250 Complex PySpark Coding Questions — With Explanations & Code

Table of Contents (partial):

1. Window Functions & Analytics: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` 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 window functions & analytics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Window Functions & Analytics (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

We use window partitions by `user_id` ordered by `created_at` to compute analytics like rolling sums, lag/lead, and first/last. We must guard for null timestamps and ensure a stable ordering. We also consider `rangeBetween` vs `rowsBetween` depending on semantic needs.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("user_id").orderBy(F.col("created_at").cast("timestamp"))

df_clean = (
    df
    .withColumn("created_at", F.to_timestamp("created_at"))
    .withColumn("value", F.col("value").cast("double"))
    .dropna(subset=["user_id", "created_at"])
)

result = (
    df_clean
    .withColumn("prev_value", F.lag("value").over(w))
    .withColumn("rolling_sum_3", F.sum("value").over(w.rowsBetween(-2, 0)))
    .withColumn("rank_desc", F.row_number().over(w.orderBy(F.desc("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.

2. Complex Joins & Skew Handling: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `customer_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 complex joins & skew handling problem: - Ingest data with proper schema handling. - Apply necessary transformations

(null-safety, casting, deduplication). - Implement the core logic related to Complex Joins & Skew 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

Skew joins cause a few keys to dominate shuffles. We first profile key frequency, then salt hot keys and broadcast small dimension tables where possible. Enabling AQE can also coalesce skewed partitions. We demonstrate a salting approach.

```
from pyspark.sql import functions as F
from pyspark.sql import types as T

key_freq = df.groupBy("customer_id").count()
hot = key_freq.filter("count > 1000000").select("customer_id").withColumn("is_hot", F.li
t(1))
df2 = df.join(hot, on="customer_id", how="left").fillna({"is_hot":0})

salt_mod = 16
df_salted = df2.withColumn("salt", F.when(F.col("is_hot")==1,
    F.rand()*salt_mod).otherwise(F.lit(0)).cast("int"))

dim_salted = dim.crossJoin(
    spark.range(salt_mod).withColumnRenamed("id","salt")
)
joined = df_salted.join(dim_salted, on=[ "customer_id", "salt" ], how="left")
```

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.

3. Nested JSON & Semi-structured Data: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `user_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 nested json & semi-structured data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Nested JSON & Semi-structured 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

For semi-structured inputs, prefer `from_json` with an explicit schema, handle `badRecordsPath`, and use `explode` for arrays. We also guard against nullable subfields and schema drift.

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

schema = T.StructType([
    T.StructField("user_id", T.StringType()),
    T.StructField("updated_at", T.TimestampType()),
    T.StructField("payload", T.StructType([
        T.StructField("items", T.ArrayType(T.StructType([
            T.StructField("sku", T.StringType()),
            T.StructField("amount", T.DoubleType())
        ])))
    ]))
])

raw = (spark.read
       .option("multiLine", True)
       .option("badRecordsPath", "/tmp/bad_records")
       .json("/data/metrics/*.json"))

dfj = raw.select(F.from_json(F.col("value").cast("string"), schema).alias("r")).select("
r.*")
items = dfj.select("user_id", "updated_at", F.explode_outer("payload.items").alias("it")
)
result = items.select("user_id", "updated_at", F.col("it.sku").alias("sku"),
    F.col(f"it.amount").alias("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.

4. UDFs vs Pandas UDFs & Vectorization: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `user_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 udfs vs pandas udfs & vectorization problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to UDFs vs Pandas UDFs & Vectorization (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("value", F.col("value").cast("double"))
out = df2.withColumn("z_value", zscore(F.col("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.

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.

8. File-based Incremental Ingestion: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `order_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 file-based incremental ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File-based Incremental Ingestion (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

Track high-watermarks and process only new data; design idempotent upserts.

```python
from pyspark.sql import functions as F

prev_max_ts = "2025-01-01T00:00:00"

new_df = (spark.read.parquet("/data/clicks")
          .filter(F.col("updated_at") > F.to_timestamp(F.lit(prev_max_ts))))
```

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.

9. Delta Lake Optimize/Z-Order (conceptual with PySpark): Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `customer_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 delta lake optimize/z-order (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Delta Lake Optimize/Z-Order (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.customer_id = s.customer_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

10. CDC/Merge into Delta (conceptual with PySpark): Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `user_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 cdc/merge into delta (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to CDC/Merge into Delta (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.user_id = s.user_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

11. Bucketing, Partitioning & Writer Jobs: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `account_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 bucketing, partitioning & writer jobs problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Bucketing, Partitioning & Writer Jobs (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

General advanced PySpark pattern.

```
pass
```

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.

12. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` 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 adaptive query execution (aqe) and shuffle partitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Adaptive Query Execution (AQE) and Shuffle Partitions (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

Enable AQE and tune shuffle partitions for better task balance.

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.shuffle.partitions", "200")

dfj = fact.join(F.broadcast(dim), on="session_id", how="left")
```

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.

13. Broadcast Joins and Hints: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `session_id`, `created_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 broadcast joins and hints problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Broadcast Joins and Hints (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

Broadcast small side tables to avoid shuffles.

```
from pyspark.sql import functions as F
joined = fact.hint("broadcast").join(dim, on="session_id", how="left")
```

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.

14. Skew Join Salting Techniques: Advanced Task on `logs`

Question

Scenario. You have a large `logs` 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 skew join salting techniques problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Skew Join Salting Techniques (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

General advanced PySpark pattern.

```
pass
```

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.

15. Aggregations with Complex Grouping Sets: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `user_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 aggregations with complex grouping sets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Aggregations with Complex Grouping Sets (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

Use cube/rollup for multi-level aggregations.

```
from pyspark.sql import functions as F
cube = (df.cube("user_id", "sku").agg(F.sum("score").alias("sum_score")))
```

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.

16. Explode + Window Hybrids: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `customer_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 explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (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

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("customer_id", "event_time", F.explode("items").alias("it"))
```

```
w = W.partitionBy("customer_id", "it").orderBy("event_time")
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

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.

17. Sessionization (clickstreams): Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` 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 sessionization (clickstreams) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Sessionization (clickstreams) (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

Derive sessions from gaps between events.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("device_id").orderBy("created_at")
df2 = (df
       .withColumn("prev_ts", F.lag("created_at").over(w))
       .withColumn("gap_sec", F.coalesce(F.col("created_at").cast("long") - F.col("prev_
ts").cast("long"),
    F.lit(0)))
       .withColumn("new_sess", (F.col("gap_sec") > 1800).cast("int"))
       .withColumn("session_seq", F.sum("new_sess").over(w.rowsBetween(Window.unboundedP
receding, 0)))
      )
```

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.

18. Time-series Gaps & Islands: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `account_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 time-series gaps & islands problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Time-series Gaps & Islands (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

Identify contiguous ranges (islands) using row-number differences.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("account_id").orderBy("ts")
df2 = df.withColumn("rn", F.row_number().over(w))
df3 = df2.withColumn("grp", F.expr("rn - row_number() over (partition by account_id orde
r by ts)"))
```

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.

19. Surrogate Keys & Deduplication: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `order_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 surrogate keys & deduplication problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Surrogate Keys & Deduplication (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

Deduplicate by stable ordering and build surrogate keys via hashes.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("order_id").orderBy(F.desc("event_time"))
dedup = (df.withColumn("rn", F.row_number().over(w)).filter("rn = 1").drop("rn"))
with_id = dedup.withColumn("surrogate_id", F.sha2(F.concat_ws("||", *dedup.columns), 256
))
```

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.

20. SCD Type 2 with MERGE logic (Delta/Parquet): Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `session_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 scd type 2 with merge logic (delta/parquet) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to SCD Type 2 with MERGE logic (Delta/Parquet) (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

Maintain history via effective_from/to and is_current flags; build updates and closures.

```
# See MERGE example; or implement DataFrame-based SCD2 staging logic.
```

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.

21. Advanced Window: Last non-null forward-fill: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `device_id`, `created_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 advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (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

Forward-fill values using `last(..., ignorenulls=True)`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("device_id").orderBy("created_at").rowsBetween(Window.unboundedPreceding, 0)
ff = df.withColumn("ff_val", F.last("quantity", ignorenulls=True).over(w))
```

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.

22. Top-K per Group at Scale: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` 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 top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (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

Rank items per group and filter to K.

```
from pyspark.sql import functions as F, Window as W
K = 3
w = W.partitionBy("customer_id").orderBy(F.desc("amount"))
topk = df.withColumn("r", F.row_number().over(w)).filter(F.col("r") <= K).drop("r")
```

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.

23. Rolling Distinct Counts (HLL sketch concept): Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_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 rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (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

Approximate distinct counts per rolling window with `approx_count_distinct`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("user_id").orderBy("created_at").rowsBetween(-10, 0)
roll = df.withColumn("approx_dc", F.approx_count_distinct("duration_ms").over(w))
```

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.

