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 already exist
- But
  - Explanations on core concepts is not consistent
  - Code style in examples is not consistent
- I don't like *some* things about `dlt`: room for improvement!
  - Docs are very convoluted
  - Marketing is at times very aggressive (perhaps that changed)
  - Geared toward LLMware: I don't see the fit but I understand the trends.

MikeDoesEverything • hace 1 a

Shitty Data Engineer | 1% de más votados

One of the founders spams in here a ~~shitload~~. I'm sure they'll answer your question.

2 | Premiar | Compartir | ...

Thinker_Assignment • hace 1 a • Editado hace 1 a

1% de más votados

Hi Mike,

Thank you for sharing your thoughts. I appreciate your engagement and remember your feedback from last year, which I took seriously in my subsequent posts. Are your comments about my recent actions or from 1y ago? I am glad to discuss point items for improvement.

That being said this is a dev tool which I promote to devs, it's free and not something being sold to you, so I will be active here within reasonable boundaries.

You are welcome to block me and my posts will no longer bother you

3 | Premiar | Compartir | ...

MikeDoesEverything • hace 1 a

Shitty Data Engineer | 1% de más votados

> Are your comments about my recent actions or from 1y ago?

Let's be real - you know the answer to this. That being said, I'll humour you.

As a frequent (probably spend too much time here even) user of this sub, the fact I'd still consider you spamming this sub with promotion means you're probably spamming this sub. On the other

Zen of python

```
>>> import this
```

- **In the face of ambiguity, refuse the temptation to guess**
  - Configuration are portrayed as strings (Seems to be in line with llmware trends)
  - CLI does not help to reason about pipeline state. It needs some love, overall
- **There should be one-- and preferably only one --obvious way to do it**
  - Docs don't seem to agree in how something can be done. Full imports and namespaces are missing, etc.

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')}]
```

- three fields
- two rows
- static data + dynamic data

`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",
 },
)
```

after running this with `python 1_sample_pipeline_basic.py`, we should end up with a database called `sample_pipeline.duckdb`

⚠️ Beware that duckdb does not allow for concurrent access. If you try to run the pipeline while having a client connected to the database, the pipeline will fail.

```
$ duckdb sample_pipeline.duckdb "select * from information_schema.tables"
```

table_catalog varchar	table_schema varchar	table_name varchar	table_type varchar	…	user_defined_type_… varchar	is_insertable_into varchar	is_typed varchar	commit_action varchar	TABLE_COMMENT varchar
sample_pipeline	sample_data	samples	BASE TABLE	…	NULL	YES	NO	NULL	NULL
sample_pipeline	sample_data	_dlt_loads	BASE TABLE	…	NULL	YES	NO	NULL	NULL
sample_pipeline	sample_data	_dlt_pipeline_state	BASE TABLE	…	NULL	YES	NO	NULL	NULL
sample_pipeline	sample_data	_dlt_version	BASE TABLE	…	NULL	YES	NO	NULL	NULL
4 rows								13 columns (9 shown)	

```
$ duckdb sample_pipeline.duckdb "select * from sample_data.samples"
```

id int64	name varchar	random_field varchar	_dlt_load_id varchar	_dlt_id varchar
0	Mr. a	6e9644ca-be31-4bbc-8ac4-5775ac774fa5	1757350132.6024446	NDTqVLM79jz+DA
1	Mr. b	9c128d7b-5ecb-4db5-b9df-4b9bd89b71fb	1757350132.6024446	U9/QdL4eYctNOg

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. This `Pipeline` defines the name of the pipeline and the destination
4. This pipeline is *triggered* using some definitions
    i. write disposition is `replace`. This means that we drop the table every time the pipeline is run, and we recreate the `table` from scratch.
5. `dlt` creates a `table` in a `duckdb` database, called `sample_data.samples`
6. `dlt` creates three more tables that persist the state of the pipeline
7. Unkown to you, `dlt` also created a directory in `~/.dlt/pipelines/sample_pipeline`

About the source

it only needs to be a generator that yields dictionaries. 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

Where to go next?

