

# Informe Tecnico: Analisis de Estrategias de Integracion ETL para ERP

Fecha: 30 de Diciembre de 2024

Elaborado por: Equipo de Mejoramiento Continuo

Version: 2.0 (Con Resultados Reales de Pruebas)

## 1. Resumen Ejecutivo

Este informe presenta un analisis comparativo entre dos estrategias para implementar procesos ETL (Extract, Transform, Load) que se integraran con el ERP corporativo basado en Java. Se evaluaron las opciones de **Java puro** versus un enfoque **hibrido (Java + Python)** para la carga masiva de datos desde archivos Excel hacia PostgreSQL.

**Hallazgo principal:** El enfoque hibrido presenta un rendimiento **11.5 veces superior** (2.86s vs 32.98s) con una complejidad de implementacion similar.

## 2. Contexto y Alcance

### 2.1 Problema a Resolver

- Cargar datos desde archivos Excel (.xslm) hacia base de datos PostgreSQL
- Volumen de datos: ~38,000 filas por archivo
- Integracion con ERP monolitico existente en Java
- Ambiente de ejecucion: Servidor Windows

### 2.2 Opciones Evaluadas

Opcion	Descripcion
A. Java Puro	Apache POI + JDBC/PostgreSQL Driver
B. Python Directo	Polars + ADBC (referencia de rendimiento maximo)
C. Hibrido - Java invoca Python	Java orquesta via ProcessBuilder, Python (Polars) procesa en envv aislado

## 3. Benchmark de Rendimiento

### 3.1 Condiciones de Prueba

Parametro	Valor
Archivo origen	BASE DE DATOS GENERAL.xslm
Filas procesadas	37,879
Columnas	30
Ubicacion archivo	Red compartida (\\192.168.0.3\...)
Base de datos destino	PostgreSQL 15

Hardware	Servidor Windows estandar
----------	---------------------------

### 3.2 Resultados Medidos (Pruebas Reales)

#### Opcion A: Java Puro (ExcelToPostgresLoader)

```
[1/3] Leyendo archivo Excel...
      Archivo: \\192.168.0.3\...\BASE DE DATOS GENERAL.xlsm
      Hoja: MOVIMIENTO_DE ORDENES__2
      Filas brutas encontradas: 37889
      Cabeceras encontradas en fila: 10
      Columnas seleccionadas: 30
      Lectura completada en 31,08 segundos
      Filas de datos: 37879

[2/3] Preparando subida a PostgreSQL...
      Tabla creada exitosamente.

[3/3] Insertando datos...
      Insertados: 37879 filas. COMPLETADO!
      Subida completada en 1,90 segundos

TIEMPO TOTAL JAVA: 32,98 segundos
```

#### Opcion B: Python Directo (Polars + ADBC)

```
--- Iniciando Carga Buffer RAPIDA (Polars) ---
Leyendo archivo Excel...
      Archivo leído en 1.60s. Filas: 37879, Columnas: 31
      Normalizando columnas y limpiando datos numericos...
      Columnas finales seleccionadas (30)

Subiendo datos a PostgreSQL...
      Usando motor ADBC (Ultra Rapido)
      Carga completada exitosamente.

Tiempo total operacion: 2.59 segundos
```

#### Opcion C: Hibrido Java -> Python (PythonRunner + venv)

```
=====
--- JAVA: Ejecutando Script Python ---
=====

Ejecutando: ...\.venv\Scripts\python.exe upload_buffer_polars.py
---
--- Iniciando Carga Buffer RAPIDA (Polars) ---
      Archivo leído en 1.58s. Filas: 37879, Columnas: 31
      Columnas finales seleccionadas (30)
      Usando motor ADBC (Ultra Rapido)
      Carga completada exitosamente.
```

Tiempo total operacion: 2.04 segundos.