24. Cross-file Schema Evolution: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `user_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 cross-file schema evolution problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Cross-file Schema Evolution (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/out/sessions")
```

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.

25. Dynamic File Pruning: Advanced Task on `logs`

Question

Scenario. You have a large `logs` 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 dynamic file pruning problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Dynamic File Pruning (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

Partition by time and filter by partition columns for pruning.

```
pruned = spark.read.parquet("/out/logs").filter(F.col("updated_at") >= "2025-01-01")
```

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.

26. Data Quality Checks & Expectations: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `customer_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 data quality checks & expectations problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Data Quality Checks & Expectations (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

Build rule-based validations; quarantine failures with reasons.

```
from pyspark.sql import functions as F

rules = [
    ("not_null_key", F.col("customer_id").isNotNull()),
```

```
    ("val_non_negative", F.col("value") >= 0)
]

def apply_rules(df):
    for rule_name, cond in rules:
        df = df.withColumn("rule_" + rule_name, cond)
    return df

scored = apply_rules(df)
bad = scored.filter("NOT (rule_not_null_key AND rule_val_non_negative)").withColumn("rea
son",
    F.lit("dq_failed"))
good = scored.filter("rule_not_null_key AND
    rule_val_non_negative").drop("rule_not_null_key","rule_val_non_negative")
```

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.

27. Unit Testing with pytest & chispa: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `order_id`, `created_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 unit testing with pytest & chispa problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Unit Testing with pytest & chispa (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

Use pytest + chispa to assert DataFrame equality; isolate pure transforms.

```
# pip install chispa
from chispa import assert_df_equality

def transform(df):
    return df.filter("amount > 0")

def test_transform(spark):
    input_df = spark.createDataFrame([(1, -1.0), (2, 3.0)], ["id","amount"])
    exp_df = spark.createDataFrame([(2, 3.0)], ["id","amount"])
    assert_df_equality(transform(input_df), exp_df, ignore_column_order=True)
```

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.

28. Performance Debugging with UI & Query Plans: Advanced Task on `payments`

Question

Scenario. You have a large `payments` 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 performance debugging with ui & query plans problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Performance Debugging with UI & Query Plans (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

Inspect query plans and the Spark UI; avoid Python UDFs and skew.

```
df_explain = df.select("session_id", "score").groupBy("session_id").agg(F.sum("score"))
print(df_explain._jdf.queryExecution().toString())
```

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.

30. Reusable Jobs & Parameterized Notebooks: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `user_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 reusable jobs & parameterized notebooks problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reusable Jobs & Parameterized Notebooks (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

General advanced PySpark pattern.

```
    pass
```

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.

31. DataFrame <-> Spark SQL Interop: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `order_id`, `updated_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 dataframe <-> spark sql interop problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to DataFrame <-> Spark SQL Interop (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

Register temp views to use Spark SQL alongside DataFrame API.

```
df.createOrReplaceTempView("v")
sql_df = spark.sql("select order_id, sum(latency_ms) as s from v group by order_id")
```

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.

32. Pivot/Unpivot Large Datasets: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `device_id`, `updated_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 pivot/unpivot large datasets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Pivot/Unpivot Large Datasets (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

Pivot after pre-aggregating to avoid explosion.

```
from pyspark.sql import functions as F
piv = df.groupBy("device_id").pivot("sku").agg(F.sum("latency_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.

33. Joins over Ranges (temporal joins): Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `session_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 joins over ranges (temporal joins) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Joins over Ranges (temporal joins) (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

Join facts to dimensions where timestamp falls within validity range.

```
joined = fact.join(dim, (fact["created_at"].between(dim["start"], dim["end"])) &
    (fact["session_id"]==dim["session_id"]), "left")
```

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.

34. Windowed UDAFs (via Pandas UDFs): Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `order_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 windowed udafs (via pandas udfs) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Windowed UDAFs (via Pandas UDFs) (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```python
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("quantity", F.col("quantity").cast("double"))
out = df2.withColumn("z_quantity", zscore(F.col("quantity")))
```

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.

35. Binary Files & Image Ingestion: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `account_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 binary files & image ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Binary Files & Image Ingestion (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

Read binary files for feature extraction or ML preprocessing.

```
images = spark.read.format("binaryFile").load("/data/images/*")
```

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.

36. Graph-Style Problems without GraphFrames: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `device_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 graph-style problems without graphframes problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Graph-Style Problems without GraphFrames (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

Approximate graph analytics via SQL/DF ops (degree counts, simple traversals).

```
edges = spark.createDataFrame([("a","b"),("b","c")], ["src","dst"])
deg = (edges.select("src").union(edges.select("dst"))
         .groupBy("src").count())
```

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.

37. MLlib Pipelines with Custom Transformers: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `order_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 mllib pipelines with custom transformers problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to MLlib Pipelines with Custom Transformers (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

Build ML pipelines; ensure proper vectorization and column roles.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

va = VectorAssembler(inputCols=["f1","f2","f3"], outputCol="features")
lr = LogisticRegression(featuresCol="features", labelCol="label")
pipe = Pipeline(stages=[va, lr]).fit(train_df)
```

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.

42. File Compaction Job: Advanced Task on `logs`

Question

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

Task. Using PySpark, implement a robust solution to solve a file compaction job problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File Compaction Job (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/logs")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/logs_compacted"))
```

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.

43. Small-file Problem Mitigation: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `session_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 small-file problem mitigation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Small-file Problem Mitigation (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/orders")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/orders_compacted"))
```

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.

44. Reading from Hive Metastore & External Tables: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `user_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 reading from hive metastore & external tables problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading from Hive Metastore & External Tables (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

Integrate with Hive catalog; repair partitions; manage external tables.

```
spark.sql("MSCK REPAIR TABLE db.tbl")
```

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.

45. Security & PII Masking Patterns: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `session_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 security & pii masking patterns problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Security & PII Masking Patterns (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

Mask/obfuscate sensitive columns; restrict access via views/catalog controls.

```
from pyspark.sql import functions as F
masked = df.withColumn("email_masked", F.sha2(F.col("email"), 256))
```

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.

46. Column-level Encryption (conceptual + UDF demo): Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `session_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 column-level encryption (conceptual + udf demo) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Column-level Encryption (conceptual + UDF demo) (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

Demo-only: emulate encryption via hashing; real systems should use KMS.

```
from pyspark.sql import functions as F
KEY = F.lit("demo-key")
enc = df.withColumn("enc", F.sha2(F.concat_ws(":", "session_id", KEY), 256))
```

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.

47. Debugging Serialization / Pickling issues: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `order_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 debugging serialization / pickling issues problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Debugging Serialization / Pickling issues (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

Avoid shipping large objects to executors; use broadcast variables.

```
bc = spark.sparkContext.broadcast({"a":1,"b":2})
```

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.

48. Handling Very Wide Schemas: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` 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 handling very wide schemas problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Handling Very Wide Schemas (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/out/sessions")
```

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.

49. Reading Multi-line JSON & Corrupt Records: Advanced Task on `metrics`

Question

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

Task. Using PySpark, implement a robust solution to solve a reading multi-line json & corrupt records problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading Multi-line JSON & Corrupt Records (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

Enable multiLine mode and capture corrupt records for analysis.

```
df = (spark.read.option("multiLine", True)
               .option("mode","PERMISSIVE")
               .json("/data/mljson"))
```

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.

50. Optimizing fromRDD / mapPartitions: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_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 optimizing fromrdd / mappartitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Optimizing fromRDD / mapPartitions (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

Use mapPartitions to amortize per-connection overhead for external I/O.

```
rdd = df.rdd.mapPartitions(lambda it: (x for x in it))
df2 = spark.createDataFrame(rdd, df.schema)
```

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.

51. Window Functions & Analytics: Advanced Task on `metrics`

Question

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

Task. Using PySpark, implement a robust solution to solve a window functions & analytics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Window Functions & Analytics (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

We use window partitions by `customer_id` ordered by `ts` to compute analytics like rolling sums, lag/lead, and first/last. We must guard for null timestamps and ensure a stable ordering. We also consider `rangeBetween` vs `rowsBetween` depending on semantic needs.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("customer_id").orderBy(F.col("ts").cast("timestamp"))

df_clean = (
    df
    .withColumn("ts", F.to_timestamp("ts"))
    .withColumn("amount", F.col("amount").cast("double"))
    .dropna(subset=["customer_id", "ts"])
)

result = (
    df_clean
    .withColumn("prev_amount", F.lag("amount").over(w))
    .withColumn("rolling_sum_3", F.sum("amount").over(w.rowsBetween(-2, 0)))
    .withColumn("rank_desc", F.row_number().over(w.orderBy(F.desc("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.

52. Complex Joins & Skew Handling: Advanced Task on `transactions`

Question

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

Task. Using PySpark, implement a robust solution to solve a complex joins & skew handling problem: - Ingest data with proper schema handling. - Apply necessary transformations

(null-safety, casting, deduplication). - Implement the core logic related to Complex Joins & Skew 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

Skew joins cause a few keys to dominate shuffles. We first profile key frequency, then salt hot keys and broadcast small dimension tables where possible. Enabling AQE can also coalesce skewed partitions. We demonstrate a salting approach.

