

Tutorial de DLT

¿Por qué `dlt`?

- Pipeline ETL básico: Mover datos de A a B
- La mayoría del código es repetitivo
- Patrones de carga conocidos
- No hay muchos frameworks de código abierto disponibles
 - Airbyte: Funciona a través de UI y archivos YAML, necesita un servidor dedicado
 - Meltano: Funciona a través de CLI y archivos YAML, no es fácil de extender
 - `dlt`: Funciona a través de código Python

¿Por qué este tutorial?

- Ya existen muchos tutoriales
- No me gustan *algunas* cosas sobre `dlt`
- Contras
 - Las explicaciones sobre **conceptos centrales** no son consistentes
 - **El estilo de código** en los ejemplos *no* es consistente
 - La API se siente caótica a veces
- Pros
 - Se siente como un framework ligero que puede reemplazar mucho código repetitivo
 - Una vez que superas la curva de aprendizaje inicial, es fácil de usar

A man with a beard and mustache, wearing a grey t-shirt, stands in a workshop filled with pipes and tools. He has his hands behind his head and is looking directly at the camera with a neutral expression.

**Yeah, well you know,
that's just like your opinion, man.**

Hoja de ruta

- Comenzamos con un ejemplo simple que carga datos desde un generador a una base de datos `duckdb`
- Usaremos `postgres` como destino en algún momento, para ilustrar patrones de carga incremental y cómo configurar credenciales
- Exploraremos diferentes patrones de carga: `replace` , `append` , `merge` con `upsert` y `scd2`
- Exploraremos definiciones de esquema y contratos de datos
- Exploraremos elementos internos de `dlt` y cómo depurar pipelines
- Terminaremos con próximos pasos y recomendaciones

Parte 0: Primeros pasos

Hay **tres o cuatro** rutas para instalar lo necesario para comenzar con este tutorial:

1. Instalar python, duckdb y usar una base de datos Postgres externa de Neon (o tu proveedor preferido)
2. Si tienes `docker` y `docker compose` instalados, usa `docker compose` para configurar una base de datos Postgres local. Instala python y duckdb normalmente.
3. Si tienes `docker` y `docker compose` instalados, usa `devcontainers` para configurar todo por ti
4. Más simple: Haz fork de este repo y usa GitHub Codespaces

Consulta la parte de **Primeros Pasos** en la [guía para instrucciones detalladas sobre cómo configurar tu entorno](#)



Ejemplo simple usando duckdb

- En su forma más básica, `dlt` toma un iterable que produce diccionarios
- Y trata cada diccionario como una fila y lo inserta en un `destino`

```
from uuid import uuid4
from typing import Generator
import string

def sample_data() -> Generator[dict, None, None]:
    for x in range(2):
        yield {
            "id": x,
            "name": "Mr. " + string.ascii_letters[x],
            "random_field": uuid4(),
        }
```

Si consumimos este generador, obtenemos algo como

```
>>> list(my_sample_data)
[{'id': 0,
 'name': 'Mr. a',
 'random_field': UUID('e9564071-54ff-4574-9726-be83e92e6240'),
 {'id': 1,
 'name': 'Mr. b',
 'random_field': UUID('8fcd6e92-ea0b-4bd3-9d61-2c248963d7e1')}]
```

`dlt` toma este generador como argumento de `dlt.pipeline` y hace su trabajo

```
import dlt

pipeline = dlt.pipeline(
    pipeline_name="sample_pipeline",
    destination=dlt.destinations.duckdb,
    dataset_name="sample_data",
)

load_info = pipeline.run(
    sample_data,
    table_name="samples",
    write_disposition={
        "disposition": "replace",
    },
)
```

⚠️ los datos pasados a `dlt.pipeline` no se limitan a generadores. Podemos pasar una lista de diccionarios. Usamos un generador por eficiencia.

```
sample_data = [
    {"id": 0, "name": "Mr. a", "random_field": uuid4()},
    {"id": 1, "name": "Mr. b", "random_field": uuid4()},
]

load_info = pipeline.run(
    sample_data,
    table_name="samples",
    write_disposition={
        "disposition": "replace",
    },
)
```

