✅ Q1. Unified Batch + Streaming Pipeline in PySpark

Problem:
You want both batch and streaming consumers to read/write to the same storage (e.g., Delta Lake) to ensure a unified data view.

Design Approach:

- Use **Delta Lake** as the sink and source
- Stream and batch can both write to and read from the same Delta table
- Use `merge` or `update` logic in batch, and append/upsert in streaming
- **Ensure isolation: `mergeSchema` + `optimize` + version control**

Example:

- Batch load daily product catalog
- Stream real-time clickstream events
- Join the two downstream for personalization

Takeaway:
Delta Lake = bridge between batch and streaming

Enables **idempotent writes, schema enforcement, ACID compliance**

Use `MERGE INTO` or `UPDATE` in batch to avoid overwrite errors

Always **checkpoint** streaming jobs + use **OPTIMIZ**E to compact files

✅ Stage 1: **Processing in PySpark**

Yes — all types of joins or processing:

- 🔷 Batch + Batch
- 🔷 Batch + Streaming (via broadcast or temp view)
- 🔷 Streaming + Streaming (via watermark joins)

➡️ All are **implemented and executed in PySpark code, before writing to the Delta Lake.**

So the actual business logic — **joins, filters, transformations** — happens inside PySpark.

✅ Q2. Handling Out-of-Order Data in Streaming

Problem:
Events arrive late due to network lag. If not handled, they get dropped or placed in wrong window.

Solution:

- Use `withWatermark()` on event time
- Allow a tolerance window (e.g., 15 minutes)
- Set event-time windowing correctly

python

CopyEdit

```
df.withWatermark("event_time", "15 minutes") \
  .groupBy(window("event_time", "10 minutes")) \
  .agg(...)
```

Trade-off:
Larger watermark → more memory usage → higher latency
Smaller watermark → risk of data loss

✅ Q3. Reprocessing Historical Kafka Data

Scenario:
Bug in transformation logic. Need to reprocess last 7 days of events.

Steps:

1. Use **Kafka offset reset** (e.g., earliest or manual seek)
2. Re-run the Spark job with **new checkpoint directory**
3. Write to separate **staging Delta** table → validate → upsert to prod
4. Enable **mergeSchema = true** if schema evolved

Config:

python

CopyEdit

```
.option("startingOffsets", "earliest")
.option("endingOffsets", "latest")
```

Takeaway:
Never reuse the same checkpoint when reprocessing. Ensure idempotent writes (e.g., `merge`).

✅ In Structured Streaming, checkpointing does save Kafka offsets — along with Spark state.

It stores:

1. ✅ **Offsets** read from Kafka (source progress)
2. ✅ **Watermark info**
3. ✅ **Aggregated state** (e.g., for `groupBy`, `window`, etc.)
4. ✅ **Sink progress** (whether data was written successfully)

✅ Q4. Streaming Compaction in Delta Lake

🧨 Problem:

When Spark **Structured Streaming** writes data to **Delta Lake**, it **writes a new file per micro-batch**.

If you're writing every 10 seconds:

- That's **8640 files/day** (10 sec × 24 hrs × 6).
- Over a week → **60,000+ small files**.

📉 This leads to:

- **Read slowness** (too many files to scan)
- **Increased metadata overhead**
- **Higher cloud storage costs** (S3, GCS)

🛠️ Solution: Streaming Compaction

You run a **separate batch job** periodically to **compact those small files**.

Two ways to compact:

✅ **Delta Lake `OPTIMIZE` command**
(Built-in compaction for Databricks or Delta-compatible engines)

sql
CopyEdit
```
OPTIMIZE delta.`/path/to/table`

ZORDER BY (user_id)
```

1.

✅ **Manual coalesce + overwrite (if OPTIMIZE isn't available)**

python
CopyEdit
```
df = spark.read.format("delta").load("/path/to/table")
```

```
df.coalesce(1).write.mode("overwrite").format("delta").save("/tmp/compacted")
```

2.

🔄 How You Schedule It:

- Run **daily** or **hourly** via **Airflow**, **Apache Oozie**, or a simple cron job.

📦 Where is the final data stored?

Still in **the same Delta table directory**, like `/mnt/delta/transactions`.

