Outline

In [ ]:
```
import pandas as pd
import requests
import json
import time
import dlt
import duckdb
import os
import parquet
import glob
```

1. Data Extraction with dlt (Data Load Tool)

1.1. Extracting API Data With a Generator

Consider a scenario where we want to retrieve data from an HTTP API that supports pagination. This API divides the data into pages, with each page containing up to 1000 records. If a request is made for a page number that exceeds the available data , the API returns an empty response, indicating that there are no more records to fetch.

For this purpose, was created a simple API that returns paginated data with flask and google clound function. The API is available at the following URL:

`https://us-central1-dlthub-analytics.cloudfunctions.net/data_engineering_zoomcamp_api`
To achieve this, we can use a python generator to allow fetch and process the data on a per-page basis without needing to load all data into memory at once. The generator function is a special type of iterator that generates values on the fly as they are requested rather than storing them all at once in memory. We can achieve efficient data retrieval that scales well and conserves memory.

The following script is designed to iteratively request each page of data from the API until all pages have been retrieved, processing each page one by one, without loading the entire dataset into memory at once.

In [ ]:
```
BASE_API_URL = "https://us-central1-dlthub-analytics.cloudfunctions.net/data_engineering_zoomcamp_api"


def paginated_getter():
    '''
        This function handles pagination by requesting one page of data at a time,
        yielding the results to allow for processing in smaller, manageable "microbatches."
    '''
    page_number = 1

    while True:
        # Set the query parameters
        params = {'page': page_number}

        # Make the GET request to the API
        response = requests.get(BASE_API_URL, params=params)
        response.raise_for_status()  # Raise an HTTPError for bad responses
        page_json = response.json()
        print(f'got page number {page_number} with {len(page_json)} records')

        # if the page has no records, stop iterating
        if not page_json:
            break # No more data, break the loop
        else:
            yield page_json
            page_number += 1


if __name__ == '__main__':
    # Open the file once and write as we fetch each page
    with open('nyc_taxi_trip.jsonl', 'w') as file:
        for page_data in paginated_getter():
            for record in page_data:
                json.dump(record, file)
            file.write('\n')  # New line for next record
    print("Data written to .jsonl file")
```

```
got page number 1 with 1000 records
got page number 2 with 1000 records
got page number 3 with 1000 records
got page number 4 with 1000 records
got page number 5 with 1000 records
got page number 6 with 1000 records
got page number 7 with 1000 records
got page number 8 with 1000 records
got page number 9 with 1000 records
got page number 10 with 1000 records
got page number 11 with 0 records
Data written to .jsonl file
```

Here we use a JSON lines file to store the data. The JSON is not efficient for large datasets because the entire file must be read into memory to parse the JSON structure, potentially leading to high memory usage. On the other hand, the JSON lines format has each line treated as a separate JSON object, This allows for reading and writing in chunks and is more memory-efficient for large datasets.

This script is useful when we need to process or analyze data on the fly, page by page, especially when dealing with large datasets that wouldn't fit into memory. We can now check the data by using the `pandas.read_json` function with the `lines=True` parameter to efficiently read the JSON Lines file.

```
In [ ]: file_path = 'nyc_taxi_trip.jsonl'
        chunk_size = 1000
        # Read the .jsonl file in chunks
        chunks = pd.read_json(file_path, lines=True, chunksize=chunk_size)

        # Process each chunk
        for chunk in chunks:
            # can perform operations on each chunk if necessary
            display(chunk.head())
            # Break after first chunk for demonstration
            break
```

	End_Lat	End_Lon	Fare_Amt	Passenger_Count	Payment_Type	Rate_Code	Start_Lat	Start_Lon	Tip_Amt	Tolls_Amt	Total_A
0	40.742963	-73.980072	45.0	1	Credit	NaN	40.641525	-73.787442	9.0	4.15	58
1	40.740187	-74.005698	6.5	1	Credit	NaN	40.722065	-74.009767	1.0	0.00	8
2	40.718043	-74.004745	12.5	5	Credit	NaN	40.761945	-73.983038	2.0	0.00	15
3	40.739637	-73.985233	4.9	1	CASH	NaN	40.749802	-73.992247	0.0	0.00	5
4	40.730032	-73.852693	25.7	1	CASH	NaN	40.776825	-73.949233	0.0	4.15	29

By processing each chunk individually, is possible to perform operations on each part of the dataset sequentially without needing to load the entire dataset into memory. This approach allows transform and analyze large datasets as a whole, chunk by chunk, and then concatenate these transformed chunks into a final dataset to export to a file or database.