- [x] decorate data generator with `dlt.resource` and `dlt.source`
- [x] implement configuration through files
- [x] modify loading patterns
- [x] transform before ingesting
- [x] validate incoming data using hints or pydantic
- [x] enforcing data contracts
- [x] using `dlt init`
- [x] exploring the pipeline state with the CLI and the UI

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

```
--- 1_sample_pipeline_basic.py 2025-09-08 14:32:11.274031300 -0300
+++ 2_sample_pipeline_sources_resources.py 2025-09-08 14:35:59.285421840 -0300
@@ -5,6 +5,7 @@
 import dlt

+@dlt.resource
 def sample_data() -> Generator[dict, None, None]:
     for x in range(2):
         yield {
@@ -14,6 +15,11 @@
         }


+@dlt.source
+def sample_source():
+    yield sample_data
+
+
 if __name__ == "__main__":
     pipeline = dlt.pipeline(
         pipeline_name="sample_pipeline",
@@ -22,7 +28,7 @@
     )

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

Alternatively, one can declare resources in nested form

```
--- 1_sample_pipeline_basic.py 2025-09-08 14:32:11.274031300 -0300
+++ 2_sample_pipeline_sources_resources.py 2025-09-08 14:35:59.285421840 -0300
@@ -5,6 +5,7 @@
 import dlt


+@dlt.source
+def sample_source():
+
+ @dlt.resource
+  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 __name__ == "__main__":
     pipeline = dlt.pipeline(
         pipeline_name="sample_pipeline",
@@ -22,7 +28,7 @@
     )

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

⚠️ `dlt` recommends setting the write disposition at the resource decorator

In practice this allows for poor parametrization, and I prefer setting the write disposition and other parameters at the `pipeline.run` call.

This is something I will form an opinion after I've used dlt enough though.

What changed?

- We decorated our generator with a `resource` decorator
- We yield the resource from a decorated `sample_source` method
- We pass the decorated `source` to the `dlt.pipeline` definition
- `dlt` automatically generates configuration **specs** for functions decorated with `@dlt.source`, `@dlt.resource`, and `@dlt.destination`
- For example, **injecting configuration values**

Injecting configuration values

- We are using a `duckdb` database as a target. What about credentials?
- `duckdb` by default won't prompt for credentials. What about `postgres`?
- Let's setup an example database using docker compose

```
--- /dev/null 2025-09-08 11:55:41.490139727 -0300
+++ docker-compose.yaml 2025-09-08 14:53:05.469092973 -0300
@@ -0,0 +1,22 @@
+services:
+  db:
+    # https://hub.docker.com/_/postgres/
+    image: "postgres:15"
+    restart: no
+    environment:
+      - POSTGRES_PASSWORD=test
+    volumes:
+      - pgdata:/var/lib/postgresql/data
+    ports:
+      - 5555:5432
+    networks:
+      db_nw:
+        aliases:
+          - postgres
+
+networks:
+  db_nw:
+    driver: bridge
+
+volumes:
+  pgdata:
```

launch this doing `docker compose up -d` and enter its `psql` client with `docker compose exec -it db psql -U postgres`

```
$ docker compose exec -it db psql -U postgres
psql (15.13 (Debian 15.13-1.pgdg130+1))
Type "help" for help.

postgres=#
```

to send anything to this database, we need to modify our current pipeline as

```
--- 2_sample_pipeline_sources_resources.py 2025-09-08 14:41:34.694398414 -0300
+++ 3_sample_pipeline_postgres_config.py 2025-09-08 14:57:18.657032675 -0300
@@ -23,7 +23,7 @@
 if __name__ == "__main__":
     pipeline = dlt.pipeline(
         pipeline_name="sample_pipeline",
-        destination=dlt.destinations.duckdb,
+        destination=dlt.destinations.postgres,
         dataset_name="sample_data",
     )
```

Running this will raise an exception though, because the postgres destination need some **credentials** to work. Fortunately, `dlt` will give you some pointers for what to do to fix it

```
python 3_sample_pipeline_postgres_config.py
...
<class 'dlt.common.configuration.exceptions.ConfigFieldMissingException'>
Missing fields in configuration: ['database', 'password', 'username', 'host'] PostgresCredentials
        for field `database` the following (config providers, keys) were tried in order:
                (Environment Variables, SAMPLE_PIPELINE__DESTINATION__POSTGRES__CREDENTIALS__DATABASE)
                (Environment Variables, SAMPLE_PIPELINE__DESTINATION__CREDENTIALS__DATABASE)
                (Environment Variables, SAMPLE_PIPELINE__CREDENTIALS__DATABASE)
                (Environment Variables, DESTINATION__POSTGRES__CREDENTIALS__DATABASE)
                (Environment Variables, DESTINATION__CREDENTIALS__DATABASE)
                (Environment Variables, CREDENTIALS__DATABASE)
...
Provider `secrets.toml` loaded values from locations:
        - /home/diego/Code/playground/dlt-tutorial/.dlt/secrets.toml
        - /home/diego/.dlt/secrets.toml
WARNING: provider `secrets.toml` is empty. Locations (i.e., files) are missing or empty.
Provider `config.toml` loaded values from locations:
        - /home/diego/Code/playground/dlt-tutorial/.dlt/config.toml
        - /home/diego/.dlt/config.toml
WARNING: provider `config.toml` is empty. Locations (i.e., files) are missing or empty.

Learn more: https://dlthub.com/docs/general-usage/credentials/
```

