DLT tutorial

Why `dlt`

- Basic ETL pipeline: Move data from A to B
- Most code is repetitive
- Known loading patterns
- Not many open source frameworks available
    - Airbyte: Works through UI and through YAML files, needs a dedicated server
    - Meltano: Works through CLI and YAML files, not easy to extend
    - `dlt`: Works through Python code

Why this tutorial?

- Many tutorials exist already
- I don't like *some* things about `dlt`
- Cons
  - Explanations on **core concepts** is not consistent
  - **Code style** in examples is *not* consistent
  - API feels chaotic at times
- Pros
  - Feels like a lightweight framework that can replace a lot of boilerplate code
  - Once you get past the initial learning curve, it is easy to use


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

Roadmap

- We are starting with a simple example that loads data from a generator into a `duckdb` database
- We will use `postgres` as a target at some point, to show how to configure credentials
- We will explore different loading patterns: `replace`, `append`, `merge` with `upsert` and `scd2`
- We will explore schema definitions and data contracts
- We will explore `dlt` internals and how to debug pipelines
- We will finish with next steps and recommendations

Part 0: Getting started

There **are three** routes to install what's needed to get started with this tutorial:

1. Install python, duckdb and use an external Postgres database from Neon (or your preferred provider)
2. If you have `docker` and `docker compose` installed, use docker compose to set up a local Postgres database. Install python and duckdb normally.
3. If you have `docker` and `docker compose` installed, use `devcontainers` to set up everything for you

Check the Getting Started part in the guide for detailed instructions on how to set up your environment

Simple example using `duckdb`

- At its most basic form, `dlt` takes in an iterable that produces dictionaries
- And treats every dictionary as a row and inserts it into a `target`

```
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(),
        }
```

If we consume this generator, we get something like

```
>>> 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` takes this generator as an argument of `dlt.pipeline` and does its thing

```
import dlt

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

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

⚠️ data passed to `dlt.pipeline` is not limited to generators. We can pass a list of dictionaries. We use a generator for efficiency.

```
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",
 },
)
```

For example, when working with 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` works better if the generator yields dictionaries in batches, see

https://dlthub.com/docs/reference/performance#yield-pages-instead-of-rows

Part 1: A Sample pipeline using DuckDB

1. GO TO TUTORIAL
2. EXECUTE `python dlt_tutorial/1_sample_pipeline_basic.py`
3. CHECK DUCKDB CONTENTS WITH `sample_pipeline.duckdb -c "select * from sample_data.samples;"`
4. That's it! These are the building blocks for what we'll be covering next

What just happened?

1. We defined a `generator` that feeds from a list of dictionaries
2. This generator was consumed by a `dlt.pipeline`. It can be a plain *list of dictionaries too!*
3. `pipeline.run` was called, triggering the pipeline execution
4. `dlt` creates a `table` in a `duckdb` database, called `sample_data.samples`

Unknown to you,

1. `dlt` creates three more internal tables that persist the state of the pipeline
2. `dlt` also created a directory in `~/.dlt/pipelines/sample_pipeline`

What happens when `pipeline.run` is called?

- Extract data
  - Generates a `load package` with a unique `load_id` (`_dlt_load_id`) (docs)
- Normalize
  - Infers schema from data and evolves it if needed
  - Unnest any nested structures
  - Apply schema contracts if present
- Load
  - Runs schema migrations if needed
  - Insert data according to the `write_disposition` and `write_strategy` (more on that later)
  - Updates pipeline state and internal dlt tables

Using resources and sources

- A resource is an (optionally async) function that **yields data**. To create a resource, we add the `@dlt.resource` decorator to that function.
- A source is a function decorated with `@dlt.source` that returns one or more resources.
- At its most basic implementation, this looks like this

```
+@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` automatically generates configuration **specs** for functions decorated with `@dlt.source`, `@dlt.resource`, and `@dlt.destination`
- For example, **injecting configuration values**

Injecting configuration values

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

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

or