1.2. Extracting File Data With a Generator (Streaming)

In the context of JSONL files, where each line is a distinct JSON document (representing a data "row"), the streaming approach is straightforward. The process involves downloading and yielding each line one at a time. This method allows for immediate processing of each "row" as it arrives, optimizing both the speed of data handling and memory usage.

- Pros:
    - Streaming allows data to be processed as it's being downloaded, facilitating faster data handling without waiting for the entire download to complete.
    - Easy Memory Management: Since data is processed in chunks (line by line in the case of JSONL as did before), memory usage is minimized. This approach avoids loading the entire dataset into memory, which is particularly beneficial for large files.
- Cons:
    - Complexity with Columnar Formats: For formats like Parquet or ORC, which are organized by columns rather than rows, streaming can be challenging. These formats require downloading entire blocks of data to deserialize them into rows, complicating the streaming process.
    - Potential Code Complexity: Depending on the data format and the specific requirements of the data handling process, the code for streaming downloads can become complex, requiring careful management of the data stream and error handling.

Consider a scenario where we want to retrieve a large file from a remote server and process it line by line. To do so, we can use a generator to stream the file's contents and process each line individually as in the following code:

In [ ]:
```python
import requests
import json

url = "https://storage.googleapis.com/dtc_zoomcamp_api/yellow_tripdata_2009-06.jsonl"

def stream_download_jsonl(url):
    response = requests.get(url, stream=True)
    response.raise_for_status()  # Raise an HTTPError for bad responses
    for line in response.iter_lines():
        if line:
            yield json.loads(line)

# time the download
import time
start = time.time()

# Use the generator to iterate over rows with minimal memory usage
max_preview_rows = 5
row_counter = 0
df_preview = pd.DataFrame() # Initialize empty DataFrame
for row in stream_download_jsonl(url):
    # Convert the row (dict) into a DataFrame and append it to the df_preview
    df_preview = pd.concat([df_preview, pd.DataFrame([row])], ignore_index=True)
    row_counter += 1
    if row_counter >= max_preview_rows:
        break

display(df_preview)

end = time.time()
print('\n Total time:', end - start)
```

	vendor_name	Trip_Pickup_DateTime	Trip_Dropoff_DateTime	Passenger_Count	Trip_Distance	Start_Lon	Start_Lat	Rate_Code	st
0	VTS	2009-06-14 23:23:00	2009-06-14 23:48:00	1	17.52	-73.787442	40.641525	None	
1	VTS	2009-06-18 17:35:00	2009-06-18 17:43:00	1	1.56	-74.009767	40.722065	None	
2	VTS	2009-06-10 18:08:00	2009-06-10 18:27:00	5	3.37	-73.983038	40.761945	None	
3	VTS	2009-06-14 23:54:00	2009-06-14 23:58:00	1	1.11	-73.992247	40.749802	None	
4	VTS	2009-06-13 13:01:00	2009-06-13 13:23:00	1	11.09	-73.949233	40.776825	None	

```
Total time: 0.3186216354370117
```

1.3 Extracting Data with dlt

Now, let's explore a practical approach to downloading and inspecting data using the dlt library. The dlt library allows for the incremental processing of data from various sources, including generators, with a focus on optimizing memory usage.