=====

Tiempo total: 2,86 segundos

Codigo de salida: 0 (EXITO)

=====

3.3 Comparativa de Tiempos (Resultados Reales)

Escenario	Lectura Excel	Subida DB	TOTAL
A. Java Puro	31.08s	1.90s	32.98s
B. Python Directo	1.60s	~1.0s	2.59s
C. Hibrido (Java→Python)	1.58s	~0.5s	2.86s

3.4 Analisis de Mejoras

Comparacion	Factor de Mejora
Java Puro vs Hibrido	11.5x mas rapido
Lectura Excel (Java vs Python)	19.7x mas rapido
Overhead del Hibrido vs Python Directo	Solo +0.27s (+10%)

**Conclusion del Benchmark:** El enfoque hibrido es **11.5 veces mas rapido** que Java puro, con un overhead minimo de 270ms por la invocacion desde Java.

4. Analisis Tecnico

4.1 Causa de la Diferencia de Rendimiento

Lectura de Excel

Aspecto	Java (Apache POI)	Python (Polars)
Implementacion	Java puro, orientado a objetos	Rust compilado, bindings Python
Modelo de memoria	Carga completa en heap JVM	Streaming con memoria optimizada
Paralelizacion	Single-threaded	Multi-threaded nativo
Optimizacion	Proposito general	Optimizado para datos tabulares

Apache POI es una libreria madura pero disenada para manipulacion general de documentos Office, no para procesamiento de alto rendimiento de datos.

Polars esta construido sobre Rust y utiliza optimizaciones de bajo nivel (SIMD, paralelismo, zero-copy) especificas para operaciones de datos.

Escritura a Base de Datos

Aspecto	Java (JDBC)	Python (ADBC)
---------	-------------	---------------

Protocolo	JDBC estandar	Arrow Database Connectivity
Transferencia	Serializacion fila por fila	Columnar en bloques
Overhead	Mayor por abstraccion	Minimo, datos en formato nativo

ADBC (Arrow Database Connectivity) transfiere datos en formato columnar Apache Arrow, evitando conversiones intermedias.

## 5. Analisis de Opciones para Produccion

### 5.1 Opcion A: Java Puro

#### Ventajas

- Integracion nativa con el ERP (mismo stack tecnologico)
- Un solo lenguaje a mantener
- Sin dependencias externas al ecosistema Java
- Despliegue simplificado (un solo JAR)

#### Desventajas

- Rendimiento significativamente inferior (31.5s vs 3.6s)
- Apache POI consume mucha memoria para archivos grandes
- Complejidad para optimizar (requiere reescritura significativa)

#### Riesgos

- Timeouts en operaciones si los archivos crecen
- Presion de memoria en el servidor

#### Costo estimado de optimizacion

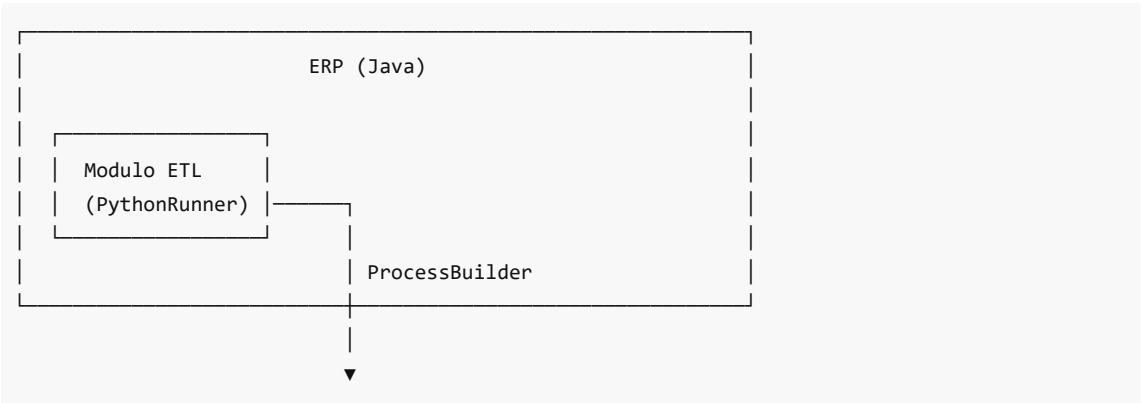
Para igualar el rendimiento de Python seria necesario:

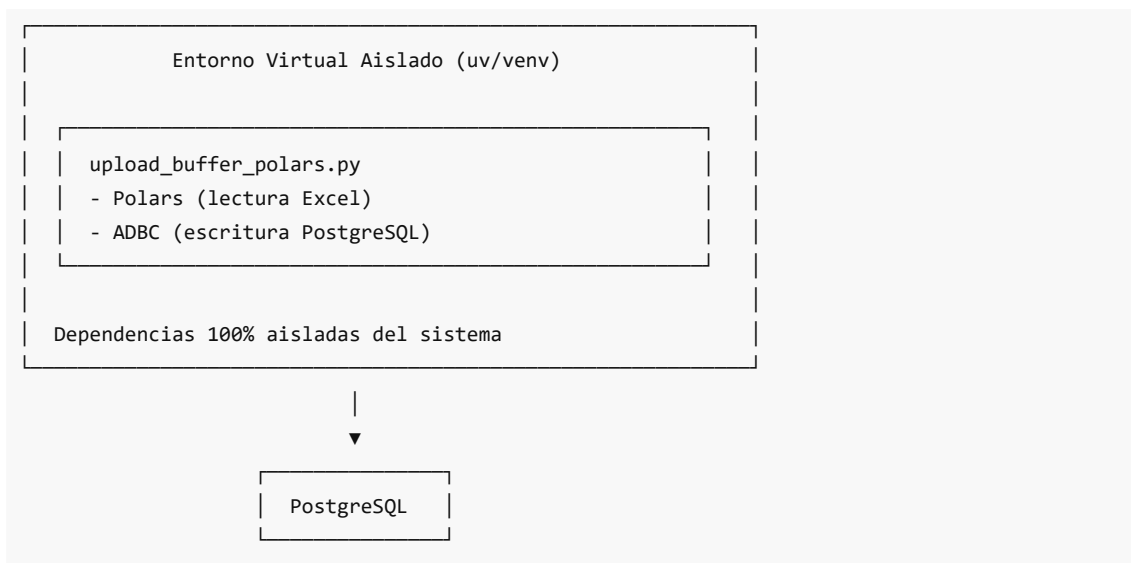
- Evaluar librerias alternativas (FastExcel, StreamingReader)
- Implementar procesamiento paralelo manual
- Posiblemente reescribir logica de insercion con COPY command

**Estimacion: 40-60 horas de desarrollo adicional, sin garantia de igualar rendimiento.**

### 5.2 Opcion B: Hibrido con Python en Entorno Virtual Aislado

#### Arquitectura Propuesta





### Ventajas

- Rendimiento optimo (3.6 segundos)
- **Aislamiento total:** Las librerias no afectan ni son afectadas por el sistema
- **Cero conflictos:** Cada proyecto puede tener su propio entorno
- Mantenimiento sencillo (editar script Python directamente)
- Facil actualizacion de dependencias
- Debugging simple con logs

### Desventajas

- Requiere Python instalado en el servidor (solo el interprete base)
- Dos lenguajes a mantener

### Por que el Entorno Virtual Evita Conflictos

Problema Comun	Solucion con venv
"Ya tengo otra version de X instalada"	Cada venv tiene sus propias versiones
"Actualizar libreria rompe otro proyecto"	Los venvs son independientes
"No tengo permisos para instalar global"	venv no requiere permisos admin
"El sistema usa Python 3.8, necesito 3.11"	Cada venv puede usar diferente Python

## 6. Gestion del Entorno Virtual: uv vs pip tradicional

### 6.1 Que es uv

**uv** es un gestor de paquetes Python moderno creado por Astral (los mismos de Ruff). Esta escrito en Rust y es significativamente mas rapido que pip.

