LanceDB

LanceDB

LanceDB is an open-source, high-performance vector database. It allows you to store data objects and perform similarity searches over them. This destination helps you load data into LanceDB from dlt resources.

Setup guide

Choosing a model provider

First, you need to decide which embedding model provider to use. You can find all supported providers by visiting the official LanceDB docs.

Install dlt with LanceDB

To use LanceDB as a destination, make sure `dlt` is installed with the `lancedb` extra:

```
pip install "dlt[lancedb]"
```

The lancedb extra only installs `dlt` and `lancedb`. You will need to install your model provider's SDK.

You can find which libraries you need by also referring to the LanceDB docs.

Configure the destination

Configure the destination in the dlt secrets file located at `~/.dlt/secrets.toml` by default. Add the following section:

```
[destination.lancedb]
embedding_model_provider = "ollama"
embedding_model = "mxbai-embed-large"
embedding_model_provider_host = "http://localhost:11434"  # Optional: custom endpoint for pr

[destination.lancedb.credentials]
uri = ".lancedb"
api_key = "api_key" # API key to connect to LanceDB Cloud. Leave out if you are using Lancel
embedding_model_provider_api_key = "embedding_model_provider_api_key" # Not needed for prov
```

- The `uri` specifies the location of your LanceDB instance. It defaults to a local, on-disk instance if not provided.
- The `api_key` is your API key for LanceDB Cloud connections. If you're using LanceDB OSS, you don't need to supply this key.
- The `embedding_model_provider` specifies the embedding provider used for generating embeddings. The default is `cohere`.
- The `embedding_model` specifies the model used by the embedding provider for generating embeddings. Check with the embedding provider which options are available. Reference https://lancedb.github.io/lancedb/embeddings/default_embedding_function
- The `embedding_model_provider_host` specifies the full host URL with protocol and port for providers that support custom endpoints (like Ollama). If not specified, the provider's default endpoint will be used.
- The `embedding_model_provider_api_key` is the API key for the embedding model provider used to generate embeddings. If you're using a provider that doesn't need authentication, such as Ollama, you don't need to supply this key.

:::info Available model providers - "gemini-text" - "bedrock-text" - "cohere" - "gte-text" - "imagebind" - "instructor" - "open-clip" - "openai" - "sentence-transformers" - "huggingface" - "colbert" - "ollama" :::

Define your data source

For example:

```
import dlt
from dlt.destinations.adapters import lancedb_adapter


movies = [
  {
    "id": 1,
    "title": "Blade Runner",
    "year": 1982,
  },
  {
    "id": 2,
    "title": "Ghost in the Shell",
    "year": 1995,
  },
  {
    "id": 3,
    "title": "The Matrix",
    "year": 1999,
  },
]
```

Create a pipeline:

```
pipeline = dlt.pipeline(
  pipeline_name="movies",
  destination="lancedb",
  dataset_name="MoviesDataset",
)
```

Run the pipeline:

```
info = pipeline.run(
  lancedb_adapter(
    movies,
    embed="title",
  )
)
```

The data is now loaded into LanceDB.

To use **vector search** after loading, you **must specify which fields LanceDB should generate embeddings for**. Do this by wrapping the data (or dlt resource) with the `lancedb_adapter` function.

Using an adapter to specify columns to vectorize

Out of the box, LanceDB will act as a normal database. To use LanceDB's embedding facilities, you'll need to specify which fields you'd like to embed in your dlt resource.

The `lancedb_adapter` is a helper function that configures the resource for the LanceDB destination:

```
lancedb_adapter(data, embed="title")
```

It accepts the following arguments:

- `data`: a dlt resource object, or a Python data structure (e.g., a list of dictionaries).
- `embed`: a name of the field or a list of names to generate embeddings for.

Returns: dlt resource object that you can pass to the `pipeline.run()`.