You have the following options

1. Define a `secrets.toml` and a `config.toml` to read configuration from. We'll do this next
2. Pass in environment variables
3. Define them at the `dlt.source` decorator level

Defining TOML files

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

Update these files as

```
--- /dev/null 2025-09-08 11:55:41.490139727 -0300
+++ .dlt/secrets.toml 2025-09-08 15:12:52.959972535 -0300
@@ -0,0 +1,6 @@
+[sample_pipeline.destination.postgres.credentials]
+host = "localhost"
+port = 5555
+username = "postgres"
+password = "test"
+database = "postgres"
```

and run the script again

```
python 3_sample_pipeline_postgres_config.py
```

`dlt` should be able to read secrets directly from this file and run the pipeline, this time on postgres.

```
docker compose exec -it db psql -U postgres -c "select * from sample_data.samples"
 id | name  |             random_field             |    _dlt_load_id    |    _dlt_id
----+-------+--------------------------------------+--------------------+---------------
  0 | Mr. a | db78147a-365f-4540-984a-a027d4b67cd1 | 1757355179.6326451 | GgHLD7ukCOqLnw
  1 | Mr. b | 16dbca89-f81b-4a77-ae88-a7f4f5e35607 | 1757355179.6326451 | adTvyoMKAVnX6Q
```

Using environment variables

The fields in the `toml` files have a direct transaction to environment variables, for example `sample_pipeline.destination.postgres.credentials.host = "localhost"` can be passed to the python command like

```
SAMPLE_PIPELINE__DESTINATION__POSTGRES__CREDENTIALS__HOST=localhost python 3_sample_pipeline_postgres_config.py
```

Check the documentation for a full spec on variable precedence.

Custom parameters

Say for example we want to pass in a custom parameter from either `config.toml` or `secrets.toml`.

```
--- 2_sample_pipeline_sources_resources.py 2025-09-08 14:41:34.694398414 -0300
+++ 3_sample_pipeline_postgres_config.py 2025-09-08 15:22:59.971186218 -0300
@@ -16,14 +16,15 @@


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

```
$ python 3_sample_pipeline_postgres_config.py
Custom parameter value: foo
...
```

We can now do something like

```
$ SAMPLE_PIPELINE__MY_CUSTOM_PARAMETER=bar python 3_sample_pipeline_postgres_config.py
Custom parameter value: bar
...
```

Or define it at the `config.toml` file as