The dlt library enables us to load data into a database in an incremental manner. This approach is particularly useful when working with large datasets, as it helps manage memory efficiently by processing data in small, manageable chunks.

we utilize DuckDB as the destination for our data. DuckDB is a lightweight, easy-to-use database optimized for analytical queries. Unlike databases such as PostgreSQL, DuckDB is designed to be used within an application. The entire database can be stored in a single file, making it very easy to distribute and deploy. This database performs computations in-memory, which can significantly speed up analytical queries and also efficiently manages memory to handle datasets larger than RAM.

To install dlt with all the necessary DuckDB dependencies:

```
pip install "dlt[duckdb]"
```

In [ ]:
```python
# define the connection to load to.
# We could switch to Bigquery later
generators_pipeline = dlt.pipeline(destination='duckdb', dataset_name='generators')


# Load any generator to a table at the pipeline destination
info = generators_pipeline.run( paginated_getter(),
                                                                table_name="http_download",
                                                                write_disposition="replace" )

# the outcome metadata is returned by the load
print(info)

# Load the next generator to the same or to a different table.
info = generators_pipeline.run( stream_download_jsonl(url),
                                                                table_name="stream_download",
                                                                write_disposition="replace" )

print(info)
```

```
got page number 1 with 1000 records
got page number 2 with 1000 records
got page number 3 with 1000 records
got page number 4 with 1000 records
got page number 5 with 1000 records
got page number 6 with 1000 records
got page number 7 with 1000 records
got page number 8 with 1000 records
got page number 9 with 1000 records
got page number 10 with 1000 records
got page number 11 with 0 records
Pipeline dlt_ipykernel_launcher load step completed in 2.35 seconds
1 load package(s) were loaded to destination duckdb and into dataset generators
The duckdb destination used duckdb:////home/marcos/GitHub/DE-zoomcamp/workshops/01-data-Ingestion/dlt_ipykernel_l
auncher.duckdb location to store data
Load package 1708189774.5354438 is LOADED and contains no failed jobs
Pipeline dlt_ipykernel_launcher load step completed in 2.25 seconds
1 load package(s) were loaded to destination duckdb and into dataset generators
The duckdb destination used duckdb:////home/marcos/GitHub/DE-zoomcamp/workshops/01-data-Ingestion/dlt_ipykernel_l
auncher.duckdb location to store data
Load package 1708189800.5849261 is LOADED and contains no failed jobs
```

We can use SQL queries to access and analyze the data stored within the `.duckdb` file. To access data from the DuckDB file in a Python environment or Jupyter notebook, you would typically do the following:

1. Connect to the DuckDB database file.
2. Execute SQL queries using the connection.
3. Fetch the results for analysis.

In [ ]:
```python
# 1. Connect to the DuckDB
conn = duckdb.connect(f"{generators_pipeline.pipeline_name}.duckdb")

# 2. Execute SQL queries
    # Set the search path to the dataset name for easy access to the tables.
conn.sql(f"SET search_path = '{generators_pipeline.dataset_name}'")
print('Loaded tables: ')
display(conn.sql("SHOW TABLES"))

print("\n\n\n http_download table:")
# 3. Fetch the results for analysis
    # Query the table and fetch the results into a Pandas DataFrame.
rides = conn.sql("SELECT * FROM http_download").df()
display(rides.head(2))

print("\n\n\n stream_download table:")
passengers = conn.sql("SELECT * FROM stream_download").df()
display(passengers.head(2))
```

```
Loaded tables:
┌─────────────────────┐
│        name         │
│      varchar        │
├─────────────────────┤
│ _dlt_loads          │
│ _dlt_pipeline_state │
│ _dlt_version        │
│ http_download       │
│ stream_download     │
└─────────────────────┘



 http_download table:
```

	end_lat	end_lon	fare_amt	passenger_count	payment_type	start_lat	start_lon	tip_amt	tolls_amt	total_amt	trip_distan
0	40.742963	-73.980072	45.0	1	Credit	40.641525	-73.787442	9.0	4.15	58.15	17.
1	40.740187	-74.005698	6.5	1	Credit	40.722065	-74.009767	1.0	0.00	8.50	1.

```
 stream_download table:
```

	vendor_name	trip_pickup_date_time	trip_dropoff_date_time	passenger_count	trip_distance	start_lon	start_lat	end_lon	
0	VTS	2009-06-14 20:23:00-03:00	2009-06-14 20:48:00-03:00	1	17.52	-73.787442	40.641525	-73.980072	40.
1	VTS	2009-06-18 14:35:00-03:00	2009-06-18 14:43:00-03:00	1	1.56	-74.009767	40.722065	-74.005698	40.