```
from pyspark.sql import functions as F
from pyspark.sql import types as T

key_freq = df.groupBy("order_id").count()
hot = key_freq.filter("count > 1000000").select("order_id").withColumn("is_hot", F.lit(1
))
df2 = df.join(hot, on="order_id", how="left").fillna({"is_hot":0})

salt_mod = 16
df_salted = df2.withColumn("salt", F.when(F.col("is_hot")==1,
    F.rand()*salt_mod).otherwise(F.lit(0)).cast("int"))

dim_salted = dim.crossJoin(
    spark.range(salt_mod).withColumnRenamed("id","salt")
)
joined = df_salted.join(dim_salted, on=[ "order_id", "salt" ], how="left")
```

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.

53. Nested JSON & Semi-structured Data: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `user_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 nested json & semi-structured data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Nested JSON & Semi-structured 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

For semi-structured inputs, prefer `from_json` with an explicit schema, handle `badRecordsPath`, and use `explode` for arrays. We also guard against nullable subfields and schema drift.

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

schema = T.StructType([
    T.StructField("user_id", T.StringType()),
    T.StructField("created_at", T.TimestampType()),
    T.StructField("payload", T.StructType([
        T.StructField("items", T.ArrayType(T.StructType([
            T.StructField("sku", T.StringType()),
            T.StructField("amount", T.DoubleType())
        ])))
    ]))
])

raw = (spark.read
       .option("multiLine", True)
       .option("badRecordsPath", "/tmp/bad_records")
       .json("/data/transactions/*.json"))

dfj = raw.select(F.from_json(F.col("value").cast("string"), schema).alias("r")).select("
r.*")
items = dfj.select("user_id", "created_at", F.explode_outer("payload.items").alias("it")
)
result = items.select("user_id", "created_at", F.col("it.sku").alias("sku"),
    F.col(f"it.amount").alias("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.

54. UDFs vs Pandas UDFs & Vectorization: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `customer_id`, `created_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 udfs vs pandas udfs & vectorization problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to UDFs vs Pandas UDFs & Vectorization (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("quantity", F.col("quantity").cast("double"))
out = df2.withColumn("z_quantity", zscore(F.col("quantity")))
```

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.

58. File-based Incremental Ingestion: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_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 file-based incremental ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File-based Incremental Ingestion (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

Track high-watermarks and process only new data; design idempotent upserts.

```python
from pyspark.sql import functions as F

prev_max_ts = "2025-01-01T00:00:00"

new_df = (spark.read.parquet("/data/payments")
          .filter(F.col("updated_at") > F.to_timestamp(F.lit(prev_max_ts))))
```

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.

59. Delta Lake Optimize/Z-Order (conceptual with PySpark): Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `order_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 delta lake optimize/z-order (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Delta Lake Optimize/Z-Order (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.order_id = s.order_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

60. CDC/Merge into Delta (conceptual with PySpark): Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `user_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 cdc/merge into delta (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to CDC/Merge into Delta (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.user_id = s.user_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

61. Bucketing, Partitioning & Writer Jobs: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `user_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 bucketing, partitioning & writer jobs problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Bucketing, Partitioning & Writer Jobs (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

General advanced PySpark pattern.

```
pass
```

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.

62. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `session_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 adaptive query execution (aqe) and shuffle partitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Adaptive Query Execution (AQE) and Shuffle Partitions (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

Enable AQE and tune shuffle partitions for better task balance.

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.shuffle.partitions", "200")

dfj = fact.join(F.broadcast(dim), on="session_id", how="left")
```

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.

63. Broadcast Joins and Hints: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `customer_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 broadcast joins and hints problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Broadcast Joins and Hints (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

Broadcast small side tables to avoid shuffles.

```
from pyspark.sql import functions as F
joined = fact.hint("broadcast").join(dim, on="customer_id", how="left")
```

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.

64. Skew Join Salting Techniques: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_id`, `updated_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 skew join salting techniques problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Skew Join Salting Techniques (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

General advanced PySpark pattern.

```
pass
```

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.

65. Aggregations with Complex Grouping Sets: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `user_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 aggregations with complex grouping sets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Aggregations with Complex Grouping Sets (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

Use cube/rollup for multi-level aggregations.

```
from pyspark.sql import functions as F
cube = (df.cube("user_id", "sku").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.

66. Explode + Window Hybrids: Advanced Task on `orders`

Question

Scenario. You have a large `orders` 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 explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (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

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("customer_id", "updated_at", F.explode("items").alias("it"))
w = W.partitionBy("customer_id", "it").orderBy("updated_at")
```

```
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

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.

67. Sessionization (clickstreams): Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` 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 sessionization (clickstreams) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Sessionization (clickstreams) (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

Derive sessions from gaps between events.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("customer_id").orderBy("updated_at")
df2 = (df
       .withColumn("prev_ts", F.lag("updated_at").over(w))
       .withColumn("gap_sec", F.coalesce(F.col("updated_at").cast("long") - F.col("prev_
ts").cast("long"),
    F.lit(0)))
       .withColumn("new_sess", (F.col("gap_sec") > 1800).cast("int"))
       .withColumn("session_seq", F.sum("new_sess").over(w.rowsBetween(Window.unboundedP
receding, 0)))
      )
```

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.

68. Time-series Gaps & Islands: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `device_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 time-series gaps & islands problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Time-series Gaps & Islands (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

Identify contiguous ranges (islands) using row-number differences.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("device_id").orderBy("event_time")
df2 = df.withColumn("rn", F.row_number().over(w))
df3 = df2.withColumn("grp", F.expr("rn - row_number() over (partition by device_id order
 by event_time)"))
```

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.

69. Surrogate Keys & Deduplication: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` 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 surrogate keys & deduplication problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Surrogate Keys & Deduplication (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

Deduplicate by stable ordering and build surrogate keys via hashes.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("device_id").orderBy(F.desc("created_at"))
dedup = (df.withColumn("rn", F.row_number().over(w)).filter("rn = 1").drop("rn"))
with_id = dedup.withColumn("surrogate_id", F.sha2(F.concat_ws("||", *dedup.columns), 256
))
```

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.

70. SCD Type 2 with MERGE logic (Delta/Parquet): Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` 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 scd type 2 with merge logic (delta/parquet) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to SCD Type 2 with MERGE logic (Delta/Parquet) (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

Maintain history via effective_from/to and is_current flags; build updates and closures.

```
# See MERGE example; or implement DataFrame-based SCD2 staging logic.
```

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.

71. Advanced Window: Last non-null forward-fill: Advanced Task on `orders`

Question

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

Task. Using PySpark, implement a robust solution to solve a advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (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

Forward-fill values using `last(..., ignorenulls=True)`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("account_id").orderBy("ts").rowsBetween(Window.unboundedPreceding, 0)
ff = df.withColumn("ff_val", F.last("amount", ignorenulls=True).over(w))
```

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.

72. Top-K per Group at Scale: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `user_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 top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (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

Rank items per group and filter to K.

```
from pyspark.sql import functions as F, Window as W
K = 3
```

```
w = W.partitionBy("user_id").orderBy(F.desc("amount"))
topk = df.withColumn("r", F.row_number().over(w)).filter(F.col("r") <= K).drop("r")
```

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.

73. Rolling Distinct Counts (HLL sketch concept): Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `session_id`, `updated_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 rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (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

Approximate distinct counts per rolling window with `approx_count_distinct`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("session_id").orderBy("updated_at").rowsBetween(-10, 0)
roll = df.withColumn("approx_dc", F.approx_count_distinct("latency_ms").over(w))
```

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.

74. Cross-file Schema Evolution: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` 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 cross-file schema evolution problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Cross-file Schema Evolution (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

75. Dynamic File Pruning: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` 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 dynamic file pruning problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Dynamic File Pruning (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

Partition by time and filter by partition columns for pruning.

```
pruned = spark.read.parquet("/out/transactions").filter(F.col("event_time") >= "2025-01-01")
```

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.

76. Data Quality Checks & Expectations: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` 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 data quality checks & expectations problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Data Quality Checks & Expectations (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

Build rule-based validations; quarantine failures with reasons.

```
from pyspark.sql import functions as F

rules = [
```

```
    ("not_null_key", F.col("order_id").isNotNull()),
    ("val_non_negative", F.col("duration_ms") >= 0)
]

def apply_rules(df):
    for rule_name, cond in rules:
        df = df.withColumn("rule_" + rule_name, cond)
    return df

scored = apply_rules(df)
bad = scored.filter("NOT (rule_not_null_key AND rule_val_non_negative)").withColumn("rea
son",
    F.lit("dq_failed"))
good = scored.filter("rule_not_null_key AND
    rule_val_non_negative").drop("rule_not_null_key","rule_val_non_negative")
```

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.

77. Unit Testing with pytest & chispa: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `session_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 unit testing with pytest & chispa problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Unit Testing with pytest & chispa (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

Use pytest + chispa to assert DataFrame equality; isolate pure transforms.