Por ejemplo, cuando trabajamos con dataframes,

```
import pandas as pd
def my_pandas_data_source():
    df = pd.read_csv("your_custom_data.csv")
    for _, row in df.iterrows():
        yield row.to_dict()
```

dlt funciona mejor si el generador produce diccionarios en lotes, ver

<https://dltlib.com/docs/reference/performance#yield-pages-instead-of-rows>

Parte 1: Un pipeline de ejemplo usando DuckDB

1. IR AL TUTORIAL
2. EJECUTAR `python dlt_tutorial/1_sample_pipeline_basic.py`
3. VERIFICAR CONTENIDOS DE DUCKDB CON `sample_pipeline.duckdb -c "select * from sample_data.samples;"`
4. ¡Eso es todo! Estos son los bloques de construcción para lo que cubriremos a continuación

¿Qué acaba de pasar?

1. Definimos un `generador` que se alimenta de una lista de diccionarios
2. Este generador fue consumido por un `dlt.pipeline`. ¡También puede ser una *lista simple de diccionarios!*
3. Se llamó a `pipeline.run`, activando la ejecución del pipeline
4. `dlt` crea una `tabla` en una base de datos `duckdb`, llamada `sample_data.samples`

Sin que lo sepas,

1. `dlt` crea tres tablas internas más que persisten el estado del pipeline
2. `dlt` también creó un directorio en `~/.dlt/pipelines/sample_pipeline`

¿Qué pasa cuando se llama a `pipeline.run`?

- Extraer datos
 - Genera un `paquete de carga` con un `load_id` único (`_dlt_load_id`)
[\(docs\)](#)
- Normalizar
 - Infiere el esquema de los datos y lo evoluciona si es necesario
 - Desanida cualquier estructura anidada
 - Aplica contratos de esquema si están presentes
- Cargar
 - Ejecuta migraciones de esquema si es necesario
 - Inserta datos según el `write_disposition` y `write_strategy` (más sobre eso después)
 - Actualiza el estado del pipeline y las tablas internas de dlt

Usando resources y sources

- Un **resource** es una función (opcionalmente `async`) que **produce datos**. Para crear un resource, agregamos el decorador `@dlt.resource` a esa función.
- Un **source** es una función decorada con `@dlt.source` que devuelve uno o más resources.
- EJECUTAR `python dlt_tutorial/2_sample_pipeline_sources_resources.py`

```
+@dlt.resource
def sample_data() -> Generator[dict, None, None]:
    for x in range(2):
        yield {
            }

+@dlt.source
+def sample_source():
+    yield sample_data
+
+
if __name__ == "__main__":
    pipeline = dlt.pipeline(
        pipeline_name="sample_pipeline",
    )

    load_info = pipeline.run(
-        sample_data,
+        sample_source(),
        table_name="samples",
        write_disposition={
            "disposition": "replace",
```

- `dlt` genera automáticamente **especificaciones** de configuración para funciones decoradas con `@dlt.source`, `@dlt.resource`, y `@dlt.destination`
- Por ejemplo, **inyectando valores de configuración**

Injectando valores de configuración

```
@dlt.source
def sample_source(my_custom_parameter: str = "foo"):
    print(f"Custom parameter value: {my_custom_parameter}")
    yield sample_data
```

```
SAMPLE_PIPELINE__MY_CUSTOM_PARAMETER=bar python dlt_tutorial/2b_sample_pipeline_sources_resources_with_config.py
```

0

```
# .dlt/config.toml
[sample_pipeline]
my_custom_parameter = "bar"
```

Consulta la [documentación](#) para una especificación completa sobre precedencia de variables.

Parte 2: Pipeline incremental con postgres



Estado actual

id	name	created_at	updated_at
1	Mr. Mario	2025-10-09 14:40:00	2025-10-09 14:50:00
2	Mr. Luigi	2025-10-08 16:15:00	2025-10-08 16:50:00

Estado entrante

id	name	created_at	updated_at
1	Jumpman	2025-10-09 14:40:00	2025-10-10 11:50:00
3	Ms. Peach	2025-10-12 13:15:00	2025-10-13 13:50:00