2. Normalization

Nested data is organized in a hierarchical structure, where some elements contain other elements within themselves. When dealing with nested data, especially in formats like JSON, transforming it into a tabular format for databases can be difficult due to the hierarchical nature of the data. This structure is common in formats like JSON or XML, where data can be deeply nested. Example of a nested data:

```
{
  "orderID": 12345,
  "customer": "John Doe",
  "items": [
    {"productID": 987, "name": "Widget", "quantity": 2},
    {"productID": 654, "name": "Gadget", "quantity": 1}
  ]
}
```

The `items` key holds an array of objects, each representing an item in the order. Each object within this array contains details about the item, such as its `productID`, `name`, and `quantity`. In other words, nested lists represent a `1:n` relationship, where each parent record might be associated with multiple child records. In such cases, it's more appropriate to represent these relationships using separate tables.

Let's use dlt to normalize the nested data into a tabular format.

In [ ]:
```
nested_data = [
    {
        "vendor_name": "VTS",
                                "record_hash": "b00361a396177a9cb410ff61f20015ad",
        "time": {
            "pickup": "2009-06-14 23:23:00",
            "dropoff": "2009-06-14 23:48:00"
        },
        "Trip_Distance": 17.52,
        # nested dictionaries could be flattened
        "coordinates": { # coordinates__start__lon
            "start": {
                "lon": -73.787442,
                "lat": 40.641525
            },
            "end": {
                "lon": -73.980072,
                "lat": 40.742963
            }
        },
        "Rate_Code": None,
        "store_and_forward": None,
        "Payment": {
            "type": "Credit",
            "amt": 20.5,
            "surcharge": 0,
            "mta_tax": None,
            "tip": 9,
            "tolls": 4.15,
                        "status": "booked"
        },
        "Passenger_Count": 2,
        # nested lists need to be expressed as separate tables
        "passengers": [
            {"name": "John", "rating": 4.9},
            {"name": "Jack", "rating": 3.9}
        ],
        "Stops": [
            {"lon": -73.6, "lat": 40.6},
            {"lon": -73.5, "lat": 40.5}
        ]
    },
]


# define the connection to load to.
# We now use duckdb, but you can switch to Bigquery later
pipeline = dlt.pipeline(destination='duckdb', dataset_name='taxi_rides')



# run with merge write disposition.
# This is so scaffolding is created for the next example,
# where we look at merging data

info = pipeline.run(    nested_data,
                        table_name="rides",
                        write_disposition="merge",
                        primary_key="record_hash"    )

print(info)
```

```
Pipeline dlt_ipykernel_launcher load step completed in 0.49 seconds
1 load package(s) were loaded to destination duckdb and into dataset taxi_rides
The duckdb destination used duckdb:////home/marcos/GitHub/DE-zoomcamp/workshops/01-data-Ingestion/dlt_ipykernel_l
auncher.duckdb location to store data
Load package 1708189805.4310012 is LOADED and contains no failed jobs
```

Once loaded into a database using the dlt library and normalized into a flat, tabular structure using SQL, the data can be easily queried and analyzed using standard SQL queries.

- dlt library processes the nested JSON data and automatically normalizes it by flattening the nested structure and splitting sub-documents into separate tables.
- During the normalization process, dlt generates unique identifiers (`_dlt_id`) for each record in the parent table and corresponding foreign keys (`_dlt_parent_id`) in the child tables. These generated keys enable us to re-establish the relationships between the parent and child data by joining the tables on these keys.
- dlt library automatically converts data types from the JSON format to database-compatible types. For example, timestamp strings in the JSON data are converted to actual timestamp data types in the database.