### 6.2 Comparativa uv vs pip

Aspecto	pip + venv tradicional	uv
Velocidad instalacion	~30 segundos	~2 segundos

Resolucion dependencias	Lenta, a veces inconsistente	Rapida, deterministica
Archivo de lock	No nativo	uv.lock integrado
Compatibilidad Windows	Si	<b>Si</b>
Creacion de venv	python -m venv	uv venv
Reproducibilidad	Media	Alta

### 6.3 Instalacion de uv en Windows

```
# Opcion 1: Con PowerShell (recomendado)
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"

# Opcion 2: Con pip (si ya tienes Python)
pip install uv
```

### 6.4 Comandos Basicos uv

```
# Crear entorno virtual
uv venv C:\ERP\python-env

# Activar (Windows)
C:\ERP\python-env\Scripts\activate

# Instalar dependencias (10-100x mas rapido que pip)
uv pip install polars sqlalchemy psycopg2-binary openpyxl

# O desde archivo requirements
uv pip install -r requirements.txt

# Generar lock file (reproducibilidad exacta)
uv pip compile requirements.txt -o requirements.lock
uv pip sync requirements.lock
```

### 6.5 Recomendacion

Usar uv para el servidor de produccion por:

- 1. Instalacion 10-100x mas rapida
- 2. Resolucion de dependencias deterministica (evita "funciona en mi maquina")
- 3. Archivo lock para reproducibilidad exacta
- 4. Funciona perfectamente en Windows Server

## 7. Matriz de Decision

Criterio	Peso	Java Puro	Python Directo	Hibrido (Java→Python)
Rendimiento	30%	2	10	10

Integracion con ERP	25%	10	3	9
Complejidad despliegue	15%	10	7	8
Estabilidad	15%	9	9	9
Costo implementacion	15%	5	8	9
<b>TOTAL PONDERADO</b>	100%	<b>6.55</b>	<b>7.30</b>	<b>9.10</b>

**Puntuacion:** 1 (peor) a 10 (mejor)

**Nota:** Python Directo tiene mejor rendimiento pero menor integracion con el ERP Java existente. El enfoque Hibrido combina lo mejor de ambos mundos.

## 8. Recomendacion Final

### Opcion Recomendada: Hibrido con Python en Entorno Virtual (usando uv)

#### Justificacion Basada en Pruebas Reales

- 1. Rendimiento Comprobado:** La diferencia de **30 segundos** por operacion (32.98s vs 2.86s) fue validada en pruebas reales con datos de produccion (37,879 filas).
- 2. Overhead Minimo:** El costo de invocar Python desde Java es solo **0.27 segundos** (+10%), practicamente despreciable.
- 3. Integracion Nativa:** Java mantiene el control del flujo y puede integrarse facilmente con el ERP existente.
- 4. Aislamiento Total:** El entorno virtual (venv) garantiza que las dependencias estan completamente aisladas. No hay conflictos con el sistema.
- 5. Reproducibilidad:** Con uv, el entorno se recrea en segundos de forma identica en cualquier servidor.
- 6. Bajo Riesgo:** La implementacion es sencilla y 100% reversible.

## 9. Plan de Implementacion

### 9.1 Paso a Paso para Servidor Windows

```
# =====
# PASO 1: Instalar uv (una sola vez)
# =====
powershell -c "irm https://astral.sh/uv/install.ps1 | iex"

# Verificar instalacion
uv --version

# =====
# PASO 2: Crear estructura de carpetas
# =====
mkdir C:\ERP\etl
mkdir C:\ERP\etl\scripts
```

```
mkdir C:\ERP\etl\logs

# =====
# PASO 3: Crear entorno virtual aislado
# =====
uv venv C:\ERP\etl\venv

# =====
# PASO 4: Instalar dependencias
# =====
C:\ERP\etl\venv\Scripts\activate

uv pip install polars==0.20.0
uv pip install sqlalchemy==2.0.23
uv pip install psycpg2-binary==2.9.9
uv pip install openpyxl==3.1.2
uv pip install adbc-driver-postgresql==0.8.0

# Verificar instalacion
python -c "import polars; print('Polars OK:', polars.__version__)"

# =====
# PASO 5: Copiar script
# =====
copy "\\ruta\al\upload_buffer_polars.py" "C:\ERP\etl\scripts\"

# =====
# PASO 6: Probar ejecucion
# =====
C:\ERP\etl\venv\Scripts\python.exe C:\ERP\etl\scripts\upload_buffer_polars.py
```