```
[sample_pipeline]
my_custom_parameter = "baz"
```

```
python 3_sample_pipeline_postgres_config.py
Custom parameter value: baz
...
```

Modify loading patterns

We can load data in different ways, depending on the requirements. See the documentation for a full introduction

We can define the **write disposition** and the **write strategy** when **running a pipeline**.
We can choose from "replace", "append", "merge"

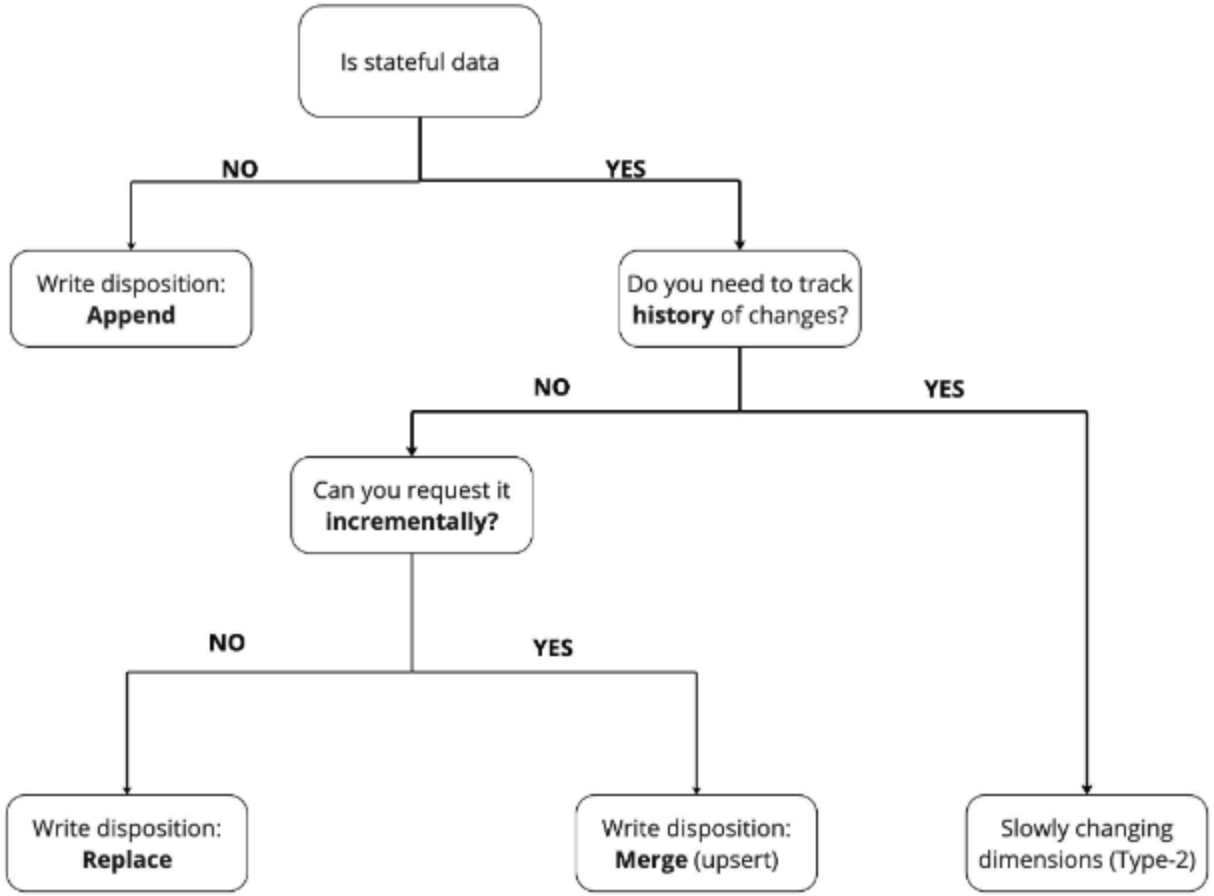
Write dispositions

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.

For the "merge" disposition

1. "delete-insert": given a **match** between incoming and existing rows on some **key**, **DELETE** the target row and **INSERT** it from incoming. This operation is **locking** or it may not be supported, depending on the target database.
2. "upsert": given a **match** between incoming and existing rows on some **key**, **UPDATE** the target row and "**INSERT**" what's changed from incoming data. This operation is not **locking** but it may not be supported, depending on the target database.
3. "scd2": given a **match** between incoming and existing rows on some **key**, leave the existing target row and "**INSERT**" a new one from the incoming data. Using some auxiliary columns, this allows for tracking the validity of the latest value, but it takes more space in disk.

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



full refresh

`refresh` *TRefreshMode, optional* - Fully or partially reset sources during pipeline run. When set here the refresh is applied on each run of the pipeline. To apply refresh only once you can pass it to `pipeline.run` or `extract` instead. The following refresh modes are supported:

- `drop_sources`: Drop tables and source and resource state for all sources currently being processed in run or extract methods of the pipeline. (Note: schema history is erased)
- `drop_resources`: Drop tables and resource state for all resources being processed. Source level state is not modified. (Note: schema history is erased)
- `drop_data`: Wipe all data and resource state for all resources being processed. Schema is not modified.

`dlt` allows for `refresh` through its `pipeline.run` method, but we can pass it as a CLI parameter to the pipeline run to allow for a full refresh using `argparse`

```
--- 3_sample_pipeline_postgres_config.py 2025-09-08 15:22:59.971186218 -0300
+++ 4_sample_pipeline_append.py 2025-09-08 15:56:13.613338195 -0300
@@ -1,8 +1,10 @@
+import argparse
 import string
 from typing import Generator
 from uuid import uuid4

 import dlt
+from dlt.pipeline import TRefreshMode


 @dlt.resource
@@ -21,18 +23,32 @@
     yield sample_data


+def parse_args():
+    parser = argparse.ArgumentParser(description="Sample DLT Pipeline with Append")
+    parser.add_argument(
+        "--refresh",
+        action="store_true",
+        help="Refresh the data in the destination (if applicable)",
+    )
+    return parser.parse_args()
+
+
 if __name__ == "__main__":
+    args = parse_args()
+    should_refresh = args.refresh
     pipeline = dlt.pipeline(
         pipeline_name="sample_pipeline",
         destination=dlt.destinations.postgres,
         dataset_name="sample_data",
     )

+    refresh_mode: TRefreshMode = "drop_sources"
     load_info = pipeline.run(
         sample_source,
         table_name="samples",
+        refresh=refresh_mode if should_refresh else None,
         write_disposition={
             "disposition": "replace",
         },
     )
```

Now we can do `python script.py --refresh` to force a full refresh.

Using the `"append"` disposition