But only the **newly compacted files** are marked active in the `_delta_log` version file.

Old small files:

- Are retained temporarily (until vacuum)
- But **not used** for query results anymore

🔐 Column Consistency Check — your concern:

> "Won't merging files with different column names/types break things?"

✅ **Answer: It won't break — if the schema is evolved properly.**

Delta enforces schema consistency during writes. If your data has schema drift:

Use:

python
CopyEdit
```
spark.conf.set("spark.databricks.delta.schema.autoMerge.enabled", "true")
```

-
- Or explicitly cast/align schema before writes.

During compaction, Delta already knows the **active schema** from metadata (`_delta_log`). Only compatible files are read + merged.

If a file has incompatible schema → **compaction will fail** unless schema evolution is handled correctly.

Metric	Before	After
# of Files	10,200	53
Read Latency	14s	2.1s
Metadata Scan Time	3.5s	0.5s

Problem	Delta + Merge Fixes	Compaction Fixes
Exactly-once (deduplication)	✅ Yes	❌ No
Handling CDC events	✅ Yes	❌ No
Too many small files	❌ No	✅ Yes
Slow downstream read performance	❌ No	✅ Yes
Costly metadata operations	❌ No	✅ Yes

✅ Q5. Joining Streaming Facts with Batch Dimensions

Problem:
Need to enrich streaming data (e.g., user events) with **slower-changing batch data** (e.g., user profiles)

Fix:

- Use **broadcast join** with periodic reload of batch dimension
- Use temp view or DF reload every few mins i.e Periodic Reload

🕒 Bonus: Periodic Reload

The batch dimension might change (e.g., user tier updated).

So you should:

- Reload it every **5–10 mins** inside a scheduled job or trigger
- Broadcast the **refreshed DF**

✅ Q6. **Using AQE in Streaming Jobs**

1. Skew Join Handling

- If streaming data is skewed (e.g., some `user_id` has too many events), Spark splits the skewed partition and avoids long straggler tasks.
- ✅ *Helps when one key receives too much data in streaming joins.*

2. Dynamic Join Strategy Switching

- Suppose Spark initially planned a Sort-Merge Join.
- But during actual stream processing, the **batch dimension table is small enough for a Broadcast Join.**
- 🔁 Spark **automatically switches** to a broadcast join to optimize.

3. Partition Coalescing

- Streaming often creates **small files or many partitions.**
- AQE dynamically **coalesces partitions** to reduce shuffle and I/O costs.

✅ Q7. Common Drift Challenge in Streaming Pipelines

Scenario:
Kafka source starts adding new fields → jobs fail

Fixes:

- Use **Delta Lake schema evolution** (`mergeSchema`)
- Use external **schema registry** (e.g., Confluent + Avro)
- Add **alerting** on schema mismatch failures

- Use **try/catch** logic in parsing logic

Proactive:

- Add schema version column
- Auto-promote minor field changes

✅ **Q8. Time Travel-Based Fixes in Delta Lake – Explained**

📌 Scenario:

You pushed **bad data** (e.g., due to a bug in streaming logic or a batch job) into a **Delta table** in the past 2 hours. Now, you want to **undo** it.

🚀 Delta Time Travel – Core Idea:

Delta Lake **keeps older versions** of your table (as snapshots). You can **"travel back"** in time to a specific version or timestamp.

🔧 Recovery Options:

🔁 Option 1: Overwrite Current Table

- Load an old version (e.g., version 387)
- Validate it
- Overwrite the current bad data

🧪 Option 2: Stage → Validate → Promote

- Load old version into a **staging Delta table**
- Run validations or unit tests
- Then **overwrite the prod table** or use it downstream safely

✅ Use Cases of Time Travel:

Use Case	Benefit
Revert corrupted writes	Quick rollback
Re-run jobs on past state	Supports reproducibility, debugging
Auditing data changes	Track who changed what and when
Compare versions	For schema drift or business validation

🧠 Key Takeaway:

- Delta Lake **versioning** gives **streaming + batch / Streaming + Streaming / Batch + Batch rollback capability**.
- It's critical for **bug recovery**, **auditability**, and **safe reprocessing**.