SERIÖS GROUP

Generating Realistic Gold Layer Mock Data in Microsoft Fabric

25/11/25

SERIÖS GROUP

Contents

SERIÖS GROUP

Generating Realistic Gold Layer Mock Data in Microsoft Fabric

Why Mock Data?

In the world of data architecture and analytics, mock data plays a crucial role in helping teams develop, test, and validate data models and visualisations, especially before real data becomes available. We regularly works with clients building complex solutions on modern cloud platforms, and one challenge we often see early in the project lifecycle is the lack of realistic test data.
To test end-to-end transformation logic, pipeline performance and dashboards, we can generate mock data to simulate 'source' data and then process it through the transformation layers. However, this relies on all transformation pipelines being in place. To enable more parallel development, we can also directly generate mock data at the gold layer – tables that will be consumed by our analytics tools.

In this scenario, mock data allows us to:

- Validate how schemas and relationships are set up.
- Test performance and scalability of dashboards.
- Simulate edge cases and data anomalies.
- Enable front-end development and dashboard prototyping.

But not all mock data is created equal. Random values can be useful for basic testing, but realistic mock data mimics the structure, distribution, volume, and logic of production datasets.

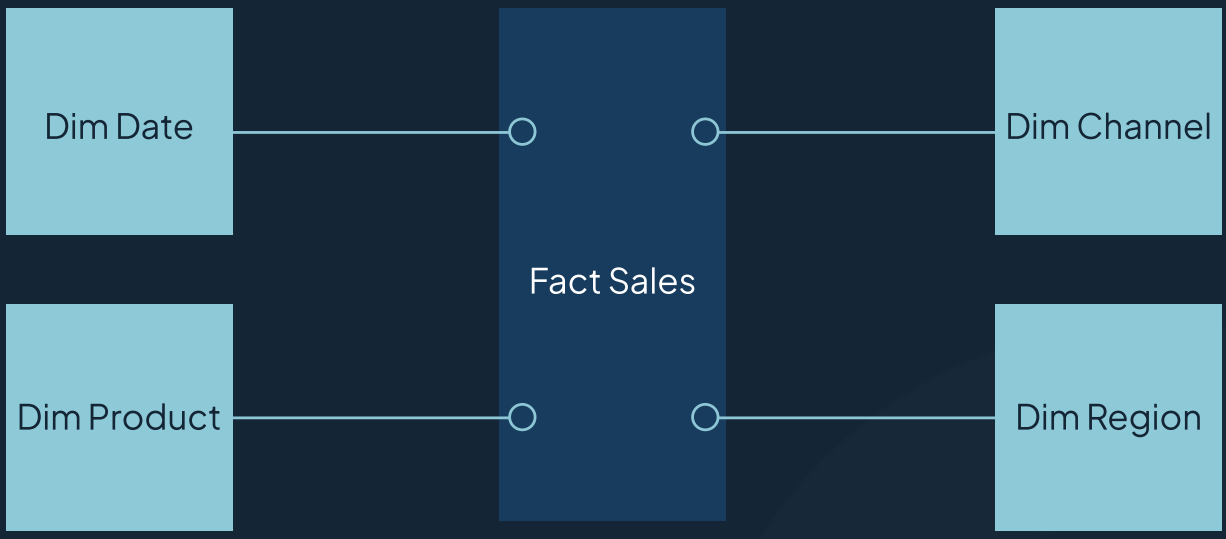
How to Generate Realistic Mock Data

Principal Data Solutions Architect at Seriös Group, Tom Legge, developed a set of PySpark notebooks that generated realistic mock data directly into Lakehouse tables in Microsoft Fabric. The final data model was a Kimball dimensional data model. Some of the dimension tables were already available in the Lakehouse (such as data dimensions, organisational hierarchy dimensions). The challenge of this mock data generation was to integrate some 'real' dimension data with simulated fact table measures. We have recreated the process here with a made-up scenario.

SERIÖS GROUP

The Basic Outline

Consider a simple dimensional data model in which a fact table aggregates sales data at a monthly grain:



Step 1: Create our Dimension Tables

We can manually build the dimension tables, or they may already exist as conformed dimensions. Either way, we'll need to know the range of surrogate key values for each dimension.

Step 2: Define our Fact Table Schema

We need columns named after the surrogate keys of the dimension tables and columns for what we want to measure in the fact table (e.g. quantity sold, unit price, and total sales):

```
sk_channel
sk_region
sk_product
sk_date
quantity_sold
unit_price
total_sales
```

SERIÖS GROUP

Step 3: Craft our Initial Notebook

Our PySpark notebook in Fabric will generate random data for each of our target fact table columns. We've used native PySpark dataframes rather than pandas dataframes, but if you wish to use pandas we recommend using the Pandas API on Spark, as it supports distributed workloads. We tested the use of the popular Faker library but found its performance wasn't close to using native pyspark data frames, even when being careful to ensure the code was executed across all worker nodes of the spark cluster. In our scenario, this was a problem as we needed to generate hundreds of millions of rows.

```
# Required imports for PySpark
from pyspark.sql import functions as F
from pyspark.sql import types as T
import random

# Configuration parameters

num_rows = 40000  # Increased for better distribution analysis

# Define value ranges
date_values = [202301, 202302, 202303, 202304, 202305, 202306,
               202307, 202308, 202309, 202310, 202311, 202312]

# Generate base data with uniform random values
df = spark.range(num_rows).select(
    # Channel: 1-2
    (F.rand() * 2 + 1).cast("integer").alias("sk_channel"),

    # Region: 1-50
    (F.rand() * 50 + 1).cast("integer").alias("sk_region"),

    # Product: 1-100
    (F.rand() * 100 + 1).cast("integer").alias("sk_product"),

    # Date: random choice from date_values
    F.array(*[F.lit(x) for x in date_values])[F.floor(F.rand() * len(date_values))].alias("sk_date"),

    # Quantity: 1-19
    (F.rand() * 19 + 1).cast("integer").alias("quantity_sold"),

    # Unit price: 5.0-100.0
    (F.rand() * 95 + 5).cast("double").alias("unit_price")
)

# Calculate total sales
df = df.withColumn("total_sales", F.round(F.col("quantity_sold") * F.col("unit_price"), 2))

# Remove duplicates based on surrogate keys
surrogate_keys = ["sk_channel", "sk_region", "sk_product", "sk_date"]
df = df.dropDuplicates(surrogate_keys)


# Write to Lakehouse table
schema_name = "dbo"
table_name = "sales_fact"
df.write.format("delta").mode("overwrite").option("overwriteSchema", True).saveAsTable(f"{schema_name}.{table_name}")
print(F"{table_name} table generation completed.")
```

1 https://spark.apache.org/docs/latest/api/python/getting_started/quickstart_ps.html

SERIÖS GROUP

Lets Make It Better

Our initial script allows immediate use of the mock data in downstream analytics, dashboards, or machine learning workflows. However, it doesn't yet meet all our requirements, so let's improve it.

Skewing Data

To mimic real data, in which surrogate key values will not be uniform, we can use several methods to skew our randomly generated values.

Method 1: skew our list of values by handcrafting a list

```
# skew our channel surrogate keys towards one value
sk_channel_values = [1,1,1,1,1,1,2,3,4]

# Generate base data with uniform random values
df = spark.range(num_rows).select(
    # Channel: 1-4
    F.array(*[F.lit(x) for x in sk_channel_values])[F.floor(F.rand() * len(sk_channel_values))].alias("sk_channel"))
```

A simple method is to insert multiples of the surrogate key value we want to skew towards. In the above example, we use a list (sk_channel_values) with more ones than twos, threes and fours and then randomly select from this list during the data generation step.

Method 2: assign a probability to each possible outcome

```
# Channel weights
channel_weights = [0.50, 0.10, 0.05, 0.35]
channel_cumulative = [sum(channel_weights[:i+1]) for i in range(len(channel_weights))]


# Generate base data with uniform random values
df = spark.range(num_rows).withColumn("rand_val", F.rand()).select(
    # Channel: weighted random (1-4) using cumulative probabilities
    F.when(F.col("rand_val") < channel_cumulative[0], 1)
     .when(F.col("rand_val") < channel_cumulative[1], 2)
     .when(F.col("rand_val") < channel_cumulative[2], 3)
     .otherwise(4).cast("integer").alias("sk_channel"),
```

We can get a little smarter by assigning a probability to each of the 4 possible values. Whilst it is more code, it is easier to adjust the distributions in future.

Completeness

You'll notice in our code we generate lots of rows of data, some of which turn out to be duplicates. We then remove the duplicates in a later step. The alternative approach is to iterate over every combination of surrogate key until we have a complete set. Whilst the second option guarantees completeness, we found the approach to be less adaptable to new surrogate keys, but there may be scenarios where guaranteed completeness is required.

Business Logic

It is often the case that some combinations of surrogate keys don't make sense and would result in unrealistic mock data. In our current example, consider products only sold online (hypothetically represented by sk_channel = 1). We wouldn't expect to see combinations of those products with other sk_channel values.

We can account for these rules by ensuring every instance of those products is mapped to sk_channel = 1. We should be careful to perform this step before removing duplicates.

```
# Apply Business Rules
vals = [1, 5, 15, 16, 45]
df = df.withColumn("sk_channel", F.when(F.col("sk_product").isin(*vals), F.lit(1)).otherwise(F.col("sk_channel")))
```

Handling 'real' Dimensional Data

Consider the scenario of a pre-existing dimension table for 'regions', in which we only want to generate mock data for a subset of the regions. If our subset is small, we can put our values in a list directly in the code, like we do with datekeys. But if larger, this approach is unwieldy. Instead, we store the desired list of region surrogate keys in a csv file and import into a python list:

```
# Actual region sk values
df_region = spark.read.csv('Files/region_mapping.csv', header=True)
sk_region_list = list(int(x) for x in df_region.select('sk_region_actual').rdd.flatMap(lambda x: x).collect())
```

We then randomly selected from our list when generating mock data.

```
# Region: 1-50
F.array(*[F.lit(x) for x in sk_region_list])[F.floor(F.rand() * len(date_values))].alias("sk_region"),
```

Introducing Bad Data

We can further modify the mock data output by introducing “bad” or unexpected data values, such as negative numbers or nulls. To do this, we can introduce such data to a subset of our generated records

```
# Add bad or unexpected data
# Add nulls for some fraction of sk_region
df = df.withColumn(
    "sk_region",
    F.when(
        F.rand() < 0.5,
        None
    ).otherwise(F.col("sk_region"))
)
```

Conclusion

Mock data isn’t just a filler; it’s a powerful way to speed up development and improve quality from the start. At Seriös Group, we believe in building robust, scalable, and smart data solutions, and realistic mock data is a key part of that journey.

If you need help designing or testing your data architecture, get in touch via our website www.seriosgroup.com.