```
--- 3_sample_pipeline_postgres_config.py 2025-09-08 15:22:59.971186218 -0300
+++ 4_sample_pipeline_append.py 2025-09-08 15:42:43.623577193 -0300
@@ -32,7 +32,7 @@
          sample_source,
          table_name="samples",
          write_disposition={
-             "disposition": "replace",
+             "disposition": "append",
          },
      )
```

run `python 4_sample_pipeline_append.py` a few times and check the result

```
❯ docker compose exec -it db psql -U postgres -c "select * from sample_data.samples"
 id | name  |             random_field             |    _dlt_load_id    |    _dlt_id
----+-------+--------------------------------------+--------------------+---------------
  0 | Mr. a | f40b3d6a-d539-45d2-959f-2a2ba9fa8e45 | 1757356952.281308  | Cip3HAdQFRIXWQ
  1 | Mr. b | bc20dfe8-d896-4d16-8a38-e21e94daefe7 | 1757356952.281308  | C4bMT88tQZY/vw
  0 | Mr. a | 2427f65e-ac86-4a25-a3b3-dfa5d36af1c6 | 1757356967.9445448 | ERDMbw1z7JZtqA
  1 | Mr. b | bc63e9fb-5f02-418f-ac24-4d6f6e8404f1 | 1757356967.9445448 | VmeyrIPpL2B9Aw
  0 | Mr. a | 517203af-b1e2-4963-9f38-ca26f9d78beb | 1757356982.8301303 | cxVSicIRfQVj8A
  1 | Mr. b | e5a30c70-3329-46ba-9a2a-abf72cc8123b | 1757356982.8301303 | d8ic9sSRqQp8aA
```

Examples for upsert

```
--- 4_sample_pipeline_append.py 2025-09-08 16:00:03.781448586 -0300
+++ 5_sample_pipeline_merge_upsert.py 2025-09-08 16:07:25.627308248 -0300
@@ -7,7 +7,9 @@
 from dlt.pipeline import TRefreshMode
-@dlt.resource
+@dlt.resource(
+    primary_key="id",
+)
 def sample_data() -> Generator[dict, None, None]:
     for x in range(2):
         yield {
@@ -48,7 +50,8 @@
         table_name="samples",
         refresh=refresh_mode if should_refresh else None,
         write_disposition={
-            "disposition": "append",
+            "disposition": "merge",
+            "strategy": "upsert",
         },
     )
```

Let's modify the second row outside of dlt to induce an UPSERT operation at the next pipeline run

```
$ docker compose exec -it db psql -U postgres -c "update sample_data.samples set name='myname' where id = 1"
UPDATE 1
```

```
$ docker compose exec -it db psql -U postgres -c "select * from sample_data.samples"
 id |  name  |             random_field             |   _dlt_load_id    |    _dlt_id
----+--------+--------------------------------------+-------------------+---------------
  0 | Mr. a  | 46086c3d-826f-45ca-9383-62fdf892fece | 1757358448.498475 | USG1q5Faq1m8TA
  1 | myname | 86c6ef1a-f546-4425-9e6a-d0e5c8ae37e8 | 1757358448.498475 | z5jIxXIbUnmbKw
(2 rows)
```

Run the pipeline again and check the results. dlt will update the row with `id = 1`

```
docker compose exec -it db psql -U postgres -c "select * from sample_data.samples"
 id | name  |              random_field              |   _dlt_load_id    |    _dlt_id
----+-------+----------------------------------------+-------------------+---------------
  0 | Mr. a | 35c4b477-25bf-4299-9275-417eb220669e | 1757358675.778588 | USG1q5Faq1m8TA
  1 | Mr. b | 8527b857-5a5f-4334-a85a-62c0ca487780 | 1757358675.778588 | z5jIxXIbUnmbKw
```

Using `scd2`

```
--- 5_sample_pipeline_merge_upsert.py 2025-09-08 16:07:25.627308248 -0300
+++ 6_sample_pipeline_merge_scd2.py 2025-09-08 16:12:25.570742251 -0300
@@ -51,7 +51,7 @@
         refresh=refresh_mode if should_refresh else None,
         write_disposition={
             "disposition": "merge",
-            "strategy": "upsert",
+            "strategy": "scd2",
         },
     )
```

After running `6_sample_pipeline_merge_scd2.py`, `dlt` will add two new columns to the target table: `_dlt_valid_from` and `_dlt_valid_to`.

The most recent and valid value for a **primary key** is set to null, whereas previous versions will have these values set to some timestamp