TO achieve this, consider the following code:

In [ ]:
```python
conn = duckdb.connect(f"{pipeline.pipeline_name}.duckdb")

conn.sql(f"SET search_path = '{pipeline.dataset_name}'")
print('Loaded tables: ')
display(conn.sql("show tables"))
```

```
Loaded tables:
┌────────────────────┐
│        name        │
│      varchar       │
├────────────────────┤
│ _dlt_loads         │
│ _dlt_pipeline_state│
│ _dlt_version       │
│ rides              │
│ rides__passengers  │
│ rides__stops       │
└────────────────────┘
```

In [ ]:
```python
print("\n\n\n Rides table:")
rides = conn.sql("SELECT * FROM rides").df()
display(rides)

print("\n\n\n Passengers table:")
passengers = conn.sql("SELECT * FROM rides__passengers").df()
display(passengers)

print("\n\n\n Stops table:")
stops = conn.sql("SELECT * FROM rides__stops").df()
display(stops)
```

```
 Rides table:
```

	record_hash	vendor_name	time__pickup	time__dropoff	trip_distance	coordinates__start__lon	coordinates_
0	b00361a396177a9cb410ff61f20015ad	VTS	2009-06-14 20:23:00-03:00	2009-06-14 20:48:00-03:00	17.52	-73.787442	

```
 Passengers table:
```

	name	rating	_dlt_root_id	_dlt_parent_id	_dlt_list_idx	_dlt_id
0	John	4.9	1nSAWAKECC0JEA	1nSAWAKECC0JEA	0	e/7/OvcnJD1uhA
1	Jack	3.9	1nSAWAKECC0JEA	1nSAWAKECC0JEA	1	B/ndNQLSBGpj2g

```
 Stops table:
```

	lon	lat	_dlt_root_id	_dlt_parent_id	_dlt_list_idx	_dlt_id
0	-73.6	40.6	1nSAWAKECC0JEA	1nSAWAKECC0JEA	0	YtrudpQY9gWPVw
1	-73.5	40.5	1nSAWAKECC0JEA	1nSAWAKECC0JEA	1	Q5R2dScCRMuxbg

This creates new tables for each distinct element inside the list and a flattened table for elements in the dictionary.

Let's do the following sql query:

- Select all columns from the `rides`, `rides__passengers`, and `rides__stops`
- Join the `rides` table with the `rides__passengers` and `rides__stops` tables using the `_dlt_id` from `rides` and matching it with the `_dlt_parent_id` in both `rides__passengers` and `rides__stops`.

In [ ]:
```
# To reflect the relationships between parent and child rows, let's join them
print("\n\n\n joined table")
joined = conn.sql(
"""
SELECT r.*, rp.*, rs.*
FROM rides AS r
LEFT JOIN rides__passengers AS rp
  ON r._dlt_id = rp._dlt_parent_id
LEFT JOIN rides__stops AS rs
  ON r._dlt_id = rs._dlt_parent_id
"""            ).df()

display(joined)
```

```
joined table
```

	record_hash	vendor_name	time__pickup	time__dropoff	trip_distance	coordinates__start__lon	coordinates_
0	b00361a396177a9cb410ff61f20015ad	VTS	2009-06-14 20:23:00-03:00	2009-06-14 20:48:00-03:00	17.52	-73.787442	
1	b00361a396177a9cb410ff61f20015ad	VTS	2009-06-14 20:23:00-03:00	2009-06-14 20:48:00-03:00	17.52	-73.787442	
2	b00361a396177a9cb410ff61f20015ad	VTS	2009-06-14 20:23:00-03:00	2009-06-14 20:48:00-03:00	17.52	-73.787442	
3	b00361a396177a9cb410ff61f20015ad	VTS	2009-06-14 20:23:00-03:00	2009-06-14 20:48:00-03:00	17.52	-73.787442	

4 rows × 30 columns

3. Incremental Loading

Suppose now that we want to increment the data in the database. We can use the `dlt` library to incrementally load new data into the database.