Example:

```
lancedb_adapter(
  resource,
  embed=["title", "description"],
)
```

When using the `lancedb_adapter`, it's important to apply it directly to resources, not to the whole source. Here's an example:

```
products_tables = sql_database().with_resources("products", "customers")

pipeline = dlt.pipeline(
        pipeline_name="postgres_to_lancedb_pipeline",
        destination="lancedb",
    )

# Apply adapter to the needed resources
lancedb_adapter(products_tables.products, embed="description")
lancedb_adapter(products_tables.customers, embed="bio")

info = pipeline.run(products_tables)
```

Write disposition

All write dispositions are supported by the LanceDB destination.

Replace

The replace disposition replaces the data in the destination with the data from the resource.

```
info = pipeline.run(
  lancedb_adapter(
    movies,
    embed="title",
  ),
  write_disposition="replace",
)
```

Merge

The merge write disposition merges the data from the resource with the data at the destination based on a unique identifier. The LanceDB destination merge write disposition only supports upsert strategy. This updates existing records and inserts new ones based on a unique identifier.

You can specify the merge disposition, primary key, and merge key either in a resource or adapter:

```
@dlt.resource(
  primary_key=["doc_id", "chunk_id"],
  merge_key=["doc_id"],
  write_disposition={"disposition": "merge", "strategy": "upsert"},
)
def my_rag_docs(
  data: List[DictStrAny],
```

```
) -> Generator[List[DictStrAny], None, None]:
    yield data
```

Or:

```
pipeline.run(
  lancedb_adapter(
    my_new_rag_docs,
    merge_key="doc_id"
  ),
  write_disposition={"disposition": "merge", "strategy": "upsert"},
  primary_key=["doc_id", "chunk_id"],
)
```

The `primary_key` uniquely identifies each record, typically comprising a document ID and a chunk ID. The `merge_key`, which cannot be compound, should correspond to the canonical `doc_id` used in vector databases and represent the document identifier in your data model. It must be the first element of the `primary_key`. This `merge_key` is crucial for document identification and orphan removal during merge operations. This structure ensures proper record identification and maintains consistency with vector database concepts.

Orphan Removal LanceDB **automatically removes orphaned chunks** when updating or deleting parent documents during a merge operation. To disable this feature:

```
pipeline.run(
  lancedb_adapter(
    movies,
    embed="title",
    no_remove_orphans=True # Disable with the `no_remove_orphans` flag.
  ),
  write_disposition={"disposition": "merge", "strategy": "upsert"},
  primary_key=["doc_id", "chunk_id"],
)
```

Note: While it's possible to omit the `merge_key` for brevity (in which case it is assumed to be the first entry of `primary_key`), explicitly specifying both is recommended for clarity.

Append

This is the default disposition. It will append the data to the existing data in the destination.

Additional destination options

- `dataset_separator`: The character used to separate the dataset name from table names. Defaults to "____".
- `vector_field_name`: The name of the special field to store vector embeddings. Defaults to "vector".
- `max_retries`: The maximum number of retries for embedding operations. Set to 0 to disable retries. Defaults to 3.

dbt support

The LanceDB destination doesn't support dbt integration.

Syncing of `dlt` state

The LanceDB destination supports syncing of the `dlt` state.

Current limitations

In-memory tables

Adding new fields to an existing LanceDB table requires loading the entire table data into memory as a PyArrow table. This is because PyArrow tables are immutable, so adding fields requires creating a new table with the updated schema.

For huge tables, this may impact performance and memory usage since the full table must be loaded into memory to add the new fields. Keep these considerations in mind when working with large datasets and monitor memory usage if adding fields to sizable existing tables.

Null string handling for OpenAI embeddings

OpenAI embedding service doesn't accept empty string bodies. We deal with this by replacing empty strings with a placeholder that should be very semantically dissimilar to 99.9% of queries.

If your source column (column which is embedded) has empty values, it is important to consider the impact of this. There might be a *slight* chance that semantic queries can hit these empty strings.

We reported this issue to LanceDB: https://github.com/lancedb/lancedb/issues/1577.

