**🔹 Problem 2 — Join Skew / Shuffle Optimization**

**Problem Statement  
You must join a large events table (1B rows) with a smaller users dimension (1M rows) on user\_id. However, one user (user\_id = 0) accounts for 200M rows in events. How do you efficiently join these two tables in PySpark/Databricks without causing shuffle explosion or OOM errors?**

**✅ Expected Core Answer**

**Step 1: Detect skew**

**events.groupBy("user\_id").count().orderBy(col("count").desc()).show(5)**

**Identify that user\_id=0 is extremely skewed (~20% of rows).**

**Step 2: Broadcast small dimension**

**joined = events.join(broadcast(users), "user\_id", "inner")**

* **Works since users is small.**
* **Still leaves one huge partition for user\_id=0.**

**Step 3: Salting the skewed key (recommended general solution)**

**from pyspark.sql.functions import rand, floor, col, lit**

**S = 40 # salts**

**# Add salt for skewed keys in events**

**events\_salted = events.withColumn(**

**"salt",**

**expr(f"case when user\_id = 0 then cast(floor(rand()\*{S}) as int) else 0 end")**

**)**

**# Replicate user row for skewed key**

**salt\_df = spark.range(S).withColumnRenamed("id", "salt")**

**users\_skewed = users.filter(col("user\_id") == 0).crossJoin(salt\_df)**

**users\_normal = users.filter(col("user\_id") != 0).withColumn("salt", lit(0))**

**users\_salted = users\_normal.unionByName(users\_skewed)**

**# Join on both user\_id and salt**

**joined = events\_salted.join(users\_salted, ["user\_id", "salt"], "inner")**

**Now the 200M skewed rows are distributed across ~40 partitions (5M each).**

**Step 4: Alternative: Separate heavy keys**

**heavy\_events = events.filter(col("user\_id") == 0)**

**light\_events = events.filter(col("user\_id") != 0)**

**joined\_light = light\_events.join(broadcast(users), "user\_id", "inner")**

**user0 = users.filter(col("user\_id") == 0).collect()[0]**

**heavy\_enriched = heavy\_events.withColumn("name", lit(user0["name"])) \**

**.withColumn("email", lit(user0["email"]))**

**Process heavy users separately, rest via normal join.**

**🔍 Follow-Up Q&A Pack (Interview Depth)**

**Q1. How do you pick the number of salts (S)?**

* **Compute: S = ceil(heavy\_key\_rows / target\_rows\_per\_task).**
* **Example: 200M rows for user\_id=0, target ~5M rows per task → S ≈ 40.**
* **Also match cluster parallelism (cores) so work is evenly distributed.**

**Q2. What metrics would you monitor to tune salts?**

* **Shuffle partition sizes (Spark UI: Shuffle Read/Write).**
* **Task duration (stragglers show imbalance).**
* **Memory usage (spill, GC, OOM).**
* **Rows per partition (avoid >500MB / partition).**

**Q3. How does Adaptive Query Execution (AQE) help?**

* **AQE in Spark 3+ automatically detects skew and splits large shuffle partitions.**
* **Often removes need for manual salting.**
* **But manual salting gives deterministic control.**
* **Recommendation: keep AQE enabled; combine with salting if AQE alone doesn’t solve hotspot.**

**Q4. What if multiple user\_ids are skewed, not just one?**

* **Extend salting for each skewed key.**
* **Or apply percentile-based threshold: all keys above X% of total rows get salted.**
* **Maintain a mapping table of (user\_id → S).**

**Q5. What are the tradeoffs between broadcast vs salting vs separating heavy keys?**

* **Broadcast: simplest; avoids shuffle of small side; still suffers from one skewed partition.**
* **Salting: general-purpose; balances work; extra replication overhead.**
* **Separate heavy keys: very efficient if a few heavy keys dominate; more ETL complexity.**