```
SAMPLE_PIPELINE__MY_CUSTOM_PARAMETER=bar python mypipeline.py
```

Check the documentation for a full spec on variable precedence.

Part 2: Incremental pipeline with postgres

Current state

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

Incoming state

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

Write dispositions

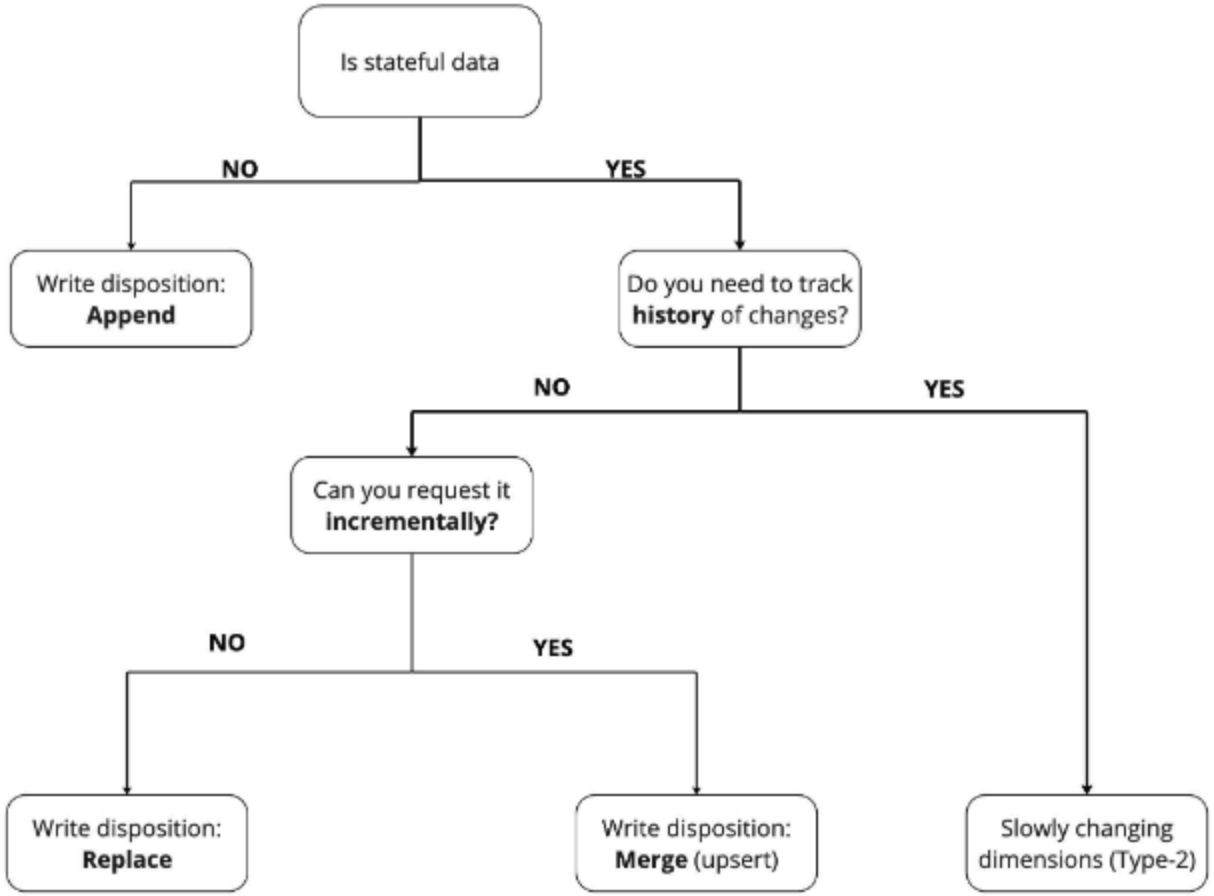
1. Write disposition = "replace": Replace the table entirely
2. Write disposition = "append": Load rows incrementally, regardless of their values
3. Write disposition = "merge": Insert only rows that are relevent to the update, e.g. that *match* data in target according to some **strategy** and some **key** that allows for identifying matching rows.

