Advanced PySpark — Streaming

Included questions:

5. Stateful Structured Streaming: Advanced Task on `metrics`
6. Watermarking & Late Data: Advanced Task on `orders`
7. Checkpointing & Exactly-once Semantics: Advanced Task on `impressions`
29. Caching vs Checkpointing vs Persist: Advanced Task on `orders`
38. Streaming Joins & State Timeout: Advanced Task on `clicks`
39. Idempotent Sinks Design: Advanced Task on `sessions`
40. Out-of-order Event Handling: Advanced Task on `logs`
41. Checkpoint Recovery Simulation: Advanced Task on `logs`
55. Stateful Structured Streaming: Advanced Task on `impressions`
56. Watermarking & Late Data: Advanced Task on `impressions`
57. Checkpointing & Exactly-once Semantics: Advanced Task on `orders`
79. Caching vs Checkpointing vs Persist: Advanced Task on `metrics`
88. Streaming Joins & State Timeout: Advanced Task on `transactions`
89. Idempotent Sinks Design: Advanced Task on `transactions`
90. Out-of-order Event Handling: Advanced Task on `sessions`
91. Checkpoint Recovery Simulation: Advanced Task on `logs`
105. Stateful Structured Streaming: Advanced Task on `clicks`
106. Watermarking & Late Data: Advanced Task on `events`
107. Checkpointing & Exactly-once Semantics: Advanced Task on `impressions`
129. Caching vs Checkpointing vs Persist: Advanced Task on `orders`
138. Streaming Joins & State Timeout: Advanced Task on `impressions`
139. Idempotent Sinks Design: Advanced Task on `metrics`
140. Out-of-order Event Handling: Advanced Task on `impressions`
141. Checkpoint Recovery Simulation: Advanced Task on `sessions`
155. Stateful Structured Streaming: Advanced Task on `payments`
156. Watermarking & Late Data: Advanced Task on `clicks`
157. Checkpointing & Exactly-once Semantics: Advanced Task on `events`
179. Caching vs Checkpointing vs Persist: Advanced Task on `metrics`
188. Streaming Joins & State Timeout: Advanced Task on `impressions`
189. Idempotent Sinks Design: Advanced Task on `events`
190. Out-of-order Event Handling: Advanced Task on `payments`

191. Checkpoint Recovery Simulation: Advanced Task on `logs`
205. Stateful Structured Streaming: Advanced Task on `events`
206. Watermarking & Late Data: Advanced Task on `transactions`
207. Checkpointing & Exactly-once Semantics: Advanced Task on `events`
229. Caching vs Checkpointing vs Persist: Advanced Task on `impressions`
238. Streaming Joins & State Timeout: Advanced Task on `events`
239. Idempotent Sinks Design: Advanced Task on `impressions`
240. Out-of-order Event Handling: Advanced Task on `orders`
241. Checkpoint Recovery Simulation: Advanced Task on `impressions`

5. Stateful Structured Streaming: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `session_id`, `event_time`, and `latency_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a stateful structured streaming problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Stateful Structured Streaming (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Stateful streaming stores per-key state for aggregations. We define a watermark on `event_time`, use `groupByKey` with `mapGroupsWithState` (or `flatMapGroupsWithState`) to maintain counters and emit derived metrics while bounding state with timeouts.

```
from pyspark.sql import functions as F, types as T
from pyspark.sql.streaming import GroupState, GroupStateTimeout

schema = " session_id string, event_time timestamp, latency_ms double "

stream = (spark.readStream.format("json")
          .schema(schema)
          .option("maxFilesPerTrigger", 1)
          .load("/data/metrics"))

def update_state(key_value, rows_iter, state: GroupState):
    total = state.get("total") if state.exists else 0.0
    for r in rows_iter:
        total += r["latency_ms"] or 0.0
    state.update({"total": total})
    state.setTimeoutDuration("1 hour")
    return [(key_value, total)]

agg = (stream
       .withWatermark("event_time", "30 minutes")
       .groupByKey(lambda r: r["session_id"])
       .flatMapGroupsWithState(
            outputMode="update",
            stateTimeout=GroupStateTimeout.ProcessingTimeTimeout(),
            func=update_state
       ))

q = (agg.toDF("session_id", "running_total")
     .writeStream
     .format("delta")
     .outputMode("update")
     .option("checkpointLocation", "/chk/metrics")
     .start("/out/metrics"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

6. Watermarking & Late Data: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `order_id`, `updated_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a watermarking & late data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Watermarking & Late Data (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Watermarks bound late data and enable state eviction.