MotherDuck

MotherDuck

Install dlt with MotherDuck

To install the dlt library with MotherDuck dependencies:

```
pip install "dlt[motherduck]"
```

If you see a lot of retries in your logs with various timeouts, decrease the number of load workers to 3-5 depending on the quality of your internet connection. Add the following to your `config.toml`:

```
[load]
workers=3
```

or export the **LOAD___WORKERS=3** env variable. See more in performance

Setup guide

1. Initialize a project with a pipeline that loads to MotherDuck by running

```
dlt init chess motherduck
```

2. Install the necessary dependencies for MotherDuck by running

```
pip install -r requirements.txt
```

This will install dlt with the **motherduck** extra which contains **duckdb** and **pyarrow** dependencies.

3. Add your MotherDuck token to `.dlt/secrets.toml`

```
[destination.motherduck.credentials]
database = "dlt_data_3"
password = "<your token here>"
```

Paste your **service token** into the password field. The `database` field is optional, but we recommend setting it. MotherDuck will create this database (in this case `dlt_data_3`) for you.

Alternatively, you can use the connection string syntax.

```
[destination]
motherduck.credentials="md:dlt_data_3?motherduck_token=<my service token>"
```

Motherduck now supports configurable **access tokens**. Please refer to the documentation

You can pass token in a native Motherduck environment variable:

```
export motherduck_token='<token>'
```

in that case you can skip **password** / **motherduck_token** secret.

database defaults to `my_db`.

More in Motherduck documentation

4. Run the pipeline

```
python3 chess_pipeline.py
```

Motherduck connection identifier

We enable Motherduck to identify that the connection is created by `dlt`. Motherduck will use this identifier to better understand the usage patterns associated with `dlt` integration. The connection identifier is `dltHub_dlt/DLT_VERSION(OS_NAME)`.

Write disposition

All write dispositions are supported.

Data loading

By default, Parquet files and the `COPY` command are used to move files to the remote duckdb database. All write dispositions are supported.

The **INSERT** format is also supported and will execute large INSERT queries directly into the remote database. This method is significantly slower and may exceed the maximum query size, so it is not advised.

dbt support

This destination integrates with dbt via dbt-duckdb, which is a community-supported package. `dbt` version >= 1.7 is required.

Multi-statement transaction support

Motherduck supports multi-statement transactions. This change happened with `duckdb 0.10.2`.

Syncing of `dlt` state

This destination fully supports dlt state sync.

Troubleshooting

My database is attached in read-only mode

i.e., `Error: Invalid Input Error: Cannot execute statement of type "CREATE" on database "dlt_data" which is attached in read-only mode!` We encountered this problem for databases created with `duckdb 0.9.x` and then migrated to `0.10.x`. After switching to `1.0.x` on Motherduck, all our databases had permission "read-only" visible in UI. We could not figure out how to change it, so we dropped and recreated our databases.

I see some exception with home_dir missing when opening `md:` connection.

Some internal component (HTTPS) requires the **HOME** env variable to be present. Export such a variable to the command line. Here is what we do in our tests:

```
os.environ["HOME"] = "/tmp"
```

before opening the connection.

Microsoft SQL Server

Microsoft SQL Server

Install dlt with MS SQL

To install the dlt library with MS SQL dependencies, use:

```
pip install "dlt[mssql]"
```

Setup guide

Prerequisites

The *Microsoft ODBC Driver for SQL Server* must be installed to use this destination. This cannot be included with `dlt`'s Python dependencies, so you must install it separately on your system. You can find the official installation instructions here.

Supported driver versions: * `ODBC Driver 18 for SQL Server` * `ODBC Driver 17 for SQL Server`

You can also configure the driver name explicitly.

Create a pipeline

1. Initialize a project with a pipeline that loads to MS SQL by running:

```
dlt init chess mssql
```

2. Install the necessary dependencies for MS SQL by running:

```
pip install -r requirements.txt
```

or run:

```
pip install "dlt[mssql]"
```

This will install `dlt` with the `mssql` extra, which contains all the dependencies required by the SQL server client.

