✅ Q1. What is Incremental Loading in Spark? How Do You Implement It?

1. **Problem Context:**
   Loading all data every time is inefficient and expensive. You want to load only new or changed records.

2. **Concept:**
   Incremental loading = processing **only the delta** (new or updated data).
   Common techniques:

   - Using **timestamp column** (`last_modified`)
   - Using **watermark** with Structured Streaming
   - Using **CDC (Change Data Capture)** with merge/upsert

3. **Design & Trade-offs:**

   - ✅ Faster processing, low cost
   - ❌ Requires reliable `last_updated` logic, idempotency for re-runs
   - Use `merge` or `insert overwrite` logic based on sink
   - Else causes memory pressure

4. **Real Example:**
   "Salesforce CDC → S3 delta files → Spark reads only `where updated_at > checkpoint_time` → 90% data skipped, job time cut from 2h → 12 mins."

5. **Key Takeaway:**
   Always track watermark/cutoff logic. Ensure job can **re-run without duplicates**.

✅ Q2. How Does Spark Structured Streaming Achieve Exactly-Once Guarantees?

1. **Problem Context:**
   Streaming systems risk duplicate processing **if retries happen**. You want **exactly-once** semantics.

2. **Concept:**
   Spark achieves exactly-once via:

- **Idempotent sinks** (like Delta Lake or Kafka with txn IDs i.e Transaction ID)
- **Checkpointing** of offsets + state
- **Write-ahead logs** in sinks like Delta

3. **Trade-offs:**
   - Requires careful sink design
   - Not all sinks are exactly-once **(e.g., file append = at-least-once)**
   - **Delta + Merge is preferred for end-to-end guarantees AND kafka Transaction ID for exactly once semantics.**
4. **Real Example:**
   "Kafka → Spark → Delta → Merge with keys = idempotent write.
   Even on driver restart, same offsets reprocessed but output was deduped."
5. **Key Takeaway:**
   Exactly-once is possible only with **strong source + sink + logic alignment**.

✅ Q3. What is Checkpointing in Spark? Why Is It Critical?

1. **Problem Context:**
   Long-running streaming apps need fault tolerance.
2. **Concept:**
   Checkpointing saves:
   - Offsets from source (e.g., Kafka)
   - Aggregated state (e.g., groupBy count)
   - Watermark and progress info
3. **Trade-offs:**
   - Required for **stateful aggregations**
   - Checkpoints must be **highly durable (e.g., HDFS, S3)**
   - Corrupted checkpoint = loss of state

4. **Real Example:**
   "S3 checkpoint for 3-day stateful window. After crash, restarted from same offset + state.
   When checkpoint path was on local disk, state was lost — caused data reprocessing."

5. **Key Takeaway:**
   Always use reliable checkpoint path. Without checkpoint, state = zero on restart.

✅ Q4. What Are Watermarks in Spark? Why Are They Needed?

1. **Problem Context:**
   Late-arriving events mess up aggregations (e.g., count by 10-minute window).

Concept:
Watermarks allow Spark to track **how late is too late** for incoming data.

python
CopyEdit
```
withWatermark("event_time", "10 minutes")
```

2.
3. **Trade-offs:**
   - ✅ Allows bounded state retention
   - ❌ Dropped records if they arrive after watermark
   - Needs well-synced event time clocks

4. **Real Example:**
   "Ad impressions arrived 15 mins late → Watermark = 10 mins → 5% of records lost.
   Changed watermark to 20 mins → memory usage doubled."

5. **Key Takeaway:**
   Tune watermark based on real-world delay. **Trade between latency vs correctness.**

Watermark tells Spark how late data is allowed to arrive.
It helps Spark decide when to drop old state and trigger window outputs.

- **Small watermark (low delay) = Low latency but might drop valid late data ❌(use this if speed matters most)**

- Large watermark (high delay) = Handles late data correctly ✅, but increases latency and memory usage ❌ (use this if accuracy matters)

1️⃣ Watermark = 5 minutes

- 🔄 Output is generated quickly → good for low-latency dashboards
- ❌ Late events beyond 5 min will be discarded
- ❌ Aggregations will miss some data

2️⃣ Watermark = 15 minutes

- ✅ Captures more late-arriving events
- ✅ Accurate counts
- ❌ Output is delayed by 15 minutes
- ❌ Higher memory usage (state is retained longer)

✅ Q5. What Is the Common Drift Challenge? How Do You Detect and Handle It?