```
# pip install chispa
from chispa import assert_df_equality

def transform(df):
    return df.filter("amount > 0")

def test_transform(spark):
    input_df = spark.createDataFrame([(1, -1.0), (2, 3.0)], ["id","amount"])
    exp_df = spark.createDataFrame([(2, 3.0)], ["id","amount"])
    assert_df_equality(transform(input_df), exp_df, ignore_column_order=True)
```

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.

78. Performance Debugging with UI & Query Plans: Advanced Task on `events`

Question

Scenario. You have a large `events` 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 performance debugging with ui & query plans problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Performance Debugging with UI & Query Plans (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

Inspect query plans and the Spark UI; avoid Python UDFs and skew.

```
df_explain = df.select("session_id", "duration_ms").groupBy("session_id").agg(F.sum("duration_ms"))
print(df_explain._jdf.queryExecution().toString())
```

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.

80. Reusable Jobs & Parameterized Notebooks: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `customer_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 reusable jobs & parameterized notebooks problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reusable Jobs & Parameterized Notebooks (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

General advanced PySpark pattern.

```
    pass
```

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.

81. DataFrame <-> Spark SQL Interop: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` 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 dataframe <-> spark sql interop problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to DataFrame <-> Spark SQL Interop (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

Register temp views to use Spark SQL alongside DataFrame API.

```
df.createOrReplaceTempView("v")
sql_df = spark.sql("select order_id, sum(duration_ms) as s from v group by order_id")
```

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.

82. Pivot/Unpivot Large Datasets: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `user_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 pivot/unpivot large datasets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Pivot/Unpivot Large Datasets (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

Pivot after pre-aggregating to avoid explosion.

```
from pyspark.sql import functions as F
piv = df.groupBy("user_id").pivot("sku").agg(F.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.

83. Joins over Ranges (temporal joins): Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `user_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 joins over ranges (temporal joins) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Joins over Ranges (temporal joins) (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

Join facts to dimensions where timestamp falls within validity range.

```
joined = fact.join(dim, (fact["ts"].between(dim["start"], dim["end"])) & (fact["user_id"
]==dim["user_id"]),
    "left")
```

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.

84. Windowed UDAFs (via Pandas UDFs): Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` 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 windowed udafs (via pandas udfs) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Windowed UDAFs (via Pandas UDFs) (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas

UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("amount", F.col("amount").cast("double"))
out = df2.withColumn("z_amount", zscore(F.col("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.

85. Binary Files & Image Ingestion: Advanced Task on `orders`

Question

Scenario. You have a large `orders` 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 binary files & image ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Binary Files & Image Ingestion (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

Read binary files for feature extraction or ML preprocessing.

```
images = spark.read.format("binaryFile").load("/data/images/*")
```

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.

86. Graph-Style Problems without GraphFrames: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `session_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 graph-style problems without graphframes problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Graph-Style Problems without GraphFrames (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

Approximate graph analytics via SQL/DF ops (degree counts, simple traversals).

```
edges = spark.createDataFrame([("a","b"),("b","c")], ["src","dst"])
deg = (edges.select("src").union(edges.select("dst"))
            .groupBy("src").count())
```

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.

87. MLlib Pipelines with Custom Transformers: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `user_id`, `created_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 mllib pipelines with custom transformers problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to MLlib Pipelines with Custom Transformers (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

Build ML pipelines; ensure proper vectorization and column roles.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

va = VectorAssembler(inputCols=["f1","f2","f3"], outputCol="features")
lr = LogisticRegression(featuresCol="features", labelCol="label")
pipe = Pipeline(stages=[va, lr]).fit(train_df)
```

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.

92. File Compaction Job: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `account_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 file compaction job problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File Compaction Job (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/impressions")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/impressions_compacted"))
```

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.

93. Small-file Problem Mitigation: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_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 small-file problem mitigation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Small-file Problem Mitigation (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/payments")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/payments_compacted"))
```

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.

94. Reading from Hive Metastore & External Tables: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `customer_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 reading from hive metastore & external tables problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading from Hive Metastore & External Tables (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

Integrate with Hive catalog; repair partitions; manage external tables.

```
spark.sql("MSCK REPAIR TABLE db.tbl")
```

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.

95. Security & PII Masking Patterns: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `session_id`, `updated_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 security & pii masking patterns problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Security & PII Masking Patterns (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

Mask/obfuscate sensitive columns; restrict access via views/catalog controls.

```
from pyspark.sql import functions as F
masked = df.withColumn("email_masked", F.sha2(F.col("email"), 256))
```

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.

96. Column-level Encryption (conceptual + UDF demo): Advanced Task on `events`

Question

Scenario. You have a large `events` 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 column-level encryption (conceptual + udf demo) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Column-level Encryption (conceptual + UDF demo) (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

Demo-only: emulate encryption via hashing; real systems should use KMS.

```
from pyspark.sql import functions as F
KEY = F.lit("demo-key")
enc = df.withColumn("enc", F.sha2(F.concat_ws(":", "user_id", KEY), 256))
```

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.

97. Debugging Serialization / Pickling issues: Advanced Task on `clicks`

Question

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

Task. Using PySpark, implement a robust solution to solve a debugging serialization / pickling issues problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Debugging Serialization / Pickling issues (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

Avoid shipping large objects to executors; use broadcast variables.

```
bc = spark.sparkContext.broadcast({"a":1,"b":2})
```

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.

98. Handling Very Wide Schemas: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `customer_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 handling very wide schemas problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Handling Very Wide Schemas (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

99. Reading Multi-line JSON & Corrupt Records: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `account_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 reading multi-line json & corrupt records problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading Multi-line JSON & Corrupt Records (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

Enable multiLine mode and capture corrupt records for analysis.

```
df = (spark.read.option("multiLine", True)
              .option("mode","PERMISSIVE")
              .json("/data/mljson"))
```

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.

100. Optimizing fromRDD / mapPartitions: Advanced Task on `logs`

Question

Scenario. You have a large `logs` 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 optimizing fromrdd / mappartitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Optimizing fromRDD / mapPartitions (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

Use mapPartitions to amortize per-connection overhead for external I/O.

```
rdd = df.rdd.mapPartitions(lambda it: (x for x in it))
df2 = spark.createDataFrame(rdd, df.schema)
```

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.

101. Window Functions & Analytics: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `account_id`, `updated_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 window functions & analytics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Window Functions & Analytics (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

We use window partitions by `account_id` ordered by `updated_at` to compute analytics like rolling sums, lag/lead, and first/last. We must guard for null timestamps and ensure a stable ordering. We also consider `rangeBetween` vs `rowsBetween` depending on semantic needs.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("account_id").orderBy(F.col("updated_at").cast("timestamp"))

df_clean = (
    df
    .withColumn("updated_at", F.to_timestamp("updated_at"))
    .withColumn("latency_ms", F.col("latency_ms").cast("double"))
    .dropna(subset=["account_id", "updated_at"])
)

result = (
    df_clean
    .withColumn("prev_latency_ms", F.lag("latency_ms").over(w))
    .withColumn("rolling_sum_3", F.sum("latency_ms").over(w.rowsBetween(-2, 0)))
    .withColumn("rank_desc", F.row_number().over(w.orderBy(F.desc("latency_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.

102. Complex Joins & Skew Handling: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `customer_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 complex joins & skew handling problem: - Ingest data with proper schema handling. - Apply necessary transformations

(null-safety, casting, deduplication). - Implement the core logic related to Complex Joins & Skew 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

Skew joins cause a few keys to dominate shuffles. We first profile key frequency, then salt hot keys and broadcast small dimension tables where possible. Enabling AQE can also coalesce skewed partitions. We demonstrate a salting approach.

```
from pyspark.sql import functions as F
from pyspark.sql import types as T

key_freq = df.groupBy("customer_id").count()
hot = key_freq.filter("count > 1000000").select("customer_id").withColumn("is_hot", F.li
t(1))
df2 = df.join(hot, on="customer_id", how="left").fillna({"is_hot":0})

salt_mod = 16
df_salted = df2.withColumn("salt", F.when(F.col("is_hot")==1,
    F.rand()*salt_mod).otherwise(F.lit(0)).cast("int"))

dim_salted = dim.crossJoin(
    spark.range(salt_mod).withColumnRenamed("id","salt")
)
joined = df_salted.join(dim_salted, on=[ "customer_id", "salt" ], how="left")
```

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.

103. Nested JSON & Semi-structured Data: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `user_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 nested json & semi-structured data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Nested JSON & Semi-structured 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

For semi-structured inputs, prefer `from_json` with an explicit schema, handle `badRecordsPath`, and use `explode` for arrays. We also guard against nullable subfields and schema drift.

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

schema = T.StructType([
    T.StructField("user_id", T.StringType()),
    T.StructField("ts", T.TimestampType()),
    T.StructField("payload", T.StructType([
        T.StructField("items", T.ArrayType(T.StructType([
            T.StructField("sku", T.StringType()),
            T.StructField("quantity", T.DoubleType())
        ])))
    ]))
])

raw = (spark.read
       .option("multiLine", True)
       .option("badRecordsPath", "/tmp/bad_records")
       .json("/data/payments/*.json"))

dfj = raw.select(F.from_json(F.col("value").cast("string"), schema).alias("r")).select("
r.*")
items = dfj.select("user_id", "ts", F.explode_outer("payload.items").alias("it"))
result = items.select("user_id", "ts", F.col("it.sku").alias("sku"), F.col(f"it.quantity
").alias("quantity"))
```

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.

