```
#check that java is installed
!java -version
```

```
openjdk version "11.0.28" 2025-07-15
OpenJDK Runtime Environment (build 11.0.28+6-post-Ubuntu-1ubuntu122.04.1)
OpenJDK 64-Bit Server VM (build 11.0.28+6-post-Ubuntu-1ubuntu122.04.1, mixed mode, sharing)
```

```
#install pyspark
!pip install pyspark
```

```
Requirement already satisfied: pyspark in /usr/local/lib/python3.11/dist-packages (3.5.1)
Requirement already satisfied: py4j==0.10.9.7 in /usr/local/lib/python3.11/dist-packages (from pyspark) (0.10.9.7)
```

```
# =============================
# STEP 1: Install & Import
# =============================
!pip install pyspark --quiet

from pyspark.sql import SparkSession
from pyspark.sql.functions import lit, rand, floor, concat_ws, explode, array, col, count
import time

# Create Spark Session
spark = SparkSession.builder \
    .appName("SaltingExample") \
    .config("spark.sql.shuffle.partitions", 8) \
    .getOrCreate()
```

```
# =============================
# STEP 2: Create Skewed Sales Data
# =============================
# Fact table: sales (skew on customer_id=1)
num_rows = 5_000_000
skew_ratio = 0.8  # 80% rows belong to customer_id=1

# Create skewed data
data_skewed = []
for i in range(num_rows):
    if i < num_rows * skew_ratio:
        data_skewed.append((1, i % 100, float(i % 500)))  # customer_id=1
    else:
        cust_id = i % 1000 + 2  # other customers
        data_skewed.append((cust_id, i % 100, float(i % 500)))

sales_df = spark.createDataFrame(data_skewed, ["customer_id", "product_id", "amount"])

# Dimension table: customers
customers = [(1, "VIP Customer")] + [(i, f"Customer_{i}") for i in range(2, 1002)]
customers_df = spark.createDataFrame(customers, ["customer_id", "customer_name"])
```

```
# =============================
# STEP 3: Show Skew Distribution
# =============================
print("📊 Customer Distribution (Skewed):")
sales_df.groupBy("customer_id").agg(count("*").alias("cnt")) \
    .orderBy(col("cnt").desc()) \
    .show(5)
```

```
📊 Customer Distribution (Skewed):
+-----------+-------+
|customer_id|    cnt|
+-----------+-------+
|          1|4000000|
|          2|   1000|
|         12|   1000|
|         26|   1000|
|         28|   1000|
+-----------+-------+
only showing top 5 rows
```

```
# ============================
# STEP 4: Skewed Join (No Salting)
# ============================
start = time.time()
joined_skewed = sales_df.join(customers_df, "customer_id")
joined_skewed.count()  # Force execution
end = time.time()

print(f"⏳ Join Time (Skewed, No Salting): {round(end - start, 2)} sec")
```

```
⏳ Join Time (Skewed, No Salting): 10.88 sec
```

```
# Partition size distribution before salting
partition_sizes_skewed = joined_skewed.rdd.mapPartitions(lambda it: [sum(1 for _ in it)]).collect()
print("\n Partition Sizes (Skewed):", partition_sizes_skewed)
```

```

 Partition Sizes (Skewed): [2499584, 2500416]
```

```
# ============================
# STEP 5: Salting the Join
# ============================
salt_size = 10  # Break heavy key into 10 parts

# Add salt to customers table
customers_salted = customers_df.withColumn("salt", floor(rand() * salt_size)) \
    .withColumn("join_key", concat_ws("_", col("customer_id"), col("salt")))

# Add all salt values for sales table
sales_salted = sales_df.withColumn("salt", explode(array([lit(i) for i in range(salt_size)]))) \
    .withColumn("join_key", concat_ws("_", col("customer_id"), col("salt")))

# Join on salted key
start = time.time()
joined_salted = sales_salted.join(customers_salted, "join_key") \
    .drop("join_key", "salt")
joined_salted.count()  # Force execution
end = time.time()

print(f"⚡ Join Time (With Salting): {round(end - start, 2)} sec")
```

```
⚡ Join Time (With Salting): 47.13 sec
```

Double-click (or enter) to edit

```
# ============================
# STEP 6: Cleanup
# ============================
spark.stop()
```

```
Start coding or generate with AI.
```

Data skew You’re grouping/joining by a key (here: user_id). One key (say A) has way more rows than the others (B, C, D). In distributed systems (like Spark), rows with the same key go to the same partition for a join/groupBy.