```
docker compose exec -it db psql -U postgres -c "select * from sample_data.samples"
 id | name  |             random_field             |    _dlt_load_id    |    _dlt_id     |        _dlt_valid_from        |         _dlt_valid_to
----+-------+--------------------------------------+--------------------+----------------+-------------------------------+-------------------------------
  0 | Mr. a | 35c4b477-25bf-4299-9275-417eb220669e | 1757358675.778588  | USG1q5Faq1m8TA |                               | 2025-09-08 19:12:39.131526+00
  1 | Mr. b | 8527b857-5a5f-4334-a85a-62c0ca487780 | 1757358675.778588  | z5jIxXIbUnmbKw |                               | 2025-09-08 19:12:39.131526+00
  0 | Mr. a | 43c88816-090f-4f52-ac9a-2ead8c34c9f3 | 1757358759.1315265 | Ir7Glr4PDwo6Iw | 2025-09-08 19:12:39.131526+00 |
  1 | Mr. b | 7454702a-27d0-49f2-905d-85b7c097490a | 1757358759.1315265 | 4jHI5UNRTpc9NQ | 2025-09-08 19:12:39.131526+00 |
```

Transforming data before insertion

For simplicity's sake, let's stick to the "replace" write disposition.

```
--- 6_sample_pipeline_merge_scd2.py 2025-09-08 16:12:25.570742251 -0300
+++ 7_sample_pipeline_transform_before.py 2025-09-09 12:46:51.975172678 -0300
@@ -50,8 +50,7 @@
          table_name="samples",
          refresh=refresh_mode if should_refresh else None,
          write_disposition={
-             "disposition": "merge",
-             "strategy": "scd2",
+             "disposition": "replace",
          },
      )
```

At its most simple version, a transformation takes in an iterable, and produces another iterable

```
--- 6_sample_pipeline_merge_scd2.py 2025-09-08 16:12:25.570742251 -0300
+++ 7_sample_pipeline_transform_before.py 2025-09-09 13:03:32.788219862 -0300
@@ -19,10 +19,19 @@
         }


+def transform_data(batch: list[dict]) -> Generator[dict, None, None]:
+    for data in batch:
+        data["my_transformed_field"] = (
+            data["name"].lower().replace(".", "_").replace(" ", "_")
+        )
+        yield data
+
+
 @dlt.source
 def sample_source(my_custom_parameter: str = "foo"):
     print(f"Custom parameter value: {my_custom_parameter}")
-    yield sample_data
+    transformed_resource = transform_data(sample_data)
+    yield transformed_resource
```

`dlt` exposes some abstractions facilitate **transforming** data

1. `add_map`
2. `dlt.transform`

`add_map`

in Python, one can use *functional programming* directives such as `map()`, `filter()` and `reduce()`. In particular

```
map(function, iterable[, iterable1, iterable2,..., iterableN])
```

accepts a *function* and *iterables*. Using these two arguments, one can do stuff like

```
>>> def square(number):
...     return number ** 2
...

>>> numbers = [1, 2, 3, 4, 5]

>>> squared = map(square, numbers)

>>> list(squared)
[1, 4, 9, 16, 25]
```

`add_map` **must operate on a dict level**, meaning that we can rewrite our previous transformation as

```
--- 7_sample_pipeline_transform_before.py 2025-09-09 13:03:32.788219862 -0300
+++ 8_sample_pipeline_transform_add_map.py 2025-09-09 13:23:17.945804988 -0300
@@ -19,19 +19,17 @@
         }
-def transform_data(batch: list[dict]) -> Generator[dict, None, None]:
-    for data in batch:
-        data["my_transformed_field"] = (
-            data["name"].lower().replace(".", "_").replace(" ", "_")
-        )
-        yield data
+def transform_data(record: dict) -> dict:
+    record["my_transformed_field"] = (
+        record["name"].lower().replace(".", "_").replace(" ", "_")
+    )
+    return record
 @dlt.source
 def sample_source(my_custom_parameter: str = "foo"):
     print(f"Custom parameter value: {my_custom_parameter}")
-    transformed_resource = transform_data(sample_data)
-    yield transformed_resource
+    yield sample_data.add_map(transform_data)
```

We can also **chain** mappings, for example if we want to remove a column from the record before insertion

```
--- 8_sample_pipeline_transform_add_map.py 2025-09-09 13:23:17.945804988 -0300
+++ 9_sample_pipeline_transform_remove_column.py 2025-09-09 13:22:55.972921512 -0300
+def remove_random_field(record: dict) -> dict:
+    del record["random_field"]
+    return record
+
+
 @dlt.source
 def sample_source(my_custom_parameter: str = "foo"):
     print(f"Custom parameter value: {my_custom_parameter}")
-    yield sample_data.add_map(transform_data)
+    yield sample_data.add_map(transform_data).add_map(remove_random_field)
```

Running `python 9_sample_pipeline_transform_remove_column.py --refresh` should return something like