```
from pyspark.sql import functions as F

agg = (streaming_df
       .withWatermark("updated_at", "20 minutes")
       .groupBy(F.window(F.col("updated_at"), "10 minutes"), F.col("order_id"))
       .agg(F.sum("value").alias("sum_value")))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

7. Checkpointing & Exactly-once Semantics: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `session_id`, `created_at`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpointing & exactly-once semantics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpointing & Exactly-once Semantics (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/impressions")
     .start("/out/impressions"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

29. Caching vs Checkpointing vs Persist: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `customer_id`, `updated_at`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a caching vs checkpointing vs persist problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Caching vs Checkpointing vs Persist (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/orders")
     .start("/out/orders"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

38. Streaming Joins & State Timeout: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `session_id`, `event_time`, and `quantity`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a streaming joins & state timeout problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Streaming Joins & State Timeout (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Streaming-streaming joins need watermarks on both sides and a time bound.

```
a = a.withWatermark("event_time", "10 minutes")
b = b.withWatermark("event_time", "10 minutes")
joined = a.join(b, [a["session_id"]==b["session_id"]], "inner")
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

39. Idempotent Sinks Design: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `order_id`, `ts`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a idempotent sinks design problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Idempotent Sinks Design (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Upsert with foreachBatch; avoid duplicates across retries.

```
def upsert(batch_df, batch_id):
    batch_df.createOrReplaceTempView("batch")
    spark.sql("""
    MERGE INTO tgt t
    USING batch b ON t.order_id=b.order_id
    WHEN MATCHED THEN UPDATE SET *
    WHEN NOT MATCHED THEN INSERT *
    """)

q = (streaming_df.writeStream.foreachBatch(upsert)
     .option("checkpointLocation","/chk/idem").start())
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

40. Out-of-order Event Handling: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `customer_id`, `event_time`, and `latency_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a out-of-order event handling problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Out-of-order Event Handling (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Choose watermark horizon from observed lateness; drop too-late records.

```
# See watermark example above.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

41. Checkpoint Recovery Simulation: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `device_id`, `event_time`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpoint recovery simulation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpoint Recovery Simulation (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Verify restart resumes from checkpoint; ensure deterministic sink behavior.

```
# Operational steps and assertions.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

55. Stateful Structured Streaming: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `customer_id`, `ts`, and `quantity`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a stateful structured streaming problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Stateful Structured Streaming (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Stateful streaming stores per-key state for aggregations. We define a watermark on `ts`, use `groupByKey` with `mapGroupsWithState` (or `flatMapGroupsWithState`) to maintain counters and emit derived metrics while bounding state with timeouts.

```
from pyspark.sql import functions as F, types as T
from pyspark.sql.streaming import GroupState, GroupStateTimeout
```

```
schema = " customer_id string, ts timestamp, quantity double "

stream = (spark.readStream.format("json")
          .schema(schema)
          .option("maxFilesPerTrigger", 1)
          .load("/data/impressions"))

def update_state(key_value, rows_iter, state: GroupState):
    total = state.get("total") if state.exists else 0.0
    for r in rows_iter:
        total += r["quantity"] or 0.0
    state.update({"total": total})
    state.setTimeoutDuration("1 hour")
    return [(key_value, total)]

agg = (stream
       .withWatermark("ts", "30 minutes")
       .groupByKey(lambda r: r["customer_id"])
       .flatMapGroupsWithState(
            outputMode="update",
            stateTimeout=GroupStateTimeout.ProcessingTimeTimeout(),
            func=update_state
       ))

q = (agg.toDF("customer_id", "running_total")
     .writeStream
     .format("delta")
     .outputMode("update")
     .option("checkpointLocation", "/chk/impressions")
     .start("/out/impressions"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

56. Watermarking & Late Data: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `customer_id`, `updated_at`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a watermarking & late data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Watermarking & Late Data (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Watermarks bound late data and enable state eviction.

```
from pyspark.sql import functions as F

agg = (streaming_df
       .withWatermark("updated_at", "20 minutes")
       .groupBy(F.window(F.col("updated_at"), "10 minutes"), F.col("customer_id"))
       .agg(F.sum("duration_ms").alias("sum_duration_ms")))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

57. Checkpointing & Exactly-once Semantics: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_id`, `event_time`, and `quantity`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpointing & exactly-once semantics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpointing & Exactly-once Semantics (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/orders")
     .start("/out/orders"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

79. Caching vs Checkpointing vs Persist: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `account_id`, `created_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a caching vs checkpointing vs persist problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Caching vs Checkpointing vs Persist (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/metrics")
     .start("/out/metrics"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

88. Streaming Joins & State Timeout: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `customer_id`, `created_at`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a streaming joins & state timeout problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Streaming Joins & State Timeout (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Streaming-streaming joins need watermarks on both sides and a time bound.

```
a = a.withWatermark("created_at", "10 minutes")
b = b.withWatermark("created_at", "10 minutes")
joined = a.join(b, [a["customer_id"]==b["customer_id"]], "inner")
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

89. Idempotent Sinks Design: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `order_id`, `ts`, and `score`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a idempotent sinks design problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Idempotent Sinks Design (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Upsert with foreachBatch; avoid duplicates across retries.

```
def upsert(batch_df, batch_id):
    batch_df.createOrReplaceTempView("batch")
    spark.sql("""
    MERGE INTO tgt t
    USING batch b ON t.order_id=b.order_id
    WHEN MATCHED THEN UPDATE SET *
    WHEN NOT MATCHED THEN INSERT *
    """)

q = (streaming_df.writeStream.foreachBatch(upsert)
     .option("checkpointLocation","/chk/idem").start())
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

90. Out-of-order Event Handling: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `session_id`, `ts`, and `quantity`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a out-of-order event handling problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Out-of-order Event Handling (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Choose watermark horizon from observed lateness; drop too-late records.

```
# See watermark example above.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

91. Checkpoint Recovery Simulation: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `customer_id`, `created_at`, and `latency_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpoint recovery simulation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpoint Recovery Simulation (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Verify restart resumes from checkpoint; ensure deterministic sink behavior.

```
# Operational steps and assertions.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

105. Stateful Structured Streaming: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `order_id`, `event_time`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a stateful structured streaming problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Stateful Structured Streaming (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Stateful streaming stores per-key state for aggregations. We define a watermark on `event_time`, use `groupByKey` with `mapGroupsWithState` (or `flatMapGroupsWithState`) to maintain counters and emit derived metrics while bounding state with timeouts.

```
from pyspark.sql import functions as F, types as T
from pyspark.sql.streaming import GroupState, GroupStateTimeout
```

```
schema = " order_id string, event_time timestamp, duration_ms double "

stream = (spark.readStream.format("json")
          .schema(schema)
          .option("maxFilesPerTrigger", 1)
          .load("/data/clicks"))

def update_state(key_value, rows_iter, state: GroupState):
    total = state.get("total") if state.exists else 0.0
    for r in rows_iter:
        total += r["duration_ms"] or 0.0
    state.update({"total": total})
    state.setTimeoutDuration("1 hour")
    return [(key_value, total)]

agg = (stream
       .withWatermark("event_time", "30 minutes")
       .groupByKey(lambda r: r["order_id"])
       .flatMapGroupsWithState(
            outputMode="update",
            stateTimeout=GroupStateTimeout.ProcessingTimeTimeout(),
            func=update_state
       ))

q = (agg.toDF("order_id", "running_total")
     .writeStream
     .format("delta")
     .outputMode("update")
     .option("checkpointLocation", "/chk/clicks")
     .start("/out/clicks"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

106. Watermarking & Late Data: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `device_id`, `ts`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a watermarking & late data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Watermarking & Late Data (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Watermarks bound late data and enable state eviction.

```
from pyspark.sql import functions as F

agg = (streaming_df
       .withWatermark("ts", "20 minutes")
       .groupBy(F.window(F.col("ts"), "10 minutes"), F.col("device_id"))
       .agg(F.sum("value").alias("sum_value")))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

107. Checkpointing & Exactly-once Semantics: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `account_id`, `created_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpointing & exactly-once semantics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpointing & Exactly-once Semantics (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/impressions")
     .start("/out/impressions"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

129. Caching vs Checkpointing vs Persist: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `order_id`, `created_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a caching vs checkpointing vs persist problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Caching vs Checkpointing vs Persist (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/orders")
     .start("/out/orders"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

138. Streaming Joins & State Timeout: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `order_id`, `created_at`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a streaming joins & state timeout problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Streaming Joins & State Timeout (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Streaming-streaming joins need watermarks on both sides and a time bound.

```
a = a.withWatermark("created_at", "10 minutes")
b = b.withWatermark("created_at", "10 minutes")
joined = a.join(b, [a["order_id"]==b["order_id"]], "inner")
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

139. Idempotent Sinks Design: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `device_id`, `updated_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a idempotent sinks design problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Idempotent Sinks Design (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Upsert with foreachBatch; avoid duplicates across retries.