¿Cómo deberíamos actualizar nuestra tabla?



```
[ WITH with_query [, ...] ]
MERGE INTO [ ONLY ] target_table_name [ * ] [ [ AS ] target_alias ]
    USING data_source ON join_condition
    when_clause [...]
    [ RETURNING [ WITH ( { OLD | NEW } AS output_alias [, ...] ) ]
        { * | output_expression [ [ AS ] output_name ] } [, ...] ]
```

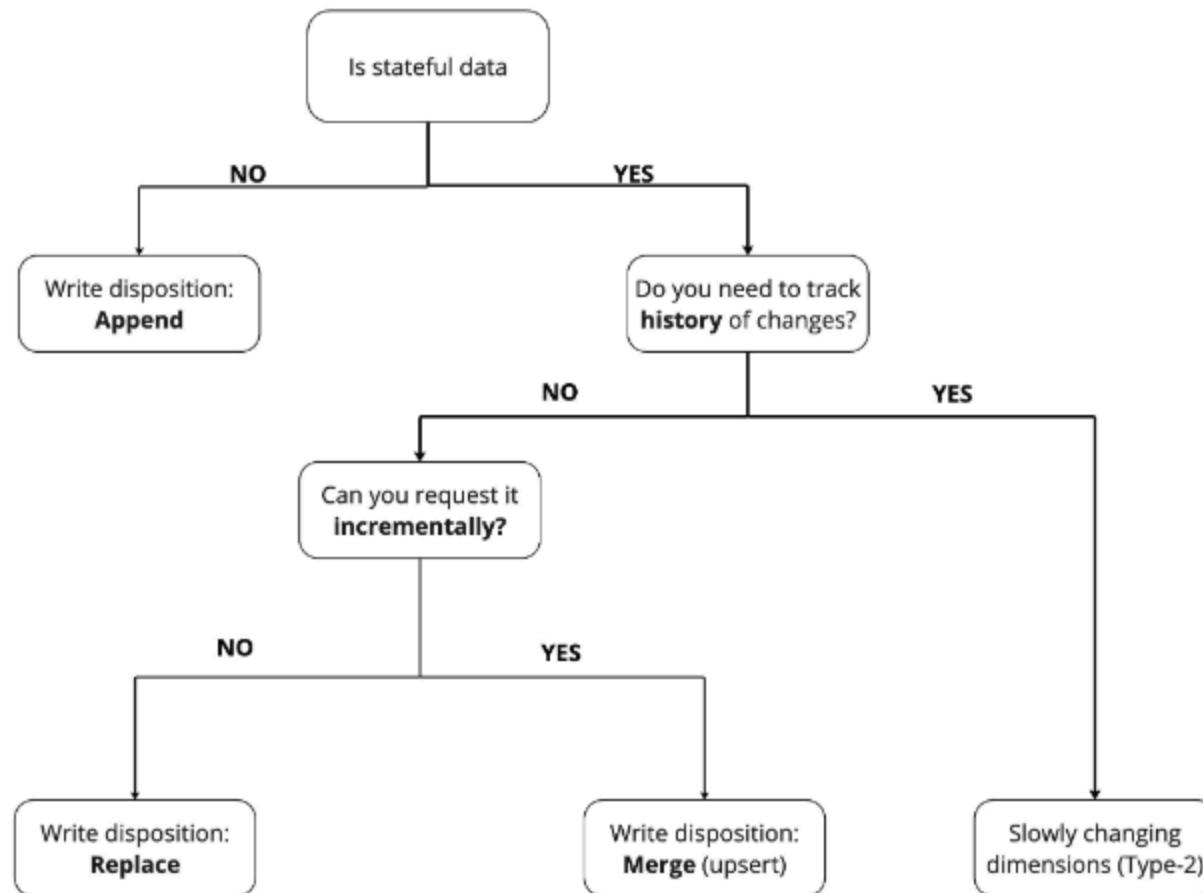
Disposiciones de escritura

1. Write disposition = "replace": Reemplazar la tabla completamente
2. Write disposition = "append": Cargar filas incrementalmente, independientemente de sus valores
3. Write disposition = "merge": Insertar solo filas que son relevantes para la actualización, ej. que *coinciden* con datos en el destino según alguna **estrategia** y alguna **clave** que permite identificar filas coincidentes.