104. UDFs vs Pandas UDFs & Vectorization: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `device_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 udfs vs pandas udfs & vectorization problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to UDFs vs Pandas UDFs & Vectorization (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("score", F.col("score").cast("double"))
out = df2.withColumn("z_score", zscore(F.col("score")))
```

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.

108. File-based Incremental Ingestion: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_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 file-based incremental ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File-based Incremental Ingestion (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

Track high-watermarks and process only new data; design idempotent upserts.

```
from pyspark.sql import functions as F

prev_max_ts = "2025-01-01T00:00:00"

new_df = (spark.read.parquet("/data/orders")
          .filter(F.col("ts") > F.to_timestamp(F.lit(prev_max_ts))))
```

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.

109. Delta Lake Optimize/Z-Order (conceptual with PySpark): Advanced Task on `logs`

Question

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

Task. Using PySpark, implement a robust solution to solve a delta lake optimize/z-order (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Delta Lake Optimize/Z-Order (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.account_id = s.account_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

110. CDC/Merge into Delta (conceptual with PySpark): Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `order_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 cdc/merge into delta (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to CDC/Merge into Delta (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.order_id = s.order_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

111. Bucketing, Partitioning & Writer Jobs: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `account_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 bucketing, partitioning & writer jobs problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Bucketing, Partitioning & Writer Jobs (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

General advanced PySpark pattern.

```
pass
```

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.

112. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` 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 adaptive query execution (aqe) and shuffle partitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Adaptive Query Execution (AQE) and Shuffle Partitions (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

Enable AQE and tune shuffle partitions for better task balance.

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.shuffle.partitions", "200")
```

```
dfj = fact.join(F.broadcast(dim), on="customer_id", how="left")
```

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.

113. Broadcast Joins and Hints: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` 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 broadcast joins and hints problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Broadcast Joins and Hints (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

Broadcast small side tables to avoid shuffles.

```
from pyspark.sql import functions as F
joined = fact.hint("broadcast").join(dim, on="session_id", how="left")
```

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.

114. Skew Join Salting Techniques: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `device_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 skew join salting techniques problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Skew Join Salting Techniques (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

General advanced PySpark pattern.

```
pass
```

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.

115. Aggregations with Complex Grouping Sets: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `account_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 aggregations with complex grouping sets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Aggregations with Complex Grouping Sets (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

Use cube/rollup for multi-level aggregations.

```
from pyspark.sql import functions as F
cube = (df.cube("account_id", "sku").agg(F.sum("quantity").alias("sum_quantity")))
```

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.

116. Explode + Window Hybrids: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `user_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 explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (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

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("user_id", "ts", F.explode("items").alias("it"))
w = W.partitionBy("user_id", "it").orderBy("ts")
```

```
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

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.

117. Sessionization (clickstreams): Advanced Task on `payments`

Question

Scenario. You have a large payments dataset with columns like user_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 sessionization (clickstreams) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Sessionization (clickstreams) (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

Derive sessions from gaps between events.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("user_id").orderBy("created_at")
df2 = (df
       .withColumn("prev_ts", F.lag("created_at").over(w))
       .withColumn("gap_sec", F.coalesce(F.col("created_at").cast("long") - F.col("prev_
ts").cast("long"),
    F.lit(0)))
       .withColumn("new_sess", (F.col("gap_sec") > 1800).cast("int"))
       .withColumn("session_seq", F.sum("new_sess").over(w.rowsBetween(Window.unboundedP
receding, 0)))
      )
```

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.

118. Time-series Gaps & Islands: Advanced Task on `logs`

Question

Scenario. You have a large logs dataset with columns like order_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 time-series gaps & islands problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Time-series Gaps & Islands (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

Identify contiguous ranges (islands) using row-number differences.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("order_id").orderBy("event_time")
df2 = df.withColumn("rn", F.row_number().over(w))
df3 = df2.withColumn("grp", F.expr("rn - row_number() over (partition by order_id order
by event_time)"))
```

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.

119. Surrogate Keys & Deduplication: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `session_id`, `created_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 surrogate keys & deduplication problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Surrogate Keys & Deduplication (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

Deduplicate by stable ordering and build surrogate keys via hashes.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("session_id").orderBy(F.desc("created_at"))
dedup = (df.withColumn("rn", F.row_number().over(w)).filter("rn = 1").drop("rn"))
with_id = dedup.withColumn("surrogate_id", F.sha2(F.concat_ws("||", *dedup.columns), 256
))
```

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.

120. SCD Type 2 with MERGE logic (Delta/Parquet): Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_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 scd type 2 with merge logic (delta/parquet) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to SCD Type 2 with MERGE logic (Delta/Parquet) (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

Maintain history via effective_from/to and is_current flags; build updates and closures.

```
# See MERGE example; or implement DataFrame-based SCD2 staging logic.
```

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.

121. Advanced Window: Last non-null forward-fill: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `session_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 advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (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

Forward-fill values using `last(..., ignorenulls=True)`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("session_id").orderBy("ts").rowsBetween(Window.unboundedPreceding, 0)
ff = df.withColumn("ff_val", F.last("value", ignorenulls=True).over(w))
```

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.

122. Top-K per Group at Scale: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `order_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 top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (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

Rank items per group and filter to K.

```
from pyspark.sql import functions as F, Window as W
K = 3
```

```
w = W.partitionBy("order_id").orderBy(F.desc("quantity"))
topk = df.withColumn("r", F.row_number().over(w)).filter(F.col("r") <= K).drop("r")
```

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.

123. Rolling Distinct Counts (HLL sketch concept): Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `session_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 rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (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

Approximate distinct counts per rolling window with `approx_count_distinct`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("session_id").orderBy("created_at").rowsBetween(-10, 0)
roll = df.withColumn("approx_dc", F.approx_count_distinct("duration_ms").over(w))
```

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.

124. Cross-file Schema Evolution: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `user_id`, `created_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 cross-file schema evolution problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Cross-file Schema Evolution (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

125. Dynamic File Pruning: Advanced Task on `orders`

Question

Scenario. You have a large `orders` 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 dynamic file pruning problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Dynamic File Pruning (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

Partition by time and filter by partition columns for pruning.

```
pruned = spark.read.parquet("/out/orders").filter(F.col("created_at") >= "2025-01-01")
```

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.

126. Data Quality Checks & Expectations: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `device_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 data quality checks & expectations problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Data Quality Checks & Expectations (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

Build rule-based validations; quarantine failures with reasons.

```
from pyspark.sql import functions as F

rules = [
    ("not_null_key", F.col("device_id").isNotNull()),
```

```
    ("val_non_negative", F.col("latency_ms") >= 0)
]

def apply_rules(df):
    for rule_name, cond in rules:
        df = df.withColumn("rule_" + rule_name, cond)
    return df

scored = apply_rules(df)
bad = scored.filter("NOT (rule_not_null_key AND rule_val_non_negative)").withColumn("rea
son",
    F.lit("dq_failed"))
good = scored.filter("rule_not_null_key AND
    rule_val_non_negative").drop("rule_not_null_key","rule_val_non_negative")
```

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.

127. Unit Testing with pytest & chispa: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `order_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 unit testing with pytest & chispa problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Unit Testing with pytest & chispa (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

Use pytest + chispa to assert DataFrame equality; isolate pure transforms.

```
# pip install chispa
from chispa import assert_df_equality

def transform(df):
    return df.filter("amount > 0")

def test_transform(spark):
    input_df = spark.createDataFrame([(1, -1.0), (2, 3.0)], ["id","amount"])
    exp_df = spark.createDataFrame([(2, 3.0)], ["id","amount"])
    assert_df_equality(transform(input_df), exp_df, ignore_column_order=True)
```

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.

128. Performance Debugging with UI & Query Plans: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `account_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 performance debugging with ui & query plans problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Performance Debugging with UI & Query Plans (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

Inspect query plans and the Spark UI; avoid Python UDFs and skew.

```
df_explain = df.select("account_id", "amount").groupBy("account_id").agg(F.sum("amount")
)
print(df_explain._jdf.queryExecution().toString())
```

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.

130. Reusable Jobs & Parameterized Notebooks: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `customer_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 reusable jobs & parameterized notebooks problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reusable Jobs & Parameterized Notebooks (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

General advanced PySpark pattern.

```
    pass
```

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.

131. DataFrame <-> Spark SQL Interop: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `device_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 dataframe <-> spark sql interop problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to DataFrame <-> Spark SQL Interop (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

Register temp views to use Spark SQL alongside DataFrame API.

```
df.createOrReplaceTempView("v")
sql_df = spark.sql("select device_id, sum(duration_ms) as s from v group by device_id")
```

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.

132. Pivot/Unpivot Large Datasets: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `order_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 pivot/unpivot large datasets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Pivot/Unpivot Large Datasets (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

Pivot after pre-aggregating to avoid explosion.

```
from pyspark.sql import functions as F
piv = df.groupBy("order_id").pivot("sku").agg(F.sum("quantity"))
```

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.

133. Joins over Ranges (temporal joins): Advanced Task on `payments`

Question

Scenario. You have a large `payments` 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 joins over ranges (temporal joins) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Joins over Ranges (temporal joins) (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

Join facts to dimensions where timestamp falls within validity range.

```
joined = fact.join(dim, (fact["updated_at"].between(dim["start"], dim["end"])) &
    (fact["device_id"]==dim["device_id"]), "left")
```

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.

134. Windowed UDAFs (via Pandas UDFs): Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `order_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 windowed udafs (via pandas udfs) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Windowed UDAFs (via Pandas UDFs) (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```python
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("amount", F.col("amount").cast("double"))
out = df2.withColumn("z_amount", zscore(F.col("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.

135. Binary Files & Image Ingestion: Advanced Task on `payments`

Question

Scenario. You have a large `payments` 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 binary files & image ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Binary Files & Image Ingestion (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

Read binary files for feature extraction or ML preprocessing.

```
images = spark.read.format("binaryFile").load("/data/images/*")
```

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.

136. Graph-Style Problems without GraphFrames: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `device_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 graph-style problems without graphframes problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Graph-Style Problems without GraphFrames (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

Approximate graph analytics via SQL/DF ops (degree counts, simple traversals).

```
edges = spark.createDataFrame([("a","b"),("b","c")], ["src","dst"])
deg = (edges.select("src").union(edges.select("dst"))
         .groupBy("src").count())
```

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.

137. MLlib Pipelines with Custom Transformers: Advanced Task on `orders`

Question

Scenario. You have a large `orders` 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 mllib pipelines with custom transformers problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to MLlib Pipelines with Custom Transformers (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

Build ML pipelines; ensure proper vectorization and column roles.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

va = VectorAssembler(inputCols=["f1","f2","f3"], outputCol="features")
lr = LogisticRegression(featuresCol="features", labelCol="label")
pipe = Pipeline(stages=[va, lr]).fit(train_df)
```

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.

142. File Compaction Job: Advanced Task on `orders`

Question

Scenario. You have a large `orders` 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 file compaction job problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File Compaction Job (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/orders")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/orders_compacted"))
```

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.

143. Small-file Problem Mitigation: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `customer_id`, `created_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 small-file problem mitigation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Small-file Problem Mitigation (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/sessions")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/sessions_compacted"))
```

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.

144. Reading from Hive Metastore & External Tables: Advanced Task on `logs`

Question

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

Task. Using PySpark, implement a robust solution to solve a reading from hive metastore & external tables problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading from Hive Metastore & External Tables (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

Integrate with Hive catalog; repair partitions; manage external tables.

```
spark.sql("MSCK REPAIR TABLE db.tbl")
```

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.

145. Security & PII Masking Patterns: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `customer_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 security & pii masking patterns problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Security & PII Masking Patterns (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

Mask/obfuscate sensitive columns; restrict access via views/catalog controls.

```
from pyspark.sql import functions as F
masked = df.withColumn("email_masked", F.sha2(F.col("email"), 256))
```

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.

146. Column-level Encryption (conceptual + UDF demo): Advanced Task on `clicks`

Question

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

Task. Using PySpark, implement a robust solution to solve a column-level encryption (conceptual + udf demo) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Column-level Encryption (conceptual + UDF demo) (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

Demo-only: emulate encryption via hashing; real systems should use KMS.

```
from pyspark.sql import functions as F
KEY = F.lit("demo-key")
enc = df.withColumn("enc", F.sha2(F.concat_ws(":", "order_id", KEY), 256))
```

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.

147. Debugging Serialization / Pickling issues: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `device_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 debugging serialization / pickling issues problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Debugging Serialization / Pickling issues (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

Avoid shipping large objects to executors; use broadcast variables.

```
bc = spark.sparkContext.broadcast({"a":1,"b":2})
```

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.

148. Handling Very Wide Schemas: Advanced Task on `impressions`

Question

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

Task. Using PySpark, implement a robust solution to solve a handling very wide schemas problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Handling Very Wide Schemas (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

149. Reading Multi-line JSON & Corrupt Records: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `device_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 reading multi-line json & corrupt records problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading Multi-line JSON & Corrupt Records (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

Enable multiLine mode and capture corrupt records for analysis.

```
df = (spark.read.option("multiLine", True)
               .option("mode","PERMISSIVE")
               .json("/data/mljson"))
```

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.

150. Optimizing fromRDD / mapPartitions: Advanced Task on `payments`

Question

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

Task. Using PySpark, implement a robust solution to solve a optimizing fromrdd / mappartitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Optimizing fromRDD / mapPartitions (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

Use mapPartitions to amortize per-connection overhead for external I/O.

```
rdd = df.rdd.mapPartitions(lambda it: (x for x in it))
df2 = spark.createDataFrame(rdd, df.schema)
```

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.

151. Window Functions & Analytics: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `device_id`, `updated_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 window functions & analytics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Window Functions & Analytics (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

We use window partitions by `device_id` ordered by `updated_at` to compute analytics like rolling sums, lag/lead, and first/last. We must guard for null timestamps and ensure a stable ordering. We also consider `rangeBetween` vs `rowsBetween` depending on semantic needs.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("device_id").orderBy(F.col("updated_at").cast("timestamp"))

df_clean = (
    df
    .withColumn("updated_at", F.to_timestamp("updated_at"))
    .withColumn("latency_ms", F.col("latency_ms").cast("double"))
    .dropna(subset=["device_id", "updated_at"])
)

result = (
    df_clean
    .withColumn("prev_latency_ms", F.lag("latency_ms").over(w))
    .withColumn("rolling_sum_3", F.sum("latency_ms").over(w.rowsBetween(-2, 0)))
    .withColumn("rank_desc", F.row_number().over(w.orderBy(F.desc("latency_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.

152. Complex Joins & Skew Handling: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `session_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 complex joins & skew handling problem: - Ingest data with proper schema handling. - Apply necessary transformations

(null-safety, casting, deduplication). - Implement the core logic related to Complex Joins & Skew 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

Skew joins cause a few keys to dominate shuffles. We first profile key frequency, then salt hot keys and broadcast small dimension tables where possible. Enabling AQE can also coalesce skewed partitions. We demonstrate a salting approach.

```
from pyspark.sql import functions as F
from pyspark.sql import types as T

key_freq = df.groupBy("session_id").count()
hot = key_freq.filter("count > 1000000").select("session_id").withColumn("is_hot", F.lit
(1))
df2 = df.join(hot, on="session_id", how="left").fillna({"is_hot":0})

salt_mod = 16
df_salted = df2.withColumn("salt", F.when(F.col("is_hot")==1,
    F.rand()*salt_mod).otherwise(F.lit(0)).cast("int"))

dim_salted = dim.crossJoin(
    spark.range(salt_mod).withColumnRenamed("id","salt")
)
joined = df_salted.join(dim_salted, on=[ "session_id", "salt" ], how="left")
```

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.

153. Nested JSON & Semi-structured Data: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `account_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 nested json & semi-structured data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Nested JSON & Semi-structured 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

For semi-structured inputs, prefer `from_json` with an explicit schema, handle `badRecordsPath`, and use `explode` for arrays. We also guard against nullable subfields and schema drift.

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

schema = T.StructType([
    T.StructField("account_id", T.StringType()),
    T.StructField("updated_at", T.TimestampType()),
    T.StructField("payload", T.StructType([
        T.StructField("items", T.ArrayType(T.StructType([
            T.StructField("sku", T.StringType()),
            T.StructField("duration_ms", T.DoubleType())
        ])))
    ]))
])

raw = (spark.read
       .option("multiLine", True)
       .option("badRecordsPath", "/tmp/bad_records")
       .json("/data/orders/*.json"))

dfj = raw.select(F.from_json(F.col("value").cast("string"), schema).alias("r")).select("
r.*")
items = dfj.select("account_id", "updated_at", F.explode_outer("payload.items").alias("i
t"))
result = items.select("account_id", "updated_at", F.col("it.sku").alias("sku"),
    F.col(f"it.duration_ms").alias("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.

154. UDFs vs Pandas UDFs & Vectorization: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `device_id`, `created_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 udfs vs pandas udfs & vectorization problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to UDFs vs Pandas UDFs & Vectorization (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("quantity", F.col("quantity").cast("double"))
out = df2.withColumn("z_quantity", zscore(F.col("quantity")))
```

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.

158. File-based Incremental Ingestion: Advanced Task on `impressions`

Question

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

Task. Using PySpark, implement a robust solution to solve a file-based incremental ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File-based Incremental Ingestion (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

Track high-watermarks and process only new data; design idempotent upserts.

```
from pyspark.sql import functions as F

prev_max_ts = "2025-01-01T00:00:00"

new_df = (spark.read.parquet("/data/impressions")
          .filter(F.col("event_time") > F.to_timestamp(F.lit(prev_max_ts))))
```

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.

159. Delta Lake Optimize/Z-Order (conceptual with PySpark): Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_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 delta lake optimize/z-order (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Delta Lake Optimize/Z-Order (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.account_id = s.account_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

160. CDC/Merge into Delta (conceptual with PySpark): Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `account_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 cdc/merge into delta (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to CDC/Merge into Delta (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.account_id = s.account_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

161. Bucketing, Partitioning & Writer Jobs: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_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 bucketing, partitioning & writer jobs problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Bucketing, Partitioning & Writer Jobs (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

General advanced PySpark pattern.

```
pass
```

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.

162. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `events`

Question

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

Task. Using PySpark, implement a robust solution to solve a adaptive query execution (aqe) and shuffle partitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Adaptive Query Execution (AQE) and Shuffle Partitions (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

Enable AQE and tune shuffle partitions for better task balance.

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.shuffle.partitions", "200")
```

```
dfj = fact.join(F.broadcast(dim), on="order_id", how="left")
```

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.

163. Broadcast Joins and Hints: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `order_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 broadcast joins and hints problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Broadcast Joins and Hints (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

Broadcast small side tables to avoid shuffles.

```
from pyspark.sql import functions as F
joined = fact.hint("broadcast").join(dim, on="order_id", how="left")
```

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.

164. Skew Join Salting Techniques: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `user_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 skew join salting techniques problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Skew Join Salting Techniques (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

General advanced PySpark pattern.

```
pass
```

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.

165. Aggregations with Complex Grouping Sets: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `session_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 aggregations with complex grouping sets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Aggregations with Complex Grouping Sets (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

Use cube/rollup for multi-level aggregations.

```
from pyspark.sql import functions as F
cube = (df.cube("session_id", "sku").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.

166. Explode + Window Hybrids: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `device_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 explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (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

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("device_id", "event_time", F.explode("items").alias("it"))
w = W.partitionBy("device_id", "it").orderBy("event_time")
```

```
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

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.

167. Sessionization (clickstreams): Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `device_id`, `created_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 sessionization (clickstreams) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Sessionization (clickstreams) (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

Derive sessions from gaps between events.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("device_id").orderBy("created_at")
df2 = (df
       .withColumn("prev_ts", F.lag("created_at").over(w))
       .withColumn("gap_sec", F.coalesce(F.col("created_at").cast("long") - F.col("prev_
ts").cast("long"),
    F.lit(0)))
       .withColumn("new_sess", (F.col("gap_sec") > 1800).cast("int"))
       .withColumn("session_seq", F.sum("new_sess").over(w.rowsBetween(Window.unboundedP
receding, 0)))
      )
```

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.

168. Time-series Gaps & Islands: Advanced Task on `clicks`

Question

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

Task. Using PySpark, implement a robust solution to solve a time-series gaps & islands problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Time-series Gaps & Islands (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

Identify contiguous ranges (islands) using row-number differences.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("session_id").orderBy("event_time")
df2 = df.withColumn("rn", F.row_number().over(w))
df3 = df2.withColumn("grp", F.expr("rn - row_number() over (partition by session_id orde
r by event_time)"))
```

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.

169. Surrogate Keys & Deduplication: Advanced Task on `sessions`

Question

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

Task. Using PySpark, implement a robust solution to solve a surrogate keys & deduplication problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Surrogate Keys & Deduplication (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

Deduplicate by stable ordering and build surrogate keys via hashes.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("user_id").orderBy(F.desc("event_time"))
dedup = (df.withColumn("rn", F.row_number().over(w)).filter("rn = 1").drop("rn"))
with_id = dedup.withColumn("surrogate_id", F.sha2(F.concat_ws("||", *dedup.columns), 256
))
```

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.

170. SCD Type 2 with MERGE logic (Delta/Parquet): Advanced Task on `payments`

Question

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

Task. Using PySpark, implement a robust solution to solve a scd type 2 with merge logic (delta/parquet) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to SCD Type 2 with MERGE logic (Delta/Parquet) (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

Maintain history via effective_from/to and is_current flags; build updates and closures.

```
# See MERGE example; or implement DataFrame-based SCD2 staging logic.
```

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.

171. Advanced Window: Last non-null forward-fill: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_id`, `created_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 advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (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

Forward-fill values using `last(..., ignorenulls=True)`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("user_id").orderBy("created_at").rowsBetween(Window.unboundedPreceding
, 0)
ff = df.withColumn("ff_val", F.last("quantity", ignorenulls=True).over(w))
```

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.

172. Top-K per Group at Scale: Advanced Task on `transactions`

Question

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

Task. Using PySpark, implement a robust solution to solve a top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (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

Rank items per group and filter to K.

```
from pyspark.sql import functions as F, Window as W
K = 3
w = W.partitionBy("session_id").orderBy(F.desc("score"))
topk = df.withColumn("r", F.row_number().over(w)).filter(F.col("r") <= K).drop("r")
```

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.

173. Rolling Distinct Counts (HLL sketch concept): Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `device_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 rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (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

Approximate distinct counts per rolling window with `approx_count_distinct`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("device_id").orderBy("created_at").rowsBetween(-10, 0)
roll = df.withColumn("approx_dc", F.approx_count_distinct("amount").over(w))
```

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.

174. Cross-file Schema Evolution: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `order_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 cross-file schema evolution problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Cross-file Schema Evolution (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

175. Dynamic File Pruning: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `device_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 dynamic file pruning problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Dynamic File Pruning (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

Partition by time and filter by partition columns for pruning.

```
pruned = spark.read.parquet("/out/payments").filter(F.col("event_time") >= "2025-01-01")
```

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.

176. Data Quality Checks & Expectations: Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `order_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 data quality checks & expectations problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Data Quality Checks & Expectations (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

Build rule-based validations; quarantine failures with reasons.

```
from pyspark.sql import functions as F

rules = [
    ("not_null_key", F.col("order_id").isNotNull()),
    ("val_non_negative", F.col("quantity") >= 0)
]
```

```
def apply_rules(df):
    for rule_name, cond in rules:
        df = df.withColumn("rule_" + rule_name, cond)
    return df

scored = apply_rules(df)
bad = scored.filter("NOT (rule_not_null_key AND rule_val_non_negative)").withColumn("rea
son",
    F.lit("dq_failed"))
good = scored.filter("rule_not_null_key AND
    rule_val_non_negative").drop("rule_not_null_key","rule_val_non_negative")
```

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.

177. Unit Testing with pytest & chispa: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `device_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 unit testing with pytest & chispa problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Unit Testing with pytest & chispa (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

Use pytest + chispa to assert DataFrame equality; isolate pure transforms.

```
# pip install chispa
from chispa import assert_df_equality

def transform(df):
    return df.filter("amount > 0")

def test_transform(spark):
    input_df = spark.createDataFrame([(1, -1.0), (2, 3.0)], ["id","amount"])
    exp_df = spark.createDataFrame([(2, 3.0)], ["id","amount"])
    assert_df_equality(transform(input_df), exp_df, ignore_column_order=True)
```

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.

178. Performance Debugging with UI & Query Plans: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `order_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 performance debugging with ui & query plans problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Performance Debugging with UI & Query Plans (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

Inspect query plans and the Spark UI; avoid Python UDFs and skew.

```
df_explain = df.select("order_id", "latency_ms").groupBy("order_id").agg(F.sum("latency_
ms"))
print(df_explain._jdf.queryExecution().toString())
```

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.

180. Reusable Jobs & Parameterized Notebooks: Advanced Task on `sessions`

Question

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

Task. Using PySpark, implement a robust solution to solve a reusable jobs & parameterized notebooks problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reusable Jobs & Parameterized Notebooks (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

General advanced PySpark pattern.

```
    pass
```

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.

181. DataFrame <-> Spark SQL Interop: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `device_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 dataframe <-> spark sql interop problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to DataFrame <-> Spark SQL Interop (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

Register temp views to use Spark SQL alongside DataFrame API.

```
df.createOrReplaceTempView("v")
sql_df = spark.sql("select device_id, sum(duration_ms) as s from v group by device_id")
```

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.

182. Pivot/Unpivot Large Datasets: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_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 pivot/unpivot large datasets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Pivot/Unpivot Large Datasets (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

Pivot after pre-aggregating to avoid explosion.

```
from pyspark.sql import functions as F
piv = df.groupBy("account_id").pivot("sku").agg(F.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.

183. Joins over Ranges (temporal joins): Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `customer_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 joins over ranges (temporal joins) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Joins over Ranges (temporal joins) (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

Join facts to dimensions where timestamp falls within validity range.

```
joined = fact.join(dim, (fact["updated_at"].between(dim["start"], dim["end"])) &
    (fact["customer_id"]==dim["customer_id"]), "left")
```

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.

184. Windowed UDAFs (via Pandas UDFs): Advanced Task on `logs`

Question

Scenario. You have a large `logs` dataset with columns like `account_id`, `created_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 windowed udafs (via pandas udfs) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Windowed UDAFs (via Pandas UDFs) (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd
```

```python
@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("score", F.col("score").cast("double"))
out = df2.withColumn("z_score", zscore(F.col("score")))
```

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.

185. Binary Files & Image Ingestion: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` 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 binary files & image ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Binary Files & Image Ingestion (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

Read binary files for feature extraction or ML preprocessing.

```
images = spark.read.format("binaryFile").load("/data/images/*")
```

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.

186. Graph-Style Problems without GraphFrames: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_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 graph-style problems without graphframes problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Graph-Style Problems without GraphFrames (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

Approximate graph analytics via SQL/DF ops (degree counts, simple traversals).

```
edges = spark.createDataFrame([("a","b"),("b","c")], ["src","dst"])
deg = (edges.select("src").union(edges.select("dst"))
          .groupBy("src").count())
```

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.

187. MLlib Pipelines with Custom Transformers: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `device_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 mllib pipelines with custom transformers problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to MLlib Pipelines with Custom Transformers (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

Build ML pipelines; ensure proper vectorization and column roles.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

va = VectorAssembler(inputCols=["f1","f2","f3"], outputCol="features")
lr = LogisticRegression(featuresCol="features", labelCol="label")
pipe = Pipeline(stages=[va, lr]).fit(train_df)
```

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.

192. File Compaction Job: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `account_id`, `created_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 file compaction job problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File Compaction Job (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/metrics")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/metrics_compacted"))
```

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.

193. Small-file Problem Mitigation: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `account_id`, `created_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 small-file problem mitigation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Small-file Problem Mitigation (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/impressions")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/impressions_compacted"))
```

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.

194. Reading from Hive Metastore & External Tables: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `user_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 reading from hive metastore & external tables problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading from Hive Metastore & External Tables (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

Integrate with Hive catalog; repair partitions; manage external tables.

```
spark.sql("MSCK REPAIR TABLE db.tbl")
```

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.

195. Security & PII Masking Patterns: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` 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 security & pii masking patterns problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Security & PII Masking Patterns (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

Mask/obfuscate sensitive columns; restrict access via views/catalog controls.

```
from pyspark.sql import functions as F
masked = df.withColumn("email_masked", F.sha2(F.col("email"), 256))
```

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.

196. Column-level Encryption (conceptual + UDF demo): Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `order_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 column-level encryption (conceptual + udf demo) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Column-level Encryption (conceptual + UDF demo) (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

Demo-only: emulate encryption via hashing; real systems should use KMS.

```
from pyspark.sql import functions as F
KEY = F.lit("demo-key")
```

```
enc = df.withColumn("enc", F.sha2(F.concat_ws(":", "order_id", KEY), 256))
```

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.

197. Debugging Serialization / Pickling issues: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` 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 debugging serialization / pickling issues problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Debugging Serialization / Pickling issues (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

Avoid shipping large objects to executors; use broadcast variables.

```
bc = spark.sparkContext.broadcast({"a":1,"b":2})
```

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.

198. Handling Very Wide Schemas: 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 handling very wide schemas problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Handling Very Wide Schemas (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

199. Reading Multi-line JSON & Corrupt Records: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `account_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 reading multi-line json & corrupt records problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading Multi-line JSON & Corrupt Records (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

Enable multiLine mode and capture corrupt records for analysis.

```
df = (spark.read.option("multiLine", True)
               .option("mode","PERMISSIVE")
               .json("/data/mljson"))
```

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.

200. Optimizing fromRDD / mapPartitions: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` 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 optimizing fromrdd / mappartitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Optimizing fromRDD / mapPartitions (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

Use mapPartitions to amortize per-connection overhead for external I/O.

```
rdd = df.rdd.mapPartitions(lambda it: (x for x in it))
df2 = spark.createDataFrame(rdd, df.schema)
```

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.

201. Window Functions & Analytics: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `session_id`, `created_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 window functions & analytics problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Window Functions & Analytics (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

We use window partitions by `session_id` ordered by `created_at` to compute analytics like rolling sums, lag/lead, and first/last. We must guard for null timestamps and ensure a stable ordering. We also consider `rangeBetween` vs `rowsBetween` depending on semantic needs.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("session_id").orderBy(F.col("created_at").cast("timestamp"))

df_clean = (
    df
    .withColumn("created_at", F.to_timestamp("created_at"))
    .withColumn("quantity", F.col("quantity").cast("double"))
    .dropna(subset=["session_id", "created_at"])
)

result = (
    df_clean
    .withColumn("prev_quantity", F.lag("quantity").over(w))
    .withColumn("rolling_sum_3", F.sum("quantity").over(w.rowsBetween(-2, 0)))
    .withColumn("rank_desc", F.row_number().over(w.orderBy(F.desc("quantity"))))
)
```

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.

202. Complex Joins & Skew Handling: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` 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 complex joins & skew handling problem: - Ingest data with proper schema handling. - Apply necessary transformations

(null-safety, casting, deduplication). - Implement the core logic related to Complex Joins & Skew 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

Skew joins cause a few keys to dominate shuffles. We first profile key frequency, then salt hot keys and broadcast small dimension tables where possible. Enabling AQE can also coalesce skewed partitions. We demonstrate a salting approach.

```
from pyspark.sql import functions as F
from pyspark.sql import types as T

key_freq = df.groupBy("order_id").count()
hot = key_freq.filter("count > 1000000").select("order_id").withColumn("is_hot", F.lit(1
))
df2 = df.join(hot, on="order_id", how="left").fillna({"is_hot":0})

salt_mod = 16
df_salted = df2.withColumn("salt", F.when(F.col("is_hot")==1,
    F.rand()*salt_mod).otherwise(F.lit(0)).cast("int"))

dim_salted = dim.crossJoin(
    spark.range(salt_mod).withColumnRenamed("id","salt")
)
joined = df_salted.join(dim_salted, on=[ "order_id", "salt" ], how="left")
```

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.

203. Nested JSON & Semi-structured Data: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `session_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 nested json & semi-structured data problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Nested JSON & Semi-structured 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

For semi-structured inputs, prefer `from_json` with an explicit schema, handle `badRecordsPath`, and use `explode` for arrays. We also guard against nullable subfields and schema drift.

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

schema = T.StructType([
    T.StructField("session_id", T.StringType()),
    T.StructField("created_at", T.TimestampType()),
    T.StructField("payload", T.StructType([
        T.StructField("items", T.ArrayType(T.StructType([
            T.StructField("sku", T.StringType()),
            T.StructField("duration_ms", T.DoubleType())
        ])))
    ]))
])

raw = (spark.read
       .option("multiLine", True)
       .option("badRecordsPath", "/tmp/bad_records")
       .json("/data/events/*.json"))

dfj = raw.select(F.from_json(F.col("value").cast("string"), schema).alias("r")).select("
r.*")
items = dfj.select("session_id", "created_at", F.explode_outer("payload.items").alias("i
t"))
result = items.select("session_id", "created_at", F.col("it.sku").alias("sku"),
    F.col(f"it.duration_ms").alias("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.

204. UDFs vs Pandas UDFs & Vectorization: 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 udfs vs pandas udfs & vectorization problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to UDFs vs Pandas UDFs & Vectorization (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("value", F.col("value").cast("double"))
out = df2.withColumn("z_value", zscore(F.col("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.

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.

208. File-based Incremental Ingestion: Advanced Task on `sessions`

Question

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

Task. Using PySpark, implement a robust solution to solve a file-based incremental ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File-based Incremental Ingestion (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

Track high-watermarks and process only new data; design idempotent upserts.

```
from pyspark.sql import functions as F

prev_max_ts = "2025-01-01T00:00:00"

new_df = (spark.read.parquet("/data/sessions")
          .filter(F.col("ts") > F.to_timestamp(F.lit(prev_max_ts))))
```

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.

209. Delta Lake Optimize/Z-Order (conceptual with PySpark): Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `device_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 delta lake optimize/z-order (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Delta Lake Optimize/Z-Order (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.device_id = s.device_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

210. CDC/Merge into Delta (conceptual with PySpark): Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `device_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 cdc/merge into delta (conceptual with pyspark) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to CDC/Merge into Delta (conceptual with PySpark) (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

Use Delta MERGE for CDC and compaction/z-order for performance (if available).

```
spark.sql("""
MERGE INTO tgt t
USING src s
ON t.device_id = s.device_id
WHEN MATCHED AND s.is_deleted = true THEN DELETE
WHEN MATCHED THEN UPDATE SET *
WHEN NOT MATCHED THEN INSERT *
""")
```

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.

211. Bucketing, Partitioning & Writer Jobs: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_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 bucketing, partitioning & writer jobs problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Bucketing, Partitioning & Writer Jobs (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

General advanced PySpark pattern.

```
pass
```

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.

212. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `customer_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 adaptive query execution (aