Medallion Pipeline – Bronze Layer (Elhub Energy Data)

Scope of this notebook (BRONZE only):

- Retrieve hourly **production** and **consumption** data from the Elhub Energy Data API
  - Production: all price areas, years **2022–2024**
    (`PRODUCTION_PER_GROUP_MBA_HOUR`)
  - Consumption: all price areas, years **2021–2024**
    (`CONSUMPTION_PER_GROUP_MBA_HOUR`)
- Normalize the raw JSON into flat tabular form with minimal processing
- Store raw-ish data to:
  - Local bronze folder (CSV/Parquet)
  - Cassandra (bronze tables)
  - MongoDB (bronze collections)
- No business rules, no joins, no aggregations → that's for **silver** and **gold**.

In [ ]:
```python
# Imports & Global Config, for the bronze level of this compulsory assignment
import os
import time
import requests
from datetime import datetime, date, timedelta, timezone
import pandas as pd
import matplotlib.pyplot as plt

# Spark and DB switches
SPARK_AVAILABLE = False
try:
    from pyspark.sql import SparkSession
    from pyspark.sql.types import (
        StructType, StructField, StringType,
        TimestampType, DoubleType, IntegerType
    )
    SPARK_AVAILABLE = True
except Exception as e:
    print("Spark unavailable:", e)

print("Pandas version:", pd.__version__)
print("Spark available:", SPARK_AVAILABLE)

# Elhub API basic config
ELHUB_BASE = "https://api.elhub.no/energy-data/v0"
ENTITY = "price-areas"

DATASETS = {
    "production":  "PRODUCTION_PER_GROUP_MBA_HOUR",
    "consumption": "CONSUMPTION_PER_GROUP_MBA_HOUR",
}
```

```
ELHUB_AUTH_TOKEN = None

# Time ranges (bronze)
PROD_START = date(2022, 1, 1)
PROD_END   = date(2024, 12, 31)

CONS_START = date(2021, 1, 1)
CONS_END   = date(2024, 12, 31)

# Setting the local storage paths for the bronze level
BRONZE_ROOT = "../Data_Assignment_4/bronze"
BRONZE_PROD_DIR = os.path.join(BRONZE_ROOT, "production")
BRONZE_CONS_DIR = os.path.join(BRONZE_ROOT, "consumption")

os.makedirs(BRONZE_PROD_DIR, exist_ok=True)
os.makedirs(BRONZE_CONS_DIR, exist_ok=True)

# Cassandra & Mongo config (bronze)
CASSANDRA_HOSTS = "127.0.0.1"
CASSANDRA_KEYSPACE = "energy"
CASSANDRA_TABLE_BRONZE_PROD = "bronze_production_hourly"
CASSANDRA_TABLE_BRONZE_CONS = "bronze_consumption_hourly"

MONGO_URI = "mongodb://localhost:27017"
MONGO_DATABASE = "ind320"
MONGO_COLL_BRONZE_PROD = "bronze_production_hourly"
MONGO_COLL_BRONZE_CONS = "bronze_consumption_hourly"
```

```
Spark unavailable: No module named 'pyspark.errors'
Pandas version: 2.3.3
Spark available: False
```

In [ ]:

```
# We set the API headers here
def elhub_headers():
    headers = {
        "Accept": "application/vnd.api+json, application/json",
    }
    if ELHUB_AUTH_TOKEN:
        headers["Authorization"] = ELHUB_AUTH_TOKEN
    return headers

# We return the ISO 8601 UTC format for datetime objects
def iso_utc(d: datetime) -> str:
    if d.tzinfo is None:
        d = d.replace(tzinfo=timezone.utc)
    else:
        d = d.astimezone(timezone.utc)
    return d.isoformat().replace("+00:00", "Z")

# Then we ensure that the response is JSON
def ensure_json(r: requests.Response):
    ct = (r.headers.get("Content-Type") or "").lower()
    if "json" not in ct:
        raise ValueError(f"Non-JSON response (status {r.status_code}) from {r.url}"
    return r.json()
```

```python
# Here we fetch data from Elhub for a given dataset and time interval
def fetch_interval_json(dataset_key: str, start_dt: datetime, end_dt: datetime):
    dataset = DATASETS[dataset_key]
    params = {
        "dataset": dataset,
        "startTime": iso_utc(start_dt),
        "endTime": iso_utc(end_dt),
        "page[size]": 10000,
    }
    url = f"{ELHUB_BASE}/{ENTITY}"
    r = requests.get(url, headers=elhub_headers(), params=params, timeout=120)
    if r.status_code != 200:
        raise RuntimeError(
            f"Elhub API error {r.status_code} for {dataset_key}: {r.text[:500]}"
        )
    payload = ensure_json(r)
    return payload
```

In [ ]:

```python
# Now we normalize the Elhub JSON payload into a pandas DataFrame
def normalize_elhub_items(payload: dict, kind: str) -> pd.DataFrame:
    data = payload.get("data") or []
    rows = []

    for item in data:
        attrs = (item or {}).get("attributes") or {}
        prod_series = attrs.get("productionPerGroupMbaHour") or []
        cons_series = attrs.get("consumptionPerGroupMbaHour") or []

        # Generic handler for series lists
        def handle_series(series, is_production: bool):
            for e in series:
                row = {}
                row["priceArea"] = e.get("priceArea") or attrs.get("name")

                if is_production:
                    row["group"] = e.get("productionGroup")
                    row["quantityKwh"] = e.get("quantityKwh")
                    row["countMeteringPoints"] = None
                else:
                    row["group"] = e.get("consumptionGroup")
                    row["quantityKwh"] = e.get("quantityKwh")
                    row["countMeteringPoints"] = e.get("countMeteringPoints")

                row["startTime"] = e.get("startTime")
                row["endTime"] = e.get("endTime")
                rows.append(row)

        if prod_series and kind == "production":
            handle_series(prod_series, is_production=True)
        if cons_series and kind == "consumption":
            handle_series(cons_series, is_production=False)

        # Fallback: sometimes a single timeseries dict instead of list
        ts = attrs.get("timeseries") or {}
        if ts and not (prod_series or cons_series):
```

```
            row = {}
            row["priceArea"] = ts.get("priceArea") or attrs.get("name")
            if kind == "production":
                row["group"] = ts.get("productionGroup")
                row["quantityKwh"] = ts.get("quantityKwh")
                row["countMeteringPoints"] = None
            else:
                row["group"] = ts.get("consumptionGroup")
                row["quantityKwh"] = ts.get("quantityKwh")
                row["countMeteringPoints"] = ts.get("countMeteringPoints")

            row["startTime"] = ts.get("startTime")
            row["endTime"] = ts.get("endTime")
            rows.append(row)

    if not rows:
        return pd.DataFrame(
            columns=["priceArea", "group", "startTime", "endTime",
                     "quantityKwh", "countMeteringPoints"]
        )

    df = pd.DataFrame(rows)

    # Light typing (still bronze-friendly)
    df["startTime"] = pd.to_datetime(df["startTime"], errors="coerce", utc=True)
    df["endTime"] = pd.to_datetime(df["endTime"], errors="coerce", utc=True)
    df["quantityKwh"] = pd.to_numeric(df["quantityKwh"], errors="coerce")
    if "countMeteringPoints" in df.columns:
        df["countMeteringPoints"] = pd.to_numeric(df["countMeteringPoints"], errors

    return df
```

In [ ]:

```
# Helper to iterate months in a date range
def iter_months(start_d: date, end_d: date):
    cur = date(start_d.year, start_d.month, 1)
    while cur <= end_d:
        # next month
        if cur.month == 12:
            nxt = date(cur.year + 1, 1, 1)
        else:
            nxt = date(cur.year, cur.month + 1, 1)
        yield cur, min(nxt - timedelta(days=1), end_d)
        cur = nxt

# We fetch & normalize data for given kind ('production' or 'consumption')
def fetch_dataset_range(kind: str, start_d: date, end_d: date, pause: float = 1.0):
    assert kind in ("production", "consumption")
    frames = []
    failures = []

    for month_start, month_end in iter_months(start_d, end_d):
        start_dt = datetime.combine(month_start, datetime.min.time()).replace(tzinf
        end_dt = datetime.combine(month_end + timedelta(days=1),
                                  datetime.min.time()).replace(tzinfo=timezone.utc)

        label = f"{month_start:%Y-%m}"
```

```
        print(f"Fetching {kind} for {label} ({start_dt} → {end_dt}) ...", end=" ")

        try:
            payload = fetch_interval_json(kind, start_dt, end_dt)
            df_month = normalize_elhub_items(payload, kind=kind)
            print(f"{len(df_month):,} rows")
            if not df_month.empty:
                frames.append(df_month)
        except Exception as e:
            print("FAILED:", e)
            failures.append((label, str(e)))

        time.sleep(pause)

    if frames:
        df_all = pd.concat(frames, ignore_index=True)
    else:
        df_all = pd.DataFrame(
            columns=["priceArea", "group", "startTime", "endTime",
                     "quantityKwh", "countMeteringPoints"]
        )

    return df_all, failures
```

In [ ]:

```
# Spark-related functions for the bronze level
def build_spark(app_name="Elhub Bronze Ingestion"):
    if not SPARK_AVAILABLE:
        raise RuntimeError("Spark is not available in this environment.")
    spark = (
        SparkSession.builder
        .appName(app_name)
        .config("spark.cassandra.connection.host", CASSANDRA_HOSTS)
        .config("spark.mongodb.write.connection.uri", MONGO_URI)
        .getOrCreate()
    )
    return spark


def to_spark(df: pd.DataFrame):
    if not SPARK_AVAILABLE:
        raise RuntimeError("Spark is not available.")
    schema = StructType([
        StructField("priceArea", StringType(), True),
        StructField("group", StringType(), True),
        StructField("startTime", TimestampType(), True),
        StructField("endTime", TimestampType(), True),
        StructField("quantityKwh", DoubleType(), True),
        StructField("countMeteringPoints", IntegerType(), True),
    ])
    spark = build_spark()
    return spark.createDataFrame(df, schema=schema)


def write_bronze_to_cassandra(sdf, table_name: str):
    (
        sdf.write
```

```
        .format("org.apache.spark.sql.cassandra")
        .mode("append")
        .options(keyspace=CASSANDRA_KEYSPACE, table=table_name)
        .save()
    )


def write_bronze_to_mongo(sdf, collection_name: str):
    (
        sdf.write
        .format("mongodb")
        .mode("append")
        .option("database", MONGO_DATABASE)
        .option("collection", collection_name)
        .save()
    )
```

In [ ]:

```
# Now we perform the bronze ingestion: PRODUCTION 2022-2024
prod_df, prod_fail = fetch_dataset_range("production", PROD_START, PROD_END)

print("\nProduction fetch done.")
print("Rows:", len(prod_df))
print("Failures:", prod_fail[:5])

# And we save the production data locally, partitioned by year
if not prod_df.empty:
    prod_df = prod_df.copy()
    prod_df["year"] = prod_df["startTime"].dt.year
    print("Years present in production:", sorted(prod_df["year"].dropna().unique()))

    for year, df_y in prod_df.groupby("year"):
        out_path = os.path.join(BRONZE_PROD_DIR, f"production_{int(year)}.csv")
        df_y.drop(columns=["year"]).to_csv(out_path, index=False)
        print(f"Saved {len(df_y):,} rows to {out_path}")
else:
    print("⚠ No production rows fetched – nothing saved.")
```

```
Fetching production for 2022-01 (2022-01-01 00:00:00+00:00 → 2022-02-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-02 (2022-02-01 00:00:00+00:00 → 2022-03-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-03 (2022-03-01 00:00:00+00:00 → 2022-04-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-04 (2022-04-01 00:00:00+00:00 → 2022-05-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-05 (2022-05-01 00:00:00+00:00 → 2022-06-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-06 (2022-06-01 00:00:00+00:00 → 2022-07-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-07 (2022-07-01 00:00:00+00:00 → 2022-08-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-08 (2022-08-01 00:00:00+00:00 → 2022-09-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-09 (2022-09-01 00:00:00+00:00 → 2022-10-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-10 (2022-10-01 00:00:00+00:00 → 2022-11-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-11 (2022-11-01 00:00:00+00:00 → 2022-12-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2022-12 (2022-12-01 00:00:00+00:00 → 2023-01-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-01 (2023-01-01 00:00:00+00:00 → 2023-02-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-02 (2023-02-01 00:00:00+00:00 → 2023-03-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-03 (2023-03-01 00:00:00+00:00 → 2023-04-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-04 (2023-04-01 00:00:00+00:00 → 2023-05-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-05 (2023-05-01 00:00:00+00:00 → 2023-06-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-06 (2023-06-01 00:00:00+00:00 → 2023-07-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-07 (2023-07-01 00:00:00+00:00 → 2023-08-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-08 (2023-08-01 00:00:00+00:00 → 2023-09-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-09 (2023-09-01 00:00:00+00:00 → 2023-10-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-10 (2023-10-01 00:00:00+00:00 → 2023-11-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-11 (2023-11-01 00:00:00+00:00 → 2023-12-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2023-12 (2023-12-01 00:00:00+00:00 → 2024-01-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-01 (2024-01-01 00:00:00+00:00 → 2024-02-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-02 (2024-02-01 00:00:00+00:00 → 2024-03-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-03 (2024-03-01 00:00:00+00:00 → 2024-04-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-04 (2024-04-01 00:00:00+00:00 → 2024-05-01 00:00:00+00:
00) ... 18,851 rows
```

```
Fetching production for 2024-05 (2024-05-01 00:00:00+00:00 → 2024-06-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-06 (2024-06-01 00:00:00+00:00 → 2024-07-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-07 (2024-07-01 00:00:00+00:00 → 2024-08-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-08 (2024-08-01 00:00:00+00:00 → 2024-09-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-09 (2024-09-01 00:00:00+00:00 → 2024-10-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-10 (2024-10-01 00:00:00+00:00 → 2024-11-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-11 (2024-11-01 00:00:00+00:00 → 2024-12-01 00:00:00+00:
00) ... 18,851 rows
Fetching production for 2024-12 (2024-12-01 00:00:00+00:00 → 2025-01-01 00:00:00+00:
00) ... 18,851 rows

Production fetch done.
Rows: 678636
Failures: []
Years present in production: [np.int32(2025)]
Saved 678,636 rows to ../Data_Assignment_4/bronze\production\production_2025.csv
```