**Q6. How would you implement this in SQL on Databricks?**

**WITH salted\_events AS (**

**SELECT e.\*, CASE WHEN user\_id = 0 THEN CAST(FLOOR(rand()\*40) AS INT) ELSE 0 END AS salt**

**FROM events e**

**),**

**users\_expanded AS (**

**SELECT u.\*, s.salt**

**FROM users u CROSS JOIN (SELECT explode(sequence(0,39)) AS salt)**

**WHERE u.user\_id = 0**

**UNION ALL**

**SELECT u.\*, 0 as salt FROM users u WHERE u.user\_id != 0**

**)**

**SELECT \* FROM salted\_events se JOIN users\_expanded ue**

**ON se.user\_id = ue.user\_id AND se.salt = ue.salt;**

**Q7. How do you ensure the job is idempotent?**

* **Salting is deterministic (rand() can be replaced by hash of event\_id % S for reproducibility).**
* **For separate heavy keys: handle as isolated ETL step → safe to rerun.**

**Q8. What if downstream aggregation is needed (e.g., count events per user)?**

* **After salted join, drop salt and re-aggregate on user\_id.**
* **Example:**

**agg = joined.groupBy("user\_id").agg(count("\*").alias("event\_count"))**

**✅ This covers all basics + advanced points: detection, broadcast, salting, separating heavy keys, AQE, picking salts, tradeoffs, SQL variant, idempotency, and downstream impact.**

**📄 Problem 2 — Join Skew / Shuffle Optimization (Answer Sheet)**

**Problem**  
Joining events (1B) with users (1M) on user\_id.  
Skew: user\_id=0 → 200M rows.

**🔹 Core Answer**

1. **Detect skew**:
2. SELECT user\_id, COUNT(\*) FROM events GROUP BY user\_id ORDER BY COUNT(\*) DESC

→ Find heavy key(s).

1. **Option 1: Broadcast join (if users small)**
2. events.join(broadcast(users), "user\_id")

✅ avoids shuffle of users, ❌ still skew for hot key.

1. **Option 2: Salting skewed key (preferred)**
   * Add random/hash salt to events for hot user(s).
   * Replicate corresponding users rows with same salts.
   * Join on (user\_id, salt).  
     ✅ spreads 200M rows across partitions.
2. **Option 3: Separate heavy key(s)**
   * Split events into heavy (user\_id=0) and light.
   * Join light normally, enrich heavy separately.  
     ✅ simple & efficient if few heavy keys.

**🔹 Follow-Up Ready Replies**

**Q: How to pick number of salts?**

* Formula: S ≈ ceil(heavy\_key\_rows / target\_rows\_per\_task).
* Example: 200M rows, target 5M per task → ~40 salts.
* Align with cluster cores.

**Q: What metrics to monitor?**

* Partition sizes in Spark UI.
* Task duration (stragglers).
* Shuffle read/write sizes.
* Spill / OOM indicators.

**Q: How does AQE help?**

* AQE (Spark 3+) splits skewed shuffle partitions dynamically.
* Often enough alone; salting = deterministic control.
* Keep AQE enabled, combine with salting if needed.

**Q: If multiple skewed keys?**

* Apply salting for each heavy key.
* Or threshold approach: salt all keys >X% of total rows.

**Q: Tradeoffs (Broadcast vs Salting vs Separate heavy keys)?**

* **Broadcast**: easy, but skew remains.
* **Salting**: balances load, extra complexity & replication.
* **Separate heavy keys**: fastest if few hot keys, adds ETL branch.

**Q: Downstream aggregation?**

* Drop salt after join.
* Re-aggregate by user\_id.

**Q: Idempotency?**

* Use deterministic salting (hash(event\_id) % S) instead of rand().
* Ensures repeatable splits across runs.

👉 Short 2–3 sentence interview summary:

“I’d first check for skew by grouping events by user\_id. Since users is small, I’d broadcast it, but that alone won’t fix the hot key. For the 200M-row user, I’d either salt the join key (replicate user row across salts) or process that user separately. AQE can help too, but I prefer deterministic salting for predictable performance.”