1. **Problem Context:**
   Source schema silently changes (e.g., new column added) → downstream Spark job fails or processes incorrectly.
2. **Concept:**
   Common Drift Challenge = mismatch between current schema vs expected schema
3. **Detection Strategies:**
   - Use schema evolution flags (`mergeSchema`, `enforceSchema`)
   - Store baseline schema version, compare before processing
   - Alert on schema mismatch
4. **Mitigation:**

- Auto-evolve schema in sink (e.g., Delta `mergeSchema=true`)
- Use schema registry if upstream emits Avro/JSON with schemas
- Create fallback logic to drop unexpected columns or default values

5. **Real Example:**
   "Kafka stream added `promo_code` column → downstream join failed silently. Added schema check before transformation → logged mismatch + alert sent."

6. **Key Takeaway:**
   Schema drift = silent killer. Validate schema and evolve proactively.

✅ Q6. How Do You Handle Stateful Aggregations in Spark Structured Streaming?

1. **Problem Context:**
   You want to aggregate over windows (e.g., 1-hour rolling count), not just microbatches.

2. **Concept:**
   Use `groupByKey().mapGroupsWithState()` or `flatMapGroupsWithState()`
   Maintains memory state per key → updated on new data

3. **Trade-offs:**
   - ✅ Enables advanced use cases like running totals, sessionization
   - ❌ Risk of memory blow-up if too many keys
   - Requires checkpointing

4. **Real Example:**
   "Used `mapGroupsWithState` to track live sessions in user behavior stream. After memory spike, applied TTL on keys to avoid buildup."

5. **Key Takeaway:**
   Stateful logic = powerful but dangerous. Always use timeout and checkpointing.

✅ Q7. What Causes Spark Streaming Job Failures and How Do You Recover?

1. **Common Causes:**
   - **Checkpoint corruption**
   - **Driver OOM / shuffle spill**
   - **Unhandled exceptions in user logic**
   - **Offset commit failure**
2. **Recovery Strategy:**
   - **Durable checkpoint (S3/HDFS)**
   - **Auto-restart logic (e.g., Airflow/Supervisor)**
   - **Alerting on last committed offset**
   - **Add retry + circuit breakers in external API calls**
3. **Real Example:**
   "Streaming job failed due to S3 checkpoint being accidentally deleted. Restored last known offset from Kafka + checkpoint backup → resumed with partial reprocessing."
4. **Takeaway:**
   Streaming needs strong observability, retry logic, and data integrity guards.

✅ Q8. How Do You Reconcile Batch and Streaming in One Pipeline?

🔍 Problem Context:

You want to build a pipeline where:

- ✅ A **streaming source** like clickstream logs or Kafka is continuously ingested
- ✅ A **batch table** like `user_profiles` or `product_catalog` is read periodically

- 🔄 You want to **join** or enrich the streaming data with the batch dataset

---

💡 Concept & Common Approaches

1️⃣Join with Static DataFrame in Streaming

- Read batch table once and broadcast it
- Join with incoming stream

python

CopyEdit

```
product_df = spark.read.parquet("s3://product_catalog/")  # batch source

product_broadcast = broadcast(product_df)  # small dimension table


stream = spark.readStream.format("kafka").load()

parsed = stream.selectExpr("CAST(value AS STRING)")  # parse Kafka stream


joined_df = parsed.join(product_broadcast, on="product_id", how="left")
```

When to use:

- Product/User table is **small**
- Doesn't change often
- Can fit in memory (broadcast)

Trade-offs:

- Not real-time updates of batch table
- You need to **relaunch job** to refresh static broadcast

2️⃣ **Delta Lake as Unified Storage Layer**

- Batch and Streaming **read/write to Delta Lake**
- You get ACID, schema evolution, and MERGE support

Architecture:

css

CopyEdit

```
Kafka (Clickstream) ──┬──▶ Spark Structured Streaming ──▶ Delta Table
                      │                                   (Fact table)
                      └──▶ Spark Batch (Product Catalog) ──▶ Delta Table (Dim table)
```

- Use **MERGE INTO** or **UPSERT** in batch
- Stream can **read and join** with fresh data

Trade-offs:

- Slightly more setup (need Delta Lake)
- Watch for **write conflicts** if batch & stream write same table

✅ Key Takeaway:

Batch and Streaming can co-exist, but:

- Design your data layout carefully (e.g., Delta tables or broadcast)
- Sync strategy must be **explicit and robust** (avoid race conditions or stale joins)