3. Enter your credentials into `.dlt/secrets.toml`.

For example, replace with your database connection info:

```toml
[destination.mssql.credentials]
database = "dlt_data"
username = "loader"
password = "<password>"
host = "loader.database.windows.net"
port = 1433
connect_timeout = 15
[destination.mssql.credentials.query]
# trust self-signed SSL certificates
TrustServerCertificate="yes"
# require SSL connection
Encrypt="yes"
# send large string as VARCHAR, not legacy TEXT
LongAsMax="yes"
```

You can also pass a SQLAlchemy-like database connection:

```toml
# Keep it at the top of your TOML file, before any section starts
destination.mssql.credentials="mssql://loader:<password>@loader.database.windows.net/dlt_dat
```

You can place any ODBC-specific settings into the query string or **destination.mssql.credentials.query** TOML table as in the example above.

To connect to an `mssql` server using Windows authentication, include `trusted_connection=yes` in the connection string.

```toml
destination.mssql.credentials="mssql://loader.database.windows.net/dlt_data?trusted_connecti
```

If you encounter missing credentials errors when using Windows authentication, set the 'username' and 'password' as empty strings in the TOML file.

To connect to a local SQL server instance running without SSL, pass the `encrypt=no` parameter:

```toml
destination.mssql.credentials="mssql://loader:loader@localhost/dlt_data?encrypt=no"
```

To allow a self-signed SSL certificate when you are getting `certificate verify failed: unable to get local issuer certificate`:

```toml
destination.mssql.credentials="mssql://loader:loader@localhost/dlt_data?TrustServerCertifica
```

To use long strings (>8k) and avoid collation errors:

```toml
destination.mssql.credentials="mssql://loader:loader@localhost/dlt_data?LongAsMax=yes"
```

To pass credentials directly, use the explicit instance of the destination

```
pipeline = dlt.pipeline(
  pipeline_name='chess',
  destination=dlt.destinations.mssql("mssql://loader:<password>@loader.database.windows.net/
  dataset_name='chess_data')
```

Write disposition

All write dispositions are supported.

If you set the `replace` strategy to `staging-optimized`, the destination tables will be dropped and recreated with an `ALTER SCHEMA ... TRANSFER`. The operation is atomic: MSSQL supports DDL transactions.

Data loading

Data is loaded via INSERT statements by default. MSSQL has a limit of 1000 rows per INSERT, and this is what we use.

Supported file formats

- insert-values is used by default

Supported column hints

mssql will create unique indexes for all columns with `unique` hints. This behavior **may be disabled**.

Table and column identifiers

SQL Server **with the default collation** uses case-insensitive identifiers but will preserve the casing of identifiers that are stored in the INFORMATION SCHEMA. You can use case-sensitive naming conventions to keep the identifier casing. Note that you risk generating identifier collisions, which are detected by `dlt` and will fail the load process.

If you change the SQL Server server/database collation to case-sensitive, this will also affect the identifiers. Configure your destination as below in order to use case-sensitive naming conventions without collisions:

```
[destination.mssql]
has_case_sensitive_identifiers=true
```

Syncing of `dlt` state

This destination fully supports dlt state sync.

Data types

MS SQL does not support JSON columns, so JSON objects are stored as strings in `nvarchar` columns.

Additional destination options

The **mssql** destination **does not** create UNIQUE indexes by default on columns with the `unique` hint (i.e., `_dlt_id`). To enable this behavior:

```
[destination.mssql]
create_indexes=true
```

You can explicitly set the ODBC driver name:

```
[destination.mssql.credentials]
driver="ODBC Driver 18 for SQL Server"
```

When using a SQLAlchemy connection string, replace spaces with `+`:

```
# Keep it at the top of your TOML file, before any section starts
destination.mssql.credentials="mssql://loader:<password>@loader.database.windows.net/dlt_dat
```

dbt support

This destination integrates with dbt via dbt-snowflake.

30+ SQL databases (powered by SQLAlchemy)

SQLAlchemy destination

The SQLAlchemy destination allows you to use any database that has an SQLAlchemy dialect implemented as a destination.

Currently, MySQL and SQLite are considered to have full support and are tested as part of the `dlt` CI suite. Other dialects are not tested but should generally work.

Install dlt with SQLAlchemy

Install dlt with the `sqlalchemy` extra dependency:

```
pip install "dlt[sqlalchemy]"
```

Note that database drivers are not included and need to be installed separately for the database you plan on using. For example, for MySQL:

```
pip install mysqlclient
```

Refer to the SQLAlchemy documentation on dialects for information about client libraries required for supported databases.