If A has most of the rows, one partition gets overloaded → slow task or OOM. Your tiny dataset (skew):

Key A = 5 rows (heavy) B = 2, C = 1, D = 2 (light)

```
import pandas as pd

# Create skewed demo data
data = {
    "user_id": ["A", "A", "A", "A", "A", "B", "B", "C", "D", "D"],
    "purchase": [120, 80, 200, 150, 90, 300, 250, 400, 500, 600]
}

df = pd.DataFrame(data)
```

```
print("📊 Skewed Demo DataFrame:")
print(df)

# Check distribution
print("\n🔍 Value counts for user_id:")
print(df["user_id"].value_counts())
```

```
📊 Skewed Demo DataFrame:
  user_id  purchase
0       A       120
1       A        80
2       A       200
3       A       150
4       A        90
5       B       300
6       B       250
7       C       400
8       D       500
9       D       600

🔍 Value counts for user_id:
user_id
A    5
B    2
D    2
C    1
Name: count, dtype: int64
```

What is salting? Idea: Split a heavy key (A) into several subkeys (A_0, A_1, A_2, …) so its rows can be processed in parallel by multiple partitions. You did exactly this by adding: salt_column = random integer in [0, 2] (so 0, 1, or 2) user_id_salt = user_id + "_" + salt_column (e.g., A_0, A_1, A_2) Now your A rows no longer all share the same key—they're spread across A_0, A_1, A_2.

```
# Add salt_column with random integers [0, 2]
import numpy as np

df['salt_column'] = np.random.randint(0, 3, size=len(df))

print("📊 Skewed DataFrame with Salt Column:")
print(df)
```

```
📊 Skewed DataFrame with Salt Column:
  user_id  purchase  salt_column
0       A       120            0
1       A        80            1
2       A       200            1
3       A       150            2
4       A        90            0
5       B       300            1
6       B       250            1
7       C       400            0
8       D       500            2
9       D       600            2
```

If a join/groupBy uses user_id_salt as the key, A's workload is now split across 3 keys → multiple partitions → no single hotspot.

Why does this help? In Spark (and similar systems), when you join or groupBy: Rows are hashed on the key and sent to partitions. Without salting: all A rows hash to the same partition. With salting: A_0 may hash to partition 1, A_1 to partition 3, A_2 to partition 5 → load is spread.

```
# Create user_id_salt by concatenating user_id and salt_column
df["user_id_salt"] = df["user_id"].astype(str) + "_" + df["salt_column"].astype(str)

print("📊 Skewed DataFrame with Salt Column and User ID Salt:")
print(df)
```

```
📊 Skewed DataFrame with Salt Column and User ID Salt:
  user_id  purchase  salt_column join_key user_salt user_id_salt
0       A       120            0      A_0       A_0          A_0
1       A        80            1      A_1       A_1          A_1
2       A       200            1      A_1       A_1          A_1
3       A       150            2      A_2       A_2          A_2
4       A        90            0      A_0       A_0          A_0
5       B       300            1      B_1       B_1          B_1
6       B       250            1      B_1       B_1          B_1
7       C       400            0      C_0       C_0          C_0
8       D       500            2      D_2       D_2          D_2
```

```
9       D       600            2      D_2       D_2          D_2
```

```
df.drop(columns=['join_key','user_salt'], inplace=True)
print("📊 Skewed DataFrame with Salt Column and User ID Salt:")
print(df)
```

```
📊 Skewed DataFrame with Salt Column and User ID Salt:
  user_id  purchase  salt_column user_id_salt
0       A       120            0          A_0
1       A        80            1          A_1
2       A       200            1          A_1
3       A       150            2          A_2
4       A        90            0          A_0
5       B       300            1          B_1
6       B       250            1          B_1
7       C       400            0          C_0
8       D       500            2          D_2
9       D       600            2          D_2
```

```
# Group by user_id_salt
grouped_df2 = df.groupby("user_id_salt")["purchase"].sum()


print("📊 Grouped by user_id_salt:")
print(grouped_df2)
```

```
📊 Grouped by user_id_salt:
user_id_salt
A_0     210
A_1     280
A_2     150
B_1     550
C_0     400
D_2    1100
Name: purchase, dtype: int64
```

```
Start coding or generate with AI.
```