```
$ docker compose exec -it db psql -U postgres -c "select * from sample_data.samples"
 id | name  | my_transformed_field |    _dlt_load_id    |    _dlt_id
----+-------+----------------------+--------------------+---------------
  0 | Mr. a | mr__a                | 1757434960.7269237 | aop7tOk+OK+u0w
  1 | Mr. b | mr__b                | 1757434960.7269237 | I3HsDBgDJgOFaA
```

Using `dlt.transform`

```
--- 8_sample_pipeline_transform_add_map.py 2025-09-09 13:23:17.945804988 -0300
+++ 10_sample_pipeline_transform_with_transformer.py 2025-09-09 13:37:31.077892625 -0300
@@ -19,17 +19,18 @@
         }


-def transform_data(record: dict) -> dict:
+@dlt.transformer(data_from=sample_data)
+def transform_data(record: dict) -> Generator[dict, None, None]:
     record["my_transformed_field"] = (
         record["name"].lower().replace(".", "_").replace(" ", "_")
     )
-    return record
+    yield record


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


 def parse_args():
```

🎈 when to use each?

`dlt` recommends using `add_map` for lightweight operations and `dlt.transform` for more complex tasks, see https://dlthub.com/docs/dlt-ecosystem/transformations/add-map#add_map-vs-dlttransformer

Other transformation approaches

I'm not covering these since they present more complex scenarios

- SQL using `with pipeline.sql_client as client: client.execute_sql("<your sql>")`, see the examples.
- dbt (see the example).
- Dataframes or Arrow tables (see the examples).

Setting a schema and validating data through contracts

Verbatim from the docs

- The schema describes the structure of normalized data (e.g., tables, columns, data types, etc.) and provides instructions on how the data should be processed and loaded.
- dlt generates schemas from the data during the normalization process. Users can affect this standard behavior by providing **hints** that change how tables, columns, and other metadata are generated and how the data is loaded.
- Such hints can be passed in the code, i.e., to the `dlt.resource` decorator or `pipeline.run` method. Schemas can also be exported and imported as files, which can be directly modified.
- `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
3. Using `hints`

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():
  ...
```

You can control the following **schema entities**:

- `tables` - the contract is applied when a new table is created
- `columns` - the contract is applied when a new column is created on an existing table
- `data_type` - the contract is applied when data cannot be coerced into a data type associated with an existing column.

You can use **contract modes** to tell `dlt` how to apply the contract for a particular entity:

- `evolve`: No constraints on schema changes.
- `freeze`: This will raise an exception if data is encountered that does not fit the existing schema, so no data will be loaded to the destination.
- `discard_row`: This will discard any extracted row if it does not adhere to the existing schema, and this row will not be loaded to the destination.
- `discard_value`: This will discard data in an extracted row that does not adhere to the existing schema, and the row will be loaded without this data.

🎈 How does "evolve" work?

The default mode (**evolve**) works as follows:

1. New tables may always be created.
2. New columns may always be appended to the existing table.
3. Data that do not coerce to the existing data type of a particular column will be sent to a variant column created for this particular type.

Using a dict specification

```
--- 3_sample_pipeline_postgres_config.py 2025-09-08 15:22:59.971186218 -0300
+++ 11_sample_pipeline_schema.py 2025-09-09 13:57:15.683130206 -0300
@@ -1,11 +1,20 @@

-@dlt.resource
+@dlt.resource(
+    primary_key="id",
+    columns={
+        "id": {"data_type": "int", "nullable": False},
+        "name": {"data_type": "text", "nullable": False},
+        "random_field": {"data_type": "text", "nullable": False},
+    },
+)
```

this will produce the following columns:

```
$ docker compose exec -it db psql -U postgres -c \
"select column_name, data_type
from information_schema.columns
where table_schema='sample_data'
and table_name='samples'"

 column_name  |     data_type
--------------+--------------------
 id           | bigint
 name         | character varying
 random_field | character varying
 _dlt_load_id | character varying
 _dlt_id      | character varying
```