Para la disposición "merge"

1. "delete-insert"
2. "upsert"
3. "scd2"

dlt resume cada caso de uso con el siguiente diagrama:



Configurando postgres

Configurando credenciales Definiendo archivos TOML

```
mkdir .dlt
touch .dlt/config.toml
touch .dlt/secrets.toml
```

Actualiza estos archivos como

```
# .dlt/secrets.toml
+[sample_pipeline.destination.postgres.credentials]
+host = "localhost"
+port = 5555
+username = "postgres"
+password = "test"
+database = "postgres"
```

0

```
# .dlt/config.toml
sample_pipeline_postgres.destination.postgres.credentials = 'postgresql://postgres:test@localhost:5555/postgres'
```

Usando la disposición "replace"

```
@dlt.resource(
    write_disposition={
        "disposition": "replace",
    },
)
...
    pipeline = dlt.pipeline(
-        pipeline_name="sample_pipeline",
+        pipeline_name="sample_pipeline_postgres",
-        destination=dlt.destinations.duckdb,
+        destination=dlt.destinations.postgres,
        dataset_name="sample_data",
    )
```

```
python dlt_tutorial/3_sample_pipeline_postgres_config.py
```

Usando la disposición "append"

```
+     primary_key="id",
write_disposition=
-
-         "disposition": "replace",
+         "disposition": "append",
},
```

```
python dlt_tutorial/4_sample_pipeline_append.py --refresh
```

Ejecútalo otra vez para ver cómo se agregan los datos

```
python dlt_tutorial/4_sample_pipeline_append.py
```

Usando "merge" y "upsert"

```
@dlt.resource(  
+    primary_key="id",  
    write_disposition={  
-        "disposition": "append",  
+        "disposition": "merge",  
+        "strategy": "upsert",  
    },
```

```
python dlt_tutorial/5_sample_pipeline_merge_upsert.py --refresh
```

Ejecútalo otra vez para ver cómo se fusionan los datos

```
python dlt_tutorial/5_sample_pipeline_merge_upsert.py
```

Ejecútalo otra vez con nuevos datos entrantes para ver cómo se fusionan los datos

```
USE_NEW_DATA=1 python dlt_tutorial/5_sample_pipeline_merge_upsert.py --refresh
```

Usando scd2

```
+     primary_key="id",
      write_disposition=
+         "disposition": "merge",
-         "strategy": "upsert",
+         "strategy": "scd2",
      },
```

```
python dlt_tutorial/6_sample_pipeline_merge_scd2.py --refresh
```

Ejecútalo otra vez para ver cómo se fusionan los datos

```
python dlt_tutorial/6_sample_pipeline_merge_scd2.py
```

Ejecútalo otra vez con nuevos datos entrantes para ver cómo se fusionan los datos

```
USE_NEW_DATA=1 python dlt_tutorial/6_sample_pipeline_merge_scd2.py
```

Validación de esquemas de datos



- El esquema describe la **estructura** de datos normalizados
- `dlt` genera esquemas de los datos durante el proceso de normalización.
- Los usuarios pueden afectar este comportamiento estándar proporcionando **pistas**
- `dlt` asocia un esquema con un **source** y un esquema de tabla con un **resource**.

En resumen, `dlt` **inferirá** el esquema pero podemos forzarlo a tipos explícitos

1. Usando una especificación `dict`
2. Usando un modelo `Pydantic`

Usando un dict

```
@dlt.resource(
    name="sample_data",
    primary_key="id",
    write_disposition="replace",
+    columns={
+        "id": {"data_type": "bigint"},
+        "name": {"data_type": "text"},
+        "uuid": {"data_type": "text"},
+        "created_at": {"data_type": "timestamp"},
+        "updated_at": {"data_type": "timestamp"},
+    },
```

Usando un modelo Pydantic

```
+class SampleDataModel(BaseModel):
+    id: int
+    name: str
+    uuid: UUID
+    created_at: dt.datetime
+    updated_at: dt.datetime
+    metadata: SampleDataMetadataModel

@dlt.resource(
    name="sample_data",
    primary_key="id",
    write_disposition="replace",
+    columns=SampleDataModel,
```