```
def upsert(batch_df, batch_id):
    batch_df.createOrReplaceTempView("batch")
    spark.sql("""
    MERGE INTO tgt t
    USING batch b ON t.device_id=b.device_id
    WHEN MATCHED THEN UPDATE SET *
    WHEN NOT MATCHED THEN INSERT *
    """)

q = (streaming_df.writeStream.foreachBatch(upsert)
     .option("checkpointLocation","/chk/idem").start())
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

140. Out-of-order Event Handling: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `device_id`, `created_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a out-of-order event handling problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Out-of-order Event Handling (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Choose watermark horizon from observed lateness; drop too-late records.

```
# See watermark example above.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

141. Checkpoint Recovery Simulation: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `session_id`, `updated_at`, and `quantity`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpoint recovery simulation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpoint Recovery Simulation (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Verify restart resumes from checkpoint; ensure deterministic sink behavior.

```
# Operational steps and assertions.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

155. Stateful Structured Streaming: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `order_id`, `ts`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a stateful structured streaming problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Stateful Structured Streaming (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Stateful streaming stores per-key state for aggregations. We define a watermark on `ts`, use `groupByKey` with `mapGroupsWithState` (or `flatMapGroupsWithState`) to maintain counters and emit derived metrics while bounding state with timeouts.

```
from pyspark.sql import functions as F, types as T
from pyspark.sql.streaming import GroupState, GroupStateTimeout
```

```
schema = " order_id string, ts timestamp, duration_ms double "

stream = (spark.readStream.format("json")
          .schema(schema)
          .option("maxFilesPerTrigger", 1)
          .load("/data/payments"))

def update_state(key_value, rows_iter, state: GroupState):
    total = state.get("total") if state.exists else 0.0
    for r in rows_iter:
        total += r["duration_ms"] or 0.0
    state.update({"total": total})
    state.setTimeoutDuration("1 hour")
    return [(key_value, total)]

agg = (stream
       .withWatermark("ts", "30 minutes")
       .groupByKey(lambda r: r["order_id"])
       .flatMapGroupsWithState(
            outputMode="update",
            stateTimeout=GroupStateTimeout.ProcessingTimeTimeout(),
            func=update_state
       ))

q = (agg.toDF("order_id", "running_total")
     .writeStream
     .format("delta")
     .outputMode("update")
     .option("checkpointLocation", "/chk/payments")
     .start("/out/payments"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

156. Watermarking & Late Data: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `session_id`, `event_time`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a watermarking & late data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Watermarking & Late Data (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Watermarks bound late data and enable state eviction.

```
from pyspark.sql import functions as F

agg = (streaming_df
       .withWatermark("event_time", "20 minutes")
       .groupBy(F.window(F.col("event_time"), "10 minutes"), F.col("session_id"))
       .agg(F.sum("duration_ms").alias("sum_duration_ms")))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

157. Checkpointing & Exactly-once Semantics: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `user_id`, `updated_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpointing & exactly-once semantics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpointing & Exactly-once Semantics (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/events")
     .start("/out/events"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

179. Caching vs Checkpointing vs Persist: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `device_id`, `updated_at`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a caching vs checkpointing vs persist problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Caching vs Checkpointing vs Persist (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/metrics")
     .start("/out/metrics"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

188. Streaming Joins & State Timeout: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `order_id`, `updated_at`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a streaming joins & state timeout problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Streaming Joins & State Timeout (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Streaming-streaming joins need watermarks on both sides and a time bound.

```
a = a.withWatermark("updated_at", "10 minutes")
b = b.withWatermark("updated_at", "10 minutes")
joined = a.join(b, [a["order_id"]==b["order_id"]], "inner")
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

189. Idempotent Sinks Design: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `user_id`, `created_at`, and `value`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a idempotent sinks design problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Idempotent Sinks Design (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Upsert with foreachBatch; avoid duplicates across retries.

```
def upsert(batch_df, batch_id):
    batch_df.createOrReplaceTempView("batch")
    spark.sql("""
    MERGE INTO tgt t
    USING batch b ON t.user_id=b.user_id
    WHEN MATCHED THEN UPDATE SET *
    WHEN NOT MATCHED THEN INSERT *
    """)

q = (streaming_df.writeStream.foreachBatch(upsert)
     .option("checkpointLocation","/chk/idem").start())
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

190. Out-of-order Event Handling: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `order_id`, `ts`, and `score`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a out-of-order event handling problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Out-of-order Event Handling (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Choose watermark horizon from observed lateness; drop too-late records.

```
# See watermark example above.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

191. Checkpoint Recovery Simulation: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `device_id`, `created_at`, and `duration_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpoint recovery simulation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpoint Recovery Simulation (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Verify restart resumes from checkpoint; ensure deterministic sink behavior.

```
# Operational steps and assertions.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

205. Stateful Structured Streaming: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `account_id`, `updated_at`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a stateful structured streaming problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Stateful Structured Streaming (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Stateful streaming stores per-key state for aggregations. We define a watermark on `updated_at`, use `groupByKey` with `mapGroupsWithState` (or `flatMapGroupsWithState`) to maintain counters and emit derived metrics while bounding state with timeouts.

```
from pyspark.sql import functions as F, types as T
from pyspark.sql.streaming import GroupState, GroupStateTimeout
```

```
schema = " account_id string, updated_at timestamp, amount double "

stream = (spark.readStream.format("json")
          .schema(schema)
          .option("maxFilesPerTrigger", 1)
          .load("/data/events"))

def update_state(key_value, rows_iter, state: GroupState):
    total = state.get("total") if state.exists else 0.0
    for r in rows_iter:
        total += r["amount"] or 0.0
    state.update({"total": total})
    state.setTimeoutDuration("1 hour")
    return [(key_value, total)]

agg = (stream
       .withWatermark("updated_at", "30 minutes")
       .groupByKey(lambda r: r["account_id"])
       .flatMapGroupsWithState(
            outputMode="update",
            stateTimeout=GroupStateTimeout.ProcessingTimeTimeout(),
            func=update_state
       ))

q = (agg.toDF("account_id", "running_total")
     .writeStream
     .format("delta")
     .outputMode("update")
     .option("checkpointLocation", "/chk/events")
     .start("/out/events"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

206. Watermarking & Late Data: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `device_id`, `updated_at`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a watermarking & late data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Watermarking & Late Data (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Watermarks bound late data and enable state eviction.

```
from pyspark.sql import functions as F

agg = (streaming_df
       .withWatermark("updated_at", "20 minutes")
       .groupBy(F.window(F.col("updated_at"), "10 minutes"), F.col("device_id"))
       .agg(F.sum("amount").alias("sum_amount")))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

207. Checkpointing & Exactly-once Semantics: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `customer_id`, `ts`, and `latency_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpointing & exactly-once semantics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpointing & Exactly-once Semantics (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/events")
     .start("/out/events"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

229. Caching vs Checkpointing vs Persist: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `session_id`, `updated_at`, and `score`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a caching vs checkpointing vs persist problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Caching vs Checkpointing vs Persist (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Checkpoint offsets/state to recover after failures; use idempotent sinks.

```
q = (streaming_df
     .writeStream
     .format("parquet")
     .option("checkpointLocation", "/chk/impressions")
     .start("/out/impressions"))
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

238. Streaming Joins & State Timeout: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `customer_id`, `created_at`, and `latency_ms`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a streaming joins & state timeout problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Streaming Joins & State Timeout (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Streaming-streaming joins need watermarks on both sides and a time bound.

```
a = a.withWatermark("created_at", "10 minutes")
b = b.withWatermark("created_at", "10 minutes")
joined = a.join(b, [a["customer_id"]==b["customer_id"]], "inner")
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

239. Idempotent Sinks Design: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `order_id`, `updated_at`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a idempotent sinks design problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Idempotent Sinks Design (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Upsert with foreachBatch; avoid duplicates across retries.

```
def upsert(batch_df, batch_id):
    batch_df.createOrReplaceTempView("batch")
    spark.sql("""
    MERGE INTO tgt t
    USING batch b ON t.order_id=b.order_id
    WHEN MATCHED THEN UPDATE SET *
    WHEN NOT MATCHED THEN INSERT *
    """)

q = (streaming_df.writeStream.foreachBatch(upsert)
     .option("checkpointLocation","/chk/idem").start())
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

240. Out-of-order Event Handling: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `account_id`, `ts`, and `score`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a out-of-order event handling problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Out-of-order Event Handling (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Choose watermark horizon from observed lateness; drop too-late records.

```
# See watermark example above.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.

241. Checkpoint Recovery Simulation: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `order_id`, `updated_at`, and `amount`. The data arrives from multiple sources as Parquet/JSON with evolving schemas.

Task. Using PySpark, implement a robust solution to solve a checkpoint recovery simulation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Checkpoint Recovery Simulation (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

- Large scale, evolving schemas, and skewed keys. - Requires balancing correctness, latency, and cost. - Involves optimizer behavior, partitions, and state (for streaming).

Solution Outline & Explanation

Verify restart resumes from checkpoint; ensure deterministic sink behavior.

```
# Operational steps and assertions.
```

Validation

- Unit tests over representative edge cases (nulls, duplicates, late/out-of-order events). - Profile partitions and task skew in Spark UI. - Compare aggregates vs. source-of-truth; implement data quality gates.