The data represents a ride record, including details about the ride, payment, passengers, and their ratings. Initially, passengers John and Jack had different ratings, but due to a payment issue ("cancelled" status in the Payment section), their ratings need to be adjusted. To update the database `write_disposition="merge"` parameter is used to merge the new data with the existing data in the database.

In [ ]:
```
data = [
    {
        "vendor_name": "VTS",
                                "record_hash": "b00361a396177a9cb410ff61f20015ad",
        "time": {
            "pickup": "2009-06-14 23:23:00",
            "dropoff": "2009-06-14 23:48:00"
        },
        "Trip_Distance": 17.52,
        "coordinates": {
            "start": {
                "lon": -73.787442,
                "lat": 40.641525
            },
            "end": {
                "lon": -73.980072,
                "lat": 40.742963
            }
        },
        "Rate_Code": None,
        "store_and_forward": None,
        "Payment": {
            "type": "Credit",
            "amt": 20.5,
            "surcharge": 0,
            "mta_tax": None,
            "tip": 9,
            "tolls": 4.15,
                        "status": "cancelled" # Status changed from booked to cancelled
        },
        "Passenger_Count": 2,
        "passengers": [
            # Changed rating for jack and john
            {"name": "John", "rating": 4.4},
            {"name": "Jack", "rating": 3.6}
        ],
        "Stops": [
            {"lon": -73.6, "lat": 40.6},
            {"lon": -73.5, "lat": 40.5}
        ]
    },
]

pipeline = dlt.pipeline(destination='duckdb', dataset_name='taxi_rides')


info = pipeline.run(    data,
                        table_name="rides",
                        write_disposition="merge",  # Merge ensures that existing records are updated
                        primary_key='record_hash')

conn = duckdb.connect(f"{pipeline.pipeline_name}.duckdb")

conn.sql(f"SET search_path = '{pipeline.dataset_name}'")
print('Loaded tables: ')
display(conn.sql("show tables"))
```

```
Loaded tables:
┌─────────────────────┐
│        name         │
│       varchar       │
├─────────────────────┤
│ _dlt_loads          │
│ _dlt_pipeline_state │
│ _dlt_version        │
│ rides               │
│ rides__passengers   │
│ rides__stops        │
└─────────────────────┘
```

In [ ]:
```
print("\n\n\n Rides table:")
rides = conn.sql("SELECT * FROM rides").df()
display(rides)

print("\n\n\n Pasengers table")
passengers = conn.sql("SELECT * FROM rides__passengers").df()
display(passengers)
print("\n\n\n Stops table")
stops = conn.sql("SELECT * FROM rides__stops").df()
display(stops)
```

```



 Rides table:
```

	record_hash	vendor_name	time__pickup	time__dropoff	trip_distance	coordinates__start__lon	coordinates_
0	b00361a396177a9cb410ff61f20015ad	VTS	2009-06-14 20:23:00-03:00	2009-06-14 20:48:00-03:00	17.52	-73.787442	

```



 Pasengers table
```

	name	rating	_dlt_root_id	_dlt_parent_id	_dlt_list_idx	_dlt_id
0	John	4.4	yTx4L9upBQMLSA	yTx4L9upBQMLSA	0	4eJubAPCUiHm7g
1	Jack	3.6	yTx4L9upBQMLSA	yTx4L9upBQMLSA	1	SjeLUr+iwrKXPw

```



 Stops table
```

