Resolving OOM using Salting — Detailed Notes

1️⃣ What is "Salting" in Data Engineering?

Definition: Salting is a technique where you add a random or evenly distributed extra key ("salt") to an existing join or groupBy key to distribute data more evenly across partitions.

Purpose: Prevent data skew — when a few keys have disproportionately large amounts of data compared to others.

2️⃣ Why Data Skew Causes OOM

Scenario Without Salting

- Suppose you have a dataset with partitioning by customer_id.
- If one customer_id has 50% of all rows, then:
  - Spark's partitioner sends all rows with that key to the same partition.
  - One executor must hold huge amounts of data in memory while others stay idle.
  - This causes:
    - OOM (heap space errors)
    - GC overhead limit exceeded
    - Very slow job execution.

Example:

sql

SELECT customer_id, SUM(amount) FROM transactions GROUP BY customer_id;

If one customer_id = 12345 has millions of rows, all go to one task → OOM risk.

3️⃣ How Salting Fixes It

- Instead of using just customer_id as the key, we add a salt value (small random number or sequence) to spread the skewed key's rows into multiple partitions.
- This breaks up the heavy key's data into smaller chunks that can be processed in parallel.

4️⃣ Salting Steps

Step 1 — Detect skew

- Find keys with disproportionately high record counts.

sql

SELECT customer_id, COUNT(*) as cnt FROM transactions GROUP BY customer_id ORDER BY cnt DESC;

Step 2 — Add a salt column

- Append a small random number (salt) only for skewed keys.
- Example: If skewed key = 12345, break it into 10 parts by adding salt values 0–9.

5️⃣ Example — Before & After Salting

Without Salting (Skew)

```
customer_id = 12345 → 100M rows
customer_id = 99999 → 10 rows
...
```

Partitioning causes:

```sql
Partition 1: 100M rows (OOM risk)
Partition 2: 10 rows
Partition 3: 20 rows
...
```

With Salting

We create a new join/groupBy key:

```
new_key = concat(customer_id, "_", salt)
salt = random_int(0, 9)
```

Now rows for 12345 are split:

```sql
12345_0 → ~10M rows
12345_1 → ~10M rows
...
12345_9 → ~10M rows
```

Result:

- Data evenly spread across partitions.
- No single task gets overloaded → OOM avoided.

6️⃣ Spark Example — Join with Salting

Problem Join (Skewed Key)

```
# Two large DataFrames with skew on 'customer_id'
df1.join(df2, "customer_id")
```

- Causes skew & possible executor OOM.

Solution — Salting

Python example :

```
from pyspark.sql.functions import rand, concat_ws, lit, floor
# Add salt to the smaller table (df2) for each customer_id

salt_size = 10  # number of splits
df2_salted = df2.withColumn("salt", floor(rand() * salt_size))
df2_salted = df2_salted.withColumn("join_key", concat_ws("_", df2_salted.customer_id, df2_salted.salt))
```

```
# Duplicate rows in larger table with all possible salts
from pyspark.sql.functions import explode, array
df1_salted = df1.withColumn("salt", explode(array([lit(i) for i in range(salt_size)])))
df1_salted = df1_salted.withColumn("join_key", concat_ws("_", df1_salted.customer_id, df1_salted.salt))

# Join on salted key
result = df1_salted.join(df2_salted, "join_key")
```

7️⃣ Benefits of Salting for OOM

- Memory load spread: Skewed keys no longer overload a single executor.
- Parallel processing: Multiple tasks process parts of the skewed key's data.
- Reduced shuffle size per task → less chance of shuffle OOM.
- Better CPU utilization: All executors do useful work.

8️⃣ Trade-offs

- Increased data size: Salting duplicates rows for skewed keys.
- Extra processing: Need a post-processing step to recombine results for the original key.
- Not useful if data is already evenly distributed.

9️⃣ Diagram — How Salting Helps

Before Salting (Skew)

```
customer_id=12345  → Partition 1: 100M rows (OOM ❌)
customer_id=99999  → Partition 2: 10 rows
```

After Salting (Balanced)

sql

```
customer_id=12345_salt0 → Partition 1: 10M rows
customer_id=12345_salt1 → Partition 2: 10M rows
...
customer_id=12345_salt9 → Partition 10: 10M rows
```

✅ Each partition fits in memory.

🔟 Key Takeaways

- OOM in distributed joins/groupBy often happens due to data skew.
- Salting splits skewed keys into smaller keys → spreads load across executors.
- Best used when a few keys dominate the dataset.
- Requires merge step after aggregation to restore original key.