Using pydantic

```
--- 11_sample_pipeline_schema.py 2025-09-09 14:09:46.263470595 -0300
+++ 12_sample_pipeline_schema_with_pydantic.py 2025-09-09 14:08:49.470184167 -0300
@@ -1,19 +1,22 @@
 import argparse
 import string
 from typing import Generator
-from uuid import uuid4
+from uuid import uuid4, UUID

 import dlt
 from dlt.pipeline import TRefreshMode
+from pydantic import BaseModel
+
+
+class SampleDataModel(BaseModel):
+    id: int
+    name: str
+    random_field: UUID


 @dlt.resource(
     primary_key="id",
-    columns={
-        "id": {"data_type": "int", "nullable": False},
-        "name": {"data_type": "text", "nullable": False},
-        "random_field": {"data_type": "text", "nullable": False},
-    },
+    columns=SampleDataModel,
 )
 def sample_data() -> Generator[dict, None, None]:
     for x in range(2):
```

Checking the schema is enforced

Try changing the type in the Pydantic model.

```
--- 12_sample_pipeline_schema_with_pydantic.py 2025-09-09 14:28:16.878957096 -0300
+++ 13_sample_pipeline_schema_breaks.py 2025-09-09 14:28:32.109031189 -0300
@@ -9,7 +9,7 @@


 class SampleDataModel(BaseModel):
-    id: int
+    id: str
     name: str
     random_field: UUID
```

Running `13_sample_pipeline_schema_breaks.py` should raise an error

```
dlt.pipeline.exceptions.PipelineStepFailed:
Pipeline execution failed at `step=extract`
when processing package with `load_id=1757438841.9585924` with exception:

<class 'dlt.extract.exceptions.ResourceExtractionError'>
In processing pipe `sample_data`: extraction of resource `sample_data`
in `transform` `PydanticValidator` caused an exception:
In Table: `sample_data` Column: `('id',)` .
Contract on `data_type` with `contract_mode=freeze` is violated.
Input should be a valid string
```

You can either:

1. amend the value,
2. or change the schema contract,
3. or update the pydantic model
   to solve the error.

Removing columns on schema changes

⚠️ dlt will keep old columns unless `refresh` is passed

Run `python 11_sample_pipeline_schema.py --refresh and then` python 3_sample_pipeline_postgres_config.py`.

What happened with the data?

```
 docker compose exec -it db psql -U postgres -c \
 "select * from sample_data.samples"

 id | name  |             random_field             | my_transformed_field |_dlt_load_id    |    _dlt_id
----+-------+--------------------------------------+----------------------+----------------+----------------
  0 | Mr. a | b7451d0d-4911-4a97-ba62-5e32eaaee3ab |                      | 1757436631.5373347 | 8sULZ9xAZnsMRw
  1 | Mr. b | a274fa1d-6099-4419-bf26-4468e9c343f2 |                      | 1757436631.5373347 | gk8dhYVY5w7J2g
(2 rows)
```

Using `dlt init`

- In these examples we have defined the source as a list of dictionaries
- But sources can be anything. The response from an API, etc.
- They only need to return data in this format
- `dlt` provides many sources and destinations in their catalog
- Abstractions for common sources and destinations

Example using `rest_api_source`

Abstracts away the boilerplate code for extracting data from REST APIs

```
import dlt
from dlt.sources.rest_api import rest_api_source
```

Need only to define the `base_url`, `auth` and `resources` in the API

```
source = rest_api_source({
    "client": {
        "base_url": "https://api.example.com/",
        "auth": {
            "token": dlt.secrets["your_api_token"],
        },
        "paginator": {
            "type": "json_link",
            "next_url_path": "paging.next",
        },
    },
    "resources": [
        # "posts" will be used as the endpoint path, the resource name,
        # and the table name in the destination. The HTTP client will send
        # a request to "https://api.example.com/posts".
        "posts",

        # The explicit configuration allows you to link resources
        # and define query string parameters.
        {
            "name": "comments",
            "endpoint": {
                "path": "posts/{resources.posts.id}/comments",
                "params": {
                    "sort": "created_at",
                },
            },
        },
    ],
})
```

Finally you run this with

```
pipeline = dlt.pipeline(
    pipeline_name="rest_api_example",
    destination="duckdb",
    dataset_name="rest_api_data",
)

load_info = pipeline.run(source)
```

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 is not always consistent! Check the generated code before running it.

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

Using the CLI

`dlt` provides a CLI tool to manage pipelines

```
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!

Advanced stuff

inspecting dlt state

The `state` is a dictionary created by dlt every pipeline run. It can happen that during a pipeline run an error may occur that will halt the execution. We can simulate an error like

```
--- 12_sample_pipeline_schema_with_pydantic.py 2025-09-09 14:28:16.878957096 -0300
+++ 14_sample_pipeline_debugging_state.py 2025-09-09 14:55:16.220515913 -0300
@@ -24,6 +24,7 @@
     },
 )
 def sample_data() -> Generator[dict, None, None]:
+    raise Exception("Debugging state: intentional error raised")
     for x in range(2):
         yield {
             "id": x,
```