn.execute('ALTER TABLE ventas ENABLE KEYS;')
conn.commit()
```

Task 3: Integridad Referencial y Validaciones (10 minutos)

La carga no termina con escribir datos; debe asegurar que las **relaciones entre tablas permanezcan válidas** y consistentes.

Manejo de Claves Foráneas

Problema: Intentar cargar pedidos con clientes que no existen.

```
# Verificar integridad antes de carga
clientes_existentes = pd.read_sql('SELECT id_cliente FROM clientes', conn)
pedidos_validos = df_pedidos[df_pedidos['id_cliente'].isin(clientes_existentes['id_cliente'])]

if len(pedidos_validos) < len(df_pedidos):
    pedidos_invalidos = df_pedidos[~df_pedidos['id_cliente'].isin(clientes_existentes['id_cliente'])]
    print(f"Pedidos con clientes inexistentes: {len(pedidos_invalidos)}")
    # Loggear o manejar errores

pedidos_validos.to_sql('pedidos', conn, index=False, if_exists='append')
```

Transacciones para Consistencia

Atomicidad: Toda la carga debe suceder o ninguna parte.

```
try:
    conn.execute('BEGIN TRANSACTION;')

    # Cargar tablas relacionadas
    df_clientes.to_sql('clientes', conn, index=False, if_exists='append')
    df_pedidos.to_sql('pedidos', conn, index=False, if_exists='append')
    df_detalles.to_sql('detalle_pedidos', conn, index=False, if_exists='append')

    conn.execute('COMMIT;')
    print("Carga completada exitosamente")

except Exception as e:
    conn.execute('ROLLBACK;')
    print(f"Error en carga, transacción revertida: {e}")
```

Validaciones Post-Carga

Contar registros: Verificar que se cargaron todos los datos.

```
registros_esperados = len(df)
registros_cargados = pd.read_sql('SELECT COUNT(*) FROM clientes', conn).iloc[0,0]

if registros_cargados != registros_esperados:
    print(f"Error: Esperados {registros_esperados}, cargados {registros_cargados}")
```

Verificar constraints: Asegurar que no se violaron reglas de integridad.

```
# Verificar claves foráneas
orphans = pd.read_sql('''
    SELECT COUNT(*) FROM pedidos p
    LEFT JOIN clientes c ON p.id_cliente = c.id_cliente
    WHERE c.id_cliente IS NULL
''', conn)
```