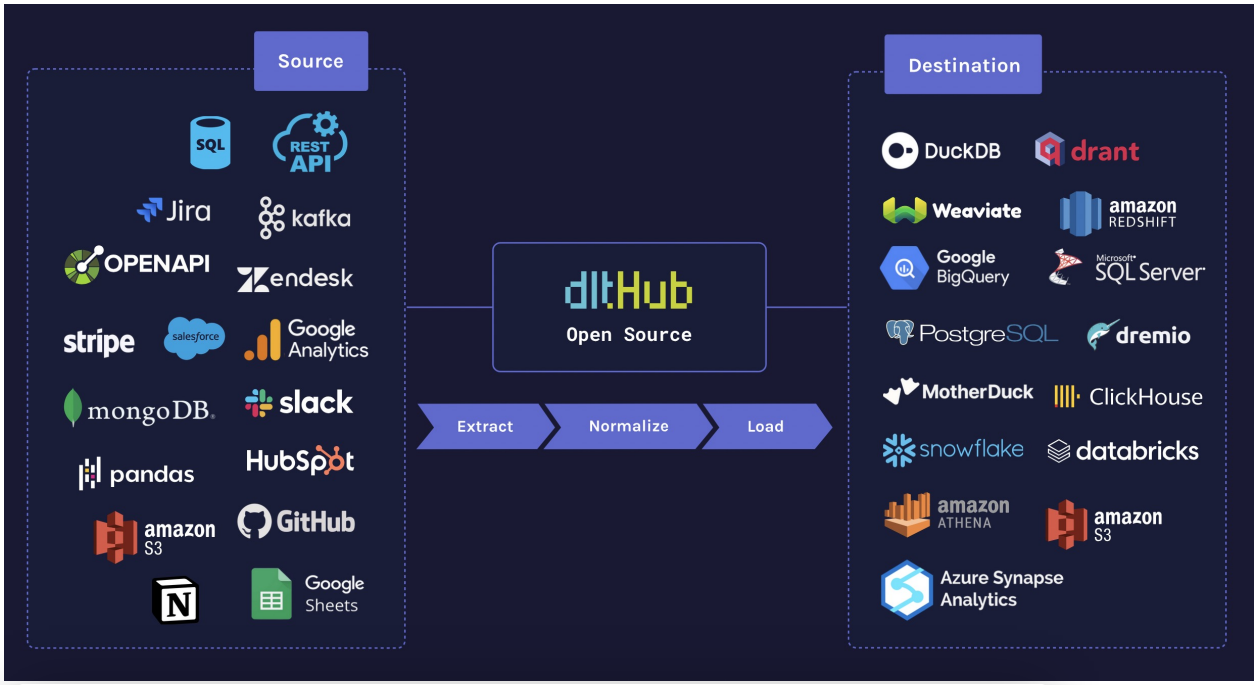
How dlt works

In a nutshell, dlt automatically turns data from a number of available sources (e.g., an API, a PostgreSQL database, or Python data structures) into a live dataset stored in a destination of your choice (e.g., Google BigQuery, a Deltalake on Azure, or by pushing the data back via reverse ETL). You can easily implement your own sources, as long as you yield data in a way that is compatible with dlt, such as JSON objects, Python lists and dictionaries, pandas dataframes, and arrow tables. dlt will be able to automatically compute the schema and move the data to your destination.



T ur source into y nally short e

```
import dlt

pipeline = dlt.pipeline(pipeline_name="my_pipeline", destination="duckdb")
pipeline.run(
    [
        {"id": 1},
        {"id": 2},
```

```
        {"id": 3, "nested": [{"id": 1}, {"id": 2}]},
    ],
    table_name="items",
)
```

This is what happens when the `run` method is executed:

1. Extract - Fully extracts the data from your source to your hard drive. In the example above, an implicit source with one resource with 3 items is created and extracted.
2. Normalize - Inspects and normalizes your data and computes a schema compatible with your destination. For the example above, the normalizer will detect one column `id` of type `int` in one table named `items`, it will furthermore detect a nested list in table items and unnest it into a child table named `items__nested`.
3. Load - Runs schema migrations if necessary on your destination and loads your data into the destination. For the example above, a new dataset on a local duckdb database is created that contains the two tables discovered in the previous steps.

The three phases

Extract

Extract can be run individually with the `extract` command on the pipeline:

```
pipeline.extract(data)
```

During the extract phase, dlt fully extracts the data from your sources to your hard drive into a new load package, which will be assigned a unique ID and will contain your raw data as received from your sources. Additionally, you can supply schema hints to define the data types of some of the columns or add a primary key and unique indexes. You can also control this phase by limiting the number of items extracted in one run, using incremental cursor fields, and by tuning the performance with parallelization. You can also apply filters and maps to obfuscate or remove personal data, and you can use transformers to create derivative data.

Normalize

Normalize can be run individually with the `normalize` command on the pipeline. Normalize is dependent on having a completed extract phase and will not do anything if there is no extracted data.

```
pipeline.normalize()
```

During the normalization phase, dlt inspects and normalizes your data and computes a schema corresponding to the input data. The schema will automatically evolve to accommodate any future source data changes like new columns or tables. dlt will also unnest nested data structures into child tables and create variant columns if detected values do not match a schema computed during a previous run. The result of the normalization phase is an updated load package that holds your normalized data in a format your destination understands and a full schema which can be used to migrate your data to your destination. You

can control the normalization phase, for example, by defining the allowed nesting level of input data, by applying schema contracts that govern how the schema might evolve, and how rows that do not fit are treated. Performance settings are also available.

Load

Load can be run individually with the `load` command on the pipeline. Load is dependent on having a completed normalize phase and will not do anything if there is no normalized data.

```
pipeline.load()
```

During the loading phase, dlt first runs schema migrations as needed on your destination and then loads your data into the destination. dlt will load your data in smaller chunks called load jobs to be able to parallelize large loads. If the connection to the destination fails, it is safe to rerun the pipeline, and dlt will continue to load all load jobs from the current load package. dlt will also create special tables that store the internal dlt schema, information about all load packages, and some state information which, among other things, are used by the incrementals to be able to restore the incremental state from a previous run to another machine. Some ways to control the loading phase are by using different `write dispositions` to replace the data in the destination, simply append to it, or merge on certain merge keys that you can configure per table. For some destinations, you can use a remote staging dataset on a bucket provider, and dlt even supports modern open table formats like deltables and iceberg, and reverse ETL is also possible.

Other notable dlt features

- dlt is simply a Python package, so it will run everywhere that Python runs — locally, in notebooks, on orchestrators — you name it.
- dlt allows you to build and test your data pipelines locally with DuckDB and then switch out the destination for deployment.
- dlt provides a user-friendly interface for accessing your data in Python, using the dashboard app, and leveraging integrations with the fabulous Ibis library. All of this even works on data lakes provided by bucket storage providers.
- dlt fully manages schema migrations on your destinations. You don't even need to know how to use SQL to update your schema. It also supports schema contracts to govern how the schema might evolve.
- dlt offers numerous options for monitoring and tracing what is happening during your loads.
- dlt supports you when you need to transform your data after the load, whether with dbt or in Python using Arrow tables and Pandas DataFrames.

Edit this page

Previous
« Release highlights: 1.15

Next
Source »

Community

Slack

Email

More

GitHub

Twitter