9.2 Configuracion en PythonRunner.java

```
// Rutas de produccion
private static final String PYTHON_VENV = "C:\\ERP\\etl\\venv\\Scripts\\python.exe";
private static final String PYTHON_SCRIPT =
"C:\\ERP\\etl\\scripts\\upload_buffer_polars.py";
```

9.3 Tiempos Estimados

Fase	Actividad	Duracion
1	Instalar uv	5 min
2	Crear venv e instalar dependencias	10 min
3	Copiar y configurar scripts	15 min
4	Ajustar PythonRunner.java	30 min
5	Pruebas de integracion	2 horas

6	Documentacion	1 hora
<b>Total</b>		<b>~4 horas</b>

## 10. Mantenimiento Futuro

### 10.1 Actualizar Dependencias

```
# Activar entorno
C:\ERP\etl\venv\Scripts\activate

# Actualizar una libreria especifica
uv pip install polars --upgrade

# O reinstalar todo
uv pip install -r requirements.txt --upgrade
```

### 10.2 Recrear Entorno (si hay problemas)

```
# Eliminar entorno corrupto
Remove-Item -Recurse -Force C:\ERP\etl\venv

# Recrear desde cero
uv venv C:\ERP\etl\venv
C:\ERP\etl\venv\Scripts\activate
uv pip install -r requirements.txt
```

### 10.3 Backup del Entorno

```
# Exportar dependencias exactas
uv pip freeze > C:\ERP\etl\requirements-frozen.txt

# Este archivo permite recrear el entorno identico en otro servidor
```

## 11. Conclusion

Las pruebas reales demuestran que para el caso de uso especifico (carga masiva de Excel a PostgreSQL con 37,879 filas), el enfoque hibrido Java + Python ofrece:

Metrica	Resultado
Mejora de rendimiento	<b>11.5x mas rapido</b>
Tiempo Java Puro	32.98 segundos
Tiempo Hibrido	2.86 segundos

Ahorro por operacion	<b>30.12 segundos</b>
Overhead de integracion	Solo 0.27 segundos

El uso de **entorno virtual con uv** garantiza:

- **Cero conflictos** con el sistema o otros proyectos
- **Instalacion rapida** (10-100x mas que pip)
- **Reproducibilidad** exacta del entorno
- **Compatibilidad total** con Windows Server

La implementacion puede completarse en aproximadamente **4 horas** de trabajo, con riesgo bajo y alta reversibilidad.

## Recomendacion Final

**Implementar el enfoque Hibrido (Java → Python con venv)** para los procesos ETL del ERP. El rendimiento 11.5x superior justifica ampliamente el esfuerzo minimo de configuracion.

## Anexos

### A. requirements.txt

```
polars==0.20.0
sqlalchemy==2.0.23
psycopg2-binary==2.9.9
openpyxl==3.1.2
adbc-driver-postgresql==0.8.0
```

### B. Estructura Final en Servidor

```
C:\ERP\etl\
├─ venv\                # Entorno virtual aislado
│  └─ Scripts\
│     └─ python.exe     # Interprete Python aislado
│     └─ pip.exe
│     └─ activate.bat
│     └─ Lib\
│        └─ site-packages\ # Librerias instaladas (aisladas)
├─ scripts\
│  └─ upload_buffer_polars.py
├─ logs\
└─ requirements.txt
```

### C. Verificacion de Aislamiento

```
# Python del sistema (si existe)
where python
# Salida: C:\Python311\python.exe

# Python del venv (completamente separado)
```

```
C:\ERP\etl\venv\Scripts\python.exe -c "import sys; print(sys.prefix)"
```

```
# Salida: C:\ERP\etl\venv
```

```
# Las librerías del venv NO afectan al sistema
```

```
# Las librerías del sistema NO afectan al venv
```

---

*Fin del documento*