Create a pipeline

1. Initialize a project with a pipeline that loads to MS SQL by running:

```
dlt init chess sqlalchemy
```

2. Install the necessary dependencies for SQLAlchemy by running:

```
pip install -r requirements.txt
```

or run:

```
pip install "dlt[sqlalchemy]"
```

3. Install your database client library.

E.g., for MySQL:

```
pip install mysqlclient
```

4. Enter your credentials into `.dlt/secrets.toml`.

For example, replace with your database connection info:

```
[destination.sqlalchemy.credentials]
database = "dlt_data"
username = "loader"
password = "<password>"
host = "localhost"
port = 3306
driver_name = "mysql"
```

Alternatively, a valid SQLAlchemy database URL can be used, either in `secrets.toml` or as an environment variable. E.g.

```
[destination.sqlalchemy]
credentials = "mysql://loader:<password>@localhost:3306/dlt_data"
```

or

```
export DESTINATION__SQLALCHEMY__CREDENTIALS="mysql://loader:<password>@localhost:3306/dlt_da
```

An SQLAlchemy `Engine` can also be passed directly by creating an instance of the destination:

```
import sqlalchemy as sa
import dlt

engine = sa.create_engine('sqlite:///chess_data.db')

pipeline = dlt.pipeline(
    pipeline_name='chess',
    destination=dlt.destinations.sqlalchemy(engine),
    dataset_name='main'
)
```

Notes on SQLite

Dataset files

When using an SQLite database file, each dataset is stored in a separate file since SQLite does not support multiple schemas in a single database file. Under the hood, this uses `ATTACH DATABASE`.

The file is stored in the same directory as the main database file (provided by your database URL).

E.g., if your SQLite URL is `sqlite:////home/me/data/chess_data.db` and your `dataset_name` is `games`, the data is stored in `/home/me/data/chess_data__games.db`

Note: If the dataset name is `main`, no additional file is created as this is the default SQLite database.

In-memory databases

In-memory databases require a persistent connection as the database is destroyed when the connection is closed. Normally, connections are opened and closed for each load job and in other stages during the pipeline run. To ensure the database persists throughout the pipeline run, you need to pass in an SQLAlchemy `Engine` object instead of credentials. This engine is not disposed of automatically by `dlt`. Example:

```
import dlt
import sqlalchemy as sa

# Create the SQLite engine
engine = sa.create_engine('sqlite:///:memory:')

# Configure the destination instance and create pipeline
pipeline = dlt.pipeline('my_pipeline', destination=dlt.destinations.sqlalchemy(engine), data

# Run the pipeline with some data
pipeline.run([1,2,3], table_name='my_table')

# The engine is still open and you can query the database
with engine.connect() as conn:
    result = conn.execute(sa.text('SELECT * FROM my_table'))
    print(result.fetchall())
```

Write dispositions

The following write dispositions are supported:

- `append`
- `replace` with `truncate-and-insert` and `insert-from-staging` replace strategies. `staging-optimized` falls back to `insert-from-staging`.
- `merge` with `delete-insert` and `scd2` merge strategies.

Data loading

Data is loaded in a dialect-agnostic manner with an `insert` statement generated by SQLAlchemy's core API. Rows are inserted in batches as long as the underlying database driver supports it. By default, the batch size is 10,000 rows.

Syncing of `dlt` state

This destination fully supports dlt state sync.

Data types

All `dlt` data types are supported, but how they are stored in the database depends on the SQLAlchemy dialect. For example, SQLite does not have `DATETIME` or `TIMESTAMP` types, so `timestamp` columns are stored as `TEXT` in ISO 8601 format.

Supported file formats

- typed-jsonl is used by default. JSON-encoded data with typing information included.
- Parquet is supported.

Supported column hints

- `unique` hints are translated to `UNIQUE` constraints via SQLAlchemy (granted the database supports it).