In [ ]:
```python
# We do some quick checks on the production DataFrame
print("prod_df dtypes:\n", prod_df.dtypes)
print("\nstartTime sample:")
print(prod_df["startTime"].head())

if not prod_df.empty:
    print("\nUnique years in startTime:")
    print(sorted(prod_df["startTime"].dt.year.dropna().unique()))
```

```
prod_df dtypes:
 priceArea                          object
group                              object
quantityKwh                       float64
countMeteringPoints               float64
startTime              datetime64[ns, UTC]
endTime                datetime64[ns, UTC]
year                                int32
dtype: object

startTime sample:
0   2025-10-22 19:00:00+00:00
1   2025-10-22 20:00:00+00:00
2   2025-10-22 21:00:00+00:00
3   2025-10-22 22:00:00+00:00
4   2025-10-22 23:00:00+00:00
Name: startTime, dtype: datetime64[ns, UTC]

Unique years in startTime:
[np.int32(2025)]
```

In [ ]:
```python
# and we do the same for the CONSUMPTION data: 2021–2024
cons_df, cons_fail = fetch_dataset_range("consumption", CONS_START, CONS_END)
```

```python
print("\nConsumption fetch done.")
print("Rows:", len(cons_df))
print("Failures:", cons_fail[:5])

# And we save the consumption data locally, partitioned by year
if not cons_df.empty:
    cons_df = cons_df.copy()
    cons_df["year"] = cons_df["startTime"].dt.year
    print("Years present in consumption:", sorted(cons_df["year"].dropna().unique()

    for year, df_y in cons_df.groupby("year"):
        out_path = os.path.join(BRONZE_CONS_DIR, f"consumption_{int(year)}.csv")
        df_y.drop(columns=["year"]).to_csv(out_path, index=False)
        print(f"Saved {len(df_y):,} rows to {out_path}")
else:
    print("⚠ No consumption rows fetched – nothing saved.")
```

```
Fetching consumption for 2021-01 (2021-01-01 00:00:00+00:00 → 2021-02-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-02 (2021-02-01 00:00:00+00:00 → 2021-03-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-03 (2021-03-01 00:00:00+00:00 → 2021-04-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-04 (2021-04-01 00:00:00+00:00 → 2021-05-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-05 (2021-05-01 00:00:00+00:00 → 2021-06-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-06 (2021-06-01 00:00:00+00:00 → 2021-07-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-07 (2021-07-01 00:00:00+00:00 → 2021-08-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-08 (2021-08-01 00:00:00+00:00 → 2021-09-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-09 (2021-09-01 00:00:00+00:00 → 2021-10-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-10 (2021-10-01 00:00:00+00:00 → 2021-11-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-11 (2021-11-01 00:00:00+00:00 → 2021-12-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2021-12 (2021-12-01 00:00:00+00:00 → 2022-01-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-01 (2022-01-01 00:00:00+00:00 → 2022-02-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-02 (2022-02-01 00:00:00+00:00 → 2022-03-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-03 (2022-03-01 00:00:00+00:00 → 2022-04-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-04 (2022-04-01 00:00:00+00:00 → 2022-05-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-05 (2022-05-01 00:00:00+00:00 → 2022-06-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-06 (2022-06-01 00:00:00+00:00 → 2022-07-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-07 (2022-07-01 00:00:00+00:00 → 2022-08-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-08 (2022-08-01 00:00:00+00:00 → 2022-09-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-09 (2022-09-01 00:00:00+00:00 → 2022-10-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-10 (2022-10-01 00:00:00+00:00 → 2022-11-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-11 (2022-11-01 00:00:00+00:00 → 2022-12-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2022-12 (2022-12-01 00:00:00+00:00 → 2023-01-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-01 (2023-01-01 00:00:00+00:00 → 2023-02-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-02 (2023-02-01 00:00:00+00:00 → 2023-03-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-03 (2023-03-01 00:00:00+00:00 → 2023-04-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-04 (2023-04-01 00:00:00+00:00 → 2023-05-01 00:00:00+0
0:00) ... 18,100 rows
```

```
Fetching consumption for 2023-05 (2023-05-01 00:00:00+00:00 → 2023-06-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-06 (2023-06-01 00:00:00+00:00 → 2023-07-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-07 (2023-07-01 00:00:00+00:00 → 2023-08-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-08 (2023-08-01 00:00:00+00:00 → 2023-09-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-09 (2023-09-01 00:00:00+00:00 → 2023-10-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-10 (2023-10-01 00:00:00+00:00 → 2023-11-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-11 (2023-11-01 00:00:00+00:00 → 2023-12-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2023-12 (2023-12-01 00:00:00+00:00 → 2024-01-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-01 (2024-01-01 00:00:00+00:00 → 2024-02-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-02 (2024-02-01 00:00:00+00:00 → 2024-03-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-03 (2024-03-01 00:00:00+00:00 → 2024-04-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-04 (2024-04-01 00:00:00+00:00 → 2024-05-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-05 (2024-05-01 00:00:00+00:00 → 2024-06-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-06 (2024-06-01 00:00:00+00:00 → 2024-07-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-07 (2024-07-01 00:00:00+00:00 → 2024-08-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-08 (2024-08-01 00:00:00+00:00 → 2024-09-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-09 (2024-09-01 00:00:00+00:00 → 2024-10-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-10 (2024-10-01 00:00:00+00:00 → 2024-11-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-11 (2024-11-01 00:00:00+00:00 → 2024-12-01 00:00:00+0
0:00) ... 18,100 rows
Fetching consumption for 2024-12 (2024-12-01 00:00:00+00:00 → 2025-01-01 00:00:00+0
0:00) ... 18,100 rows

Consumption fetch done.
Rows: 868800
Failures: []
Years present in consumption: [np.int32(2025)]
Saved 868,800 rows to ../Data_Assignment_4/bronze\consumption\consumption_2025.csv
```