Contratos de datos

`dlt` maneja cambios en tablas, columnas y tipos de datos por defecto. Puedes configurar su comportamiento explícitamente pasando valores al argumento `schema_contract` del decorador `dlt.resource`, como:

```
@dlt.resource(  
    schema_contract={  
        "tables": "evolve",  
        "columns": "freeze",  
        "data_type": "freeze",  
    })  
def my_resource():  
    ...
```

Otros trucos y recomendaciones



- Pasa `PROGRESS=log|tqdm|enlighten` python `script.py` para cambiar el estilo de la barra de progreso ([fuente](#))
- Mantente en un `pipeline.run` por script.
- Usa `dlt` para ELT, no para ETL. Transforma datos lo más cerca posible de la fuente.
- Envía la salida de `pipeline.run` a otra llamada a `pipeline.run` para persistir logs de ejecución al mismo destino
- Pasa `dev_mode=True` a `dlt.pipeline` para iteración rápida ([fuente](#))
- Separa entornos dev/prod usando `pipelines_dir` ([fuente](#))
- `dlt` soporta ejecución paralela a través de `concurrent.futures` ([fuente](#))
- Envía errores a sentry configurando `runtime.sentry_dsn="https://<...>"` en `config.toml` ([fuente](#))

Campos anidados

`dlt` es capaz de serializar desde tablas *anidadas*, y generará tablas para cada estructura anidada (ver [docs](#)). Por ejemplo:

```
data = [
  {
    "id": 1,
    "name": "Alice",
    "pets": [
      {"id": 1, "name": "Fluffy", "type": "cat"},
      {"id": 2, "name": "Spot", "type": "dog"},
    ],
  },
  {"id": 2, "name": "Bob", "pets": [{"id": 3, "name": "Fido", "type": "dog"}]},
]
```

Ejecutar un pipeline con `dataset_name='mydata'` y `table_name='users'` producirá `mydata.users`

<code>id</code>	<code>name</code>	<code>_dlt_id</code>	<code>_dlt_load_id</code>
1	Alice	wX3f5vn801W16A	1234562350.98417
2	Bob	rX8ybgTeEmAmmA	1234562350.98417

`mydata.users__pets`

<code>id</code>	<code>name</code>	<code>type</code>	<code>_dlt_id</code>	<code>_dlt_parent_id</code>	<code>_dlt_list_idx</code>
1	Fluffy	cat	w1n0PEDzuP3grw	wX3f5vn801W16A	0
2	Spot	dog	9uxh36VU9lqKpw	wX3f5vn801W16A	1
3	Fido	dog	pe3FVtCWz8VuNA	rX8ybgTeEmAmmA	0

¿Qué sigue?



Ahora deberías poder construir pipelines básicos usando `dlt`. Aquí hay algunas sugerencias sobre qué explorar a continuación:

Pregunta a la comunidad y el **dlthub bot** en Slack

Si tienes preguntas o necesitas ayuda, puedes unirte a la [comunidad DLT Slack](#) y hacer tus preguntas allí. La comunidad es muy activa y útil.

Usando la CLI

dlt proporciona una herramienta CLI para gestionar pipelines, y una UI para navegar datos visualmente. Ver más en los [docs](#)

```
dlt pipeline --list # muestra pipelines existentes  
dlt pipeline <pipeline_name> show # muestra detalles del pipeline usando streamlit  
dlt pipeline <pipeline_name> show --dashboard # muestra dashboard del pipeline usando marimo
```

¡Puedes consultar datos e inspeccionar los estados del pipeline, desde una interfaz streamlit!

Usando `dlt init`

`dlt` proporciona una herramienta CLI para inicializar un proyecto

```
dlt init rest_api duckdb
```

Esto generará el código mostrado antes y las dependencias necesarias.

1. `rest_api_pipeline.py`
2. `.dlt/` y archivos TOML
3. `requirements.txt`

¡Puedes probar esto con cualquier otra fuente y/o destino!

```
dlt init rest_api postgres
```

 ¡Ten cuidado de que el código generado puede ser a veces abrumador! Revisa el código generado antes de ejecutarlo.

No abrí con esto porque hace las cosas complicadas de inmediato.

Explora tutoriales y cursos más avanzados

Ver <https://dlthub.com/docs/tutorial/education> para más



¡Eso es todo amigos! Gracias :)