CSV

import SetTheFormat from './_set_the_format.mdx';

CSV file format

CSV is the most basic file format for storing tabular data, where all values are strings and are separated by a delimiter (typically a comma). `dlt` uses it for specific use cases - mostly for performance and compatibility reasons.

Internally, we use two implementations: - **pyarrow** CSV writer - a very fast, multithreaded writer for Arrow tables - **python stdlib writer** - a csv writer included in the Python standard library for Python objects

Supported destinations

The CSV format is supported by the following destinations: **Postgres**, **Filesystem**, **Snowflake**

How to configure

Default settings

`dlt` attempts to make both writers generate similarly looking files: * separators are commas * quotes are **“** and are escaped as **”** * `NULL` values are both empty strings and empty tokens as in the example below * UNIX new lines are used * dates are represented as ISO 8601 * quoting style is”when needed”

Example of NULLs:

```
text1,text2,text3
A,B,C
A,,""
```

In the last row, both `text2` and `text3` values are NULL. The Python `csv` writer is not able to write unquoted `None` values, so we had to settle for `""`.

Note: all destinations capable of writing CSVs must support it.

Change settings

You can change basic **csv** settings; this may be handy when working with the **filesystem** destination. Other destinations are tested with standard settings:

- delimiter: change the delimiting character (default: ‘,’)
- include_header: include the header row (default: True)
- quoting: **quote_all** - all values are quoted, **quote_needed** - quote only values that need quoting (default: `quote_needed`)

When **quote_needed** is selected: in the case of the Python csv writer, all non-numeric values are quoted. In the case of the pyarrow csv writer, the exact behavior is not described in the documentation. We observed that in some cases, strings are not quoted as well.

```
[normalize.data_writer]
delimiter="|"
include_header=false
quoting="quote_all"
```

Or using environment variables:

```
NORMALIZE__DATA_WRITER__DELIMITER=|
NORMALIZE__DATA_WRITER__INCLUDE_HEADER=False
NORMALIZE__DATA_WRITER__QUOTING=quote_all
```

Destination settings

A few additional settings are available when copying `csv` to destination tables: * **on_error_continue** - skip lines with errors (only Snowflake) * **encoding** - encoding of the `csv` file

You’ll need these settings when importing external files.

Limitations

arrow writer

- binary columns are supported only if they contain valid UTF-8 characters
- json (nested, struct) types are not supported

csv writer * binary columns are supported only if they contain valid UTF-8 characters (easy to add more encodings) * json columns dumped with json.dumps * **None** values are always quoted

INSERT

import SetTheFormat from './_set_the_format.mdx';

SQL INSERT file format

This file format contains an INSERT... VALUES statement to be executed on the destination during the `load` stage.

Additional data types are stored as follows:

- `datetime` and `date` are stored as ISO strings;
- `decimal` is stored as a text representation of a decimal number;
- `binary` storage depends on the format accepted by the destination;
- `json` storage also depends on the format accepted by the destination.

This file format is compressed by default.

Supported destinations

This format is used by default by: **DuckDB**, **Postgres**, **Redshift**, **Synapse**, **MSSQL**, **Motherduck**

It is also supported by: **Filesystem** if you'd like to store INSERT VALUES statements for some reason.

How to configure

JSONL

import SetTheFormat from './_set_the_format.mdx';

JSONL - JSON Lines - JSON Delimited

JSON delimited is a file format that stores several JSON documents in one file. The JSON documents are separated by a new line.

Additional data types are stored as follows:

- `datetime` and `date` are stored as ISO strings;
- `decimal` is stored as a text representation of a decimal number;
- `binary` is stored as a base64 encoded string;
- `HexBytes` is stored as a hex encoded string;
- `json` is serialized as a string.

This file format is compressed by default.

Supported destinations

This format is used by default by: **BigQuery**, **Snowflake**, **Filesystem**.

How to configure

Parquet

import SetTheFormat from './_set_the_format.mdx';

Parquet file format

Apache Parquet is a free and open-source column-oriented data storage format in the Apache Hadoop ecosystem. `dlt` is capable of storing data in this format when configured to do so.