```
[ 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 ] } [, ...] ]
```

For the "merge" disposition

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

`dlt` summarizes every use case with the following diagram:



Setting up postgres

Configuring credentials Defining TOML files

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

Update these files as

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

Using the `"append"` disposition

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

Using "merge" and "upsert"

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

Using `scd2`

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

Data contracts

- The schema describes the **structure** of normalized data
- `dlt` generates schemas from the data during the normalization process.
- Users can affect this standard behavior by providing **hints**
- `dlt` associates a schema with a source and a table schema with a resource.

In short, `dlt` will **infer** the schema but we can force it to explicit types

1. Using a `dict` specification
2. Using a `Pydantic` model

Using a 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"},
+    },
```

Using a Pydantic model

```
+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,
```

Data contracts

`dlt` handles changes in tables, columns and data types by default. You can set its behaviour explicitly by passing values to the `schema_contract` argument of the `dlt.resource` decorator, such as:

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

Other tricks and recommendations

- Pass `PROGRESS=log|tqdm|enlighten python script.py` to change the progress bar style (source)
- Stick to one `pipeline.run` per script.
- Use `dlt` for ELT, not for ETL. Transform data as close to the source as possible.
- Send the output of `pipeline.run` to another call to `pipeline.run` to persist execution logs to the same target
- Pass `dev_mode=True` to `dlt.pipeline` for fast iteration (source)
- Separate dev/prod environments using `pipelines_dir` (source)
- `dlt` supports parallel execution through `concurrent.futures` (source)
- Send errors to sentry by setting `runtime.sentry_dsn="https:///<...>"` in `config.toml` (source)

Nested fields

`dlt` is able to serialize from *nested* tables, and it will generate tables for each nested structure (see docs). For example:

```
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"}]},
]
```

Running a pipeline with `dataset_name='mydata'` and `table_name='users'` Will produce

`mydata.users`

```
| id | name  |     _dlt_id      |   _dlt_load_id    |
|----|-------|------------------|-------------------|
| 1  | Alice | wX3f5vn801W16A   | 1234562350.98417  |
| 2  | Bob   | rX8ybgTeEmAmmA   | 1234562350.98417  |
```

`mydata.users__pets`

```
| id | name   | type |     _dlt_id      |   _dlt_parent_id   | _dlt_list_idx |
|----|--------|------|------------------|--------------------|---------------|
| 1  | Fluffy | cat  | w1n0PEDzuP3grw   | wX3f5vn801W16A     | 0             |
| 2  | Spot   | dog  | 9uxh36VU9lqKpw   | wX3f5vn801W16A     | 1             |
| 3  | Fido   | dog  | pe3FVtCWz8VuNA   | rX8ybgTeEmAmmA     | 0             |
```

What's next?

You should now be able to build basic pipelines using `dlt`. Here are some suggestions on what to explore next:

Ask the community and the `dlthub bot` on Slack

If you have questions or need help, you can join the DLT Slack community and ask your questions there. The community is very active and helpful.

Using the CLI

`dlt` provides a CLI tool to manage pipelines, and a UI to navigate data visually. See more in the docs

```
dlt pipeline --list # shows existing pipelines
dlt pipeline <pipeline_name> show # shows pipeline details using streamlit
dlt pipeline <pipeline_name> show --dashboard # shows pipeline dashboard using marimo
```

You can query data and inspect the pipeline states, from a streamlit interface!

Using `dlt init`

`dlt` provides a CLI tool to bootstrap a project

```
dlt init rest_api duckdb
```

This will generate the code shown before and dependencies needed.

1. `rest_api_pipeline.py`
2. `.dlt/` and TOML files
3. `requirements.txt`

You can try this with any other source and/or destination!

```
dlt init rest_api postgres
```

⚠️ Beware that the code generated can be at times overwhelming! Check the generated code before running it.

I did not open with this because it makes things complicated right away.

Explore more advanced tutorials and courses

See https://dlthub.com/docs/tutorial/education for more

That's all folks! Thank you :)