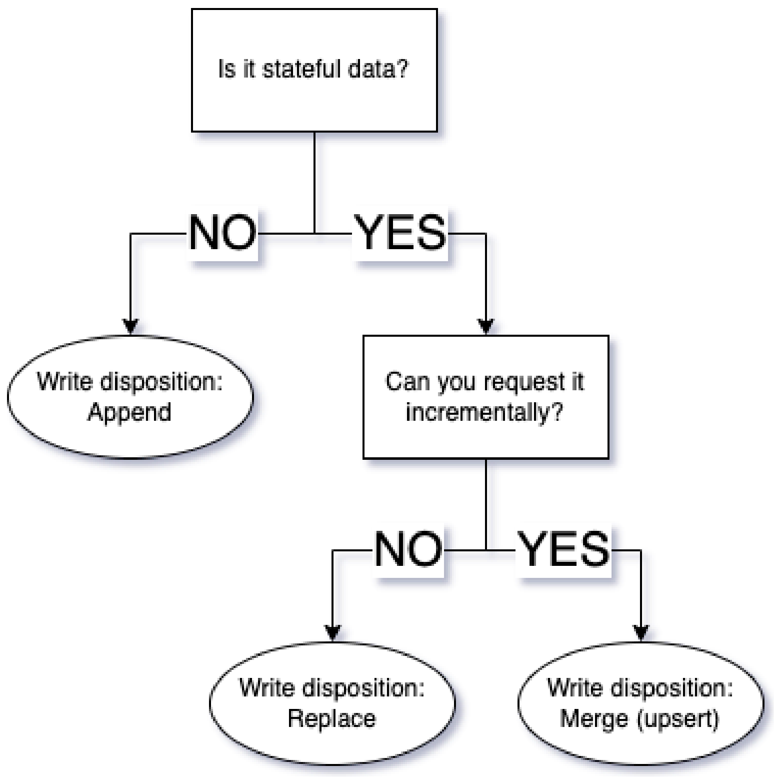
	lon	lat	_dlt_root_id	_dlt_parent_id	_dlt_list_idx	_dlt_id
0	-73.6	40.6	yTx4L9upBQMLSA	yTx4L9upBQMLSA	0	RCHbdlTVd4pZ6Q
1	-73.5	40.5	yTx4L9upBQMLSA	yTx4L9upBQMLSA	1	W/0mx9P9HNHW5w

Here we used the Merge mode, but dlt currently supports 2 ways of loading incrementally:

- Append Mode:
  - **Use Case for Immutable Data**: Ideal for loading data where records do not change over time, such as daily taxi rides. Each day's new rides can be appended to the dataset without needing to reload the entire historical data.

    - **Slowly Changing Dimension (SCD)**: Useful for tracking changes over time in mutable (stateful) data. By appending versions of the data, we can create an audit trail. For example, tracking changes in a car's color over days; each day's data is loaded, enabling the tracking of any color changes.
- Merge Mode:
    - **Dynamic Data Updates**: Best suited for data that undergoes changes. This method allows for existing records to be updated based on new information.
    - **Example Scenario**: Consider taxi rides with a "payment status" field. Initially, a ride may be marked as "booked". Later updates might change the status to "paid", "rejected", or "cancelled". Merge mode ensures these updates are accurately reflected in the dataset.

The choice of which to use can be summarized in the following diagram:


Is it stateful data?
NO
YES
Write disposition: Append
Can you request it incrementally?
NO
YES
Write disposition: Replace
Write disposition: Merge (upsert)

5. Homework

Consider the following generator to answer questions 1 and 2.

In [ ]:
```python
def square_root_generator(limit):
    n = 1
    while n <= limit:
        yield n ** 0.5
        n += 1
```

- **Question 1**: What is the sum of the outputs of the generator for limit = 5?
    - 10.23433234744176
    - 7.892332347441762
    - **8.382332347441762**
    - 9.123332347441762

In [ ]:
```python
limit = 5
total_sum = 0
generator = square_root_generator(limit)
for sqrt_value in generator:
    total_sum += sqrt_value

print("total sum:", total_sum)
```

```
total sum: 8.382332347441762
```

- **Question 2**: What is the 13th number yielded by the generator?
    - 4.236551275463989
    - **3.605551275463989**
    - 2.345551275463989
    - 5.678551275463989

In [ ]:
```python
limit = 13
generator = square_root_generator(limit)
```

```
for sqrt_value in generator:
    print(sqrt_value)
```

```
1.0
1.4142135623730951
1.7320508075688772
2.0
2.23606797749979
2.449489742783178
2.6457513110645907
2.8284271247461903
3.0
3.1622776601683795
3.3166247903554
3.4641016151377544
3.605551275463989
```