To use this format, you need the `pyarrow` package. You can get this package as a `dlt` extra as well:

```
pip install "dlt[parquet]"
```

Supported destinations

Supported by: **BigQuery**, **DuckDB**, **Snowflake**, **Filesystem**, **Athena**, **Databricks**, **Synapse**

How to configure

Destination autoconfig

`dlt` uses destination capabilities to configure the parquet writer: * It uses decimal and wei precision to pick the right **decimal type** and sets precision and scale. * It uses timestamp precision to pick the right **timestamp type** resolution (seconds, micro, or nano).

Writer settings

Under the hood, `dlt` uses the pyarrow parquet writer to create the files. The following options can be used to change the behavior of the writer:

- `flavor`: Sanitize schema or set other compatibility options to work with various target systems. Defaults to None, which is the **pyarrow** default.
- `version`: Determine which Parquet logical types are available for use, whether the reduced set from the Parquet 1.x.x format or the expanded logical types added in later format versions. Defaults to "2.6".

- `data_page_size`: Set a target threshold for the approximate encoded size of data pages within a column chunk (in bytes). Defaults to None, which is the **pyarrow** default.
- `row_group_size`: Set the number of rows in a row group. See here how this can optimize parallel processing of queries on your destination over the default setting of `pyarrow`.
- `timestamp_timezone`: A string specifying the timezone, default is UTC.
- `coerce_timestamps`: resolution to which to coerce timestamps, choose from **s**, **ms**, **us**, **ns**
- `allow_truncated_timestamps` - will raise if precision is lost on truncated timestamps.

The default parquet version used by `dlt` is 2.4. It coerces timestamps to microseconds and truncates nanoseconds silently. Such a setting provides the best interoperability with database systems, including loading panda frames which have nanosecond resolution by default.

Read the pyarrow parquet docs to learn more about these settings.

Example:

```toml
[normalize.data_writer]
# the default values
flavor="spark"
version="2.4"
data_page_size=1048576
timestamp_timezone="Europe/Berlin"
```

Or using environment variables:

```
NORMALIZE__DATA_WRITER__FLAVOR
NORMALIZE__DATA_WRITER__VERSION
NORMALIZE__DATA_WRITER__DATA_PAGE_SIZE
NORMALIZE__DATA_WRITER__TIMESTAMP_TIMEZONE
```

Timestamps and timezones

`dlt` adds timezone (UTC adjustment) to all timestamps regardless of the precision (from seconds to nanoseconds). `dlt` will also create TZ-aware timestamp columns in the destinations. DuckDB is an exception here.

Disable timezones / UTC adjustment flags

You can generate parquet files without timezone adjustment information in two ways: 1. Set the **flavor** to spark. All timestamps will be generated via the deprecated `int96` physical data type, without the logical one. 2. Set the **timestamp_timezone** to an empty string (i.e., `DATA_WRITER__TIMESTAMP_TIMEZONE=""`) to generate a logical type without UTC adjustment.

To our best knowledge, Arrow will convert your timezone-aware DateTime(s) to UTC and store them in parquet without timezone information.

Row group size

The `pyarrow` parquet writer writes each item, i.e., table or record batch, in a separate row group. This may lead to many small row groups, which may not be optimal for certain query engines. For example, `duckdb` parallelizes on a row group. `dlt` allows controlling the size of the row group by buffering and concatenating tables and batches before they are written. The concatenation is done as a zero-copy to save memory. You can control the size of the row group by setting the maximum number of rows kept in the buffer.

```
[extract.data_writer]
buffer_max_items=10e6
```

Keep in mind that `dlt` holds the tables in memory. Thus, 1,000,000 rows in the example above may consume a significant amount of RAM.

The `row_group_size` configuration setting has limited utility with the `pyarrow` writer. It may be useful when you write single very large pyarrow tables or when your in-memory buffer is really large.

Delta

Delta table format

Delta is an open-source table format. `dlt` can store data as Delta tables.

Supported destinations

Supported by: **Databricks**, **filesystem**

Iceberg

Iceberg table format

Iceberg is an open-source table format. `dlt` can store data as Iceberg tables.

Supported destinations

Supported by: **Athena**, **filesystem**