In [ ]:

```python
# We do some quick checks on the consumption DataFrame
# Checking for consumption DataFrame
if not prod_df.empty:
    print("Production time span:",
          prod_df["startTime"].min(), "→", prod_df["startTime"].max())
    print(prod_df.head())

if not cons_df.empty:
    print("Consumption time span:",
```

```
          cons_df["startTime"].min(), "→", cons_df["startTime"].max())
    print(cons_df.head())
```

```
Production time span: 2025-10-22 19:00:00+00:00 → 2025-11-21 22:00:00+00:00
  priceArea  group  quantityKwh  countMeteringPoints  \
0       NO1  hydro    2332513.0                  NaN
1       NO1  hydro    2281520.8                  NaN
2       NO1  hydro    2308167.5                  NaN
3       NO1  hydro    2253987.5                  NaN
4       NO1  hydro    2241552.0                  NaN

                  startTime                   endTime  year
0 2025-10-22 19:00:00+00:00 2025-10-22 20:00:00+00:00  2025
1 2025-10-22 20:00:00+00:00 2025-10-22 21:00:00+00:00  2025
2 2025-10-22 21:00:00+00:00 2025-10-22 22:00:00+00:00  2025
3 2025-10-22 22:00:00+00:00 2025-10-22 23:00:00+00:00  2025
4 2025-10-22 23:00:00+00:00 2025-10-23 00:00:00+00:00  2025
Consumption time span: 2025-10-22 19:00:00+00:00 → 2025-11-21 22:00:00+00:00
  priceArea  group  quantityKwh  countMeteringPoints  \
0       NO1  cabin    67069.060                  NaN
1       NO1  cabin    66767.110                  NaN
2       NO1  cabin    65964.305                  NaN
3       NO1  cabin    65159.824                  NaN
4       NO1  cabin    64141.660                  NaN

                  startTime                   endTime  year
0 2025-10-22 19:00:00+00:00 2025-10-22 20:00:00+00:00  2025
1 2025-10-22 20:00:00+00:00 2025-10-22 21:00:00+00:00  2025
2 2025-10-22 21:00:00+00:00 2025-10-22 22:00:00+00:00  2025
3 2025-10-22 22:00:00+00:00 2025-10-22 23:00:00+00:00  2025
4 2025-10-22 23:00:00+00:00 2025-10-23 00:00:00+00:00  2025
```