Consider the following generators to answer questions 3 and 4.

```
In [ ]: def people_1():
            for i in range(1, 6):
                yield {"ID": i, "Name": f"Person_{i}", "Age": 25 + i, "City": "City_A"}

        def people_2():
            for i in range(3, 9):
                yield {"ID": i, "Name": f"Person_{i}", "Age": 30 + i, "City": "City_B", "Occupation": f"Job_{i}"}
```

- **Question 3**: Append the 2 generators. After correctly appending the data, calculate the sum of all ages of people.
  - **353**
  - 365
  - 378
  - 390

```
In [ ]: pipeline = dlt.pipeline(    destination='duckdb',
                                    dataset_name='people_dataset' )

        pipeline.run(   people_1(),
                        table_name="people",
                        write_disposition="replace")

        pipeline.run(   people_2(),
                        table_name="people",
                        write_disposition="append")

        conn = duckdb.connect(f"{pipeline.pipeline_name}.duckdb")
        conn.sql(f"SET search_path = '{pipeline.dataset_name}'")
        print('Loaded tables: ')
        display(conn.sql("SHOW TABLES"))
```

```
Loaded tables:

┌─────────────────────┐
│        name         │
│       varchar       │
├─────────────────────┤
│ _dlt_loads          │
│ _dlt_pipeline_state │
│ _dlt_version        │
│ people              │
│ people_2            │
└─────────────────────┘
```

```
In [ ]: people = conn.sql("SELECT * FROM people").df()
        display(people.head(2))

        sum_ages = conn.sql("SELECT SUM(Age) FROM people")
        print("Sum of ages:\n\n", sum_ages)
```

	id	name	age	city	_dlt_load_id	_dlt_id	occupation
0	1	Person_1	26	City_A	1708195019.1028914	SVCSe9Sv7ddFxw	None
1	2	Person_2	27	City_A	1708195019.1028914	fx8wMI0m4pIhAg	None

```
Sum of ages:

 ┌──────────┐
│ sum(Age) │
│  int128  │
├──────────┤
│      353 │
└──────────┘
```

- **Question 4**: Merge the 2 generators using the ID column. Calculate the sum of ages of all the people loaded as described above.
    - **215**
    - 266
    - 241
    - 258

Since they have overlapping IDs, some of the records from the first load should be replaced by the ones from the second load.

```
In [ ]: pipeline = dlt.pipeline(    destination='duckdb',
                                    dataset_name='people_dataset' )

        pipeline.run(   people_1(),
                        table_name="people_2",
                        write_disposition="replace")

        pipeline.run(   people_2(),
                        table_name="people_2",
                        write_disposition="merge")

        conn = duckdb.connect(f"{pipeline.pipeline_name}.duckdb")
        conn.sql(f"SET search_path = '{pipeline.dataset_name}'")
        print('Loaded tables: ')
        display(conn.sql("SHOW TABLES"))
```

```
Loaded tables:
┌────────────────────┐
│        name        │
│      varchar       │
├────────────────────┤
│ _dlt_loads         │
│ _dlt_pipeline_state│
│ _dlt_version       │
│ people             │
│ people_2           │
└────────────────────┘
```

```
In [ ]: people = conn.sql("SELECT * FROM people_2").df()
        display(people.head(2))

        sum_ages = conn.sql("SELECT SUM(Age) FROM people_2")
        print("Sum of ages:\n\n", sum_ages)
```

	id	name	age	city	_dlt_load_id	_dlt_id	occupation
0	3	Person_3	33	City_B	1708194986.4371371	qUaDbJnA/CWpfQ	Job_3
1	4	Person_4	34	City_B	1708194986.4371371	t5THk7Qxmf7W3g	Job_4

```
Sum of ages:

 ┌──────────┐
│ sum(Age) │
│  int128  │
├──────────┤
│      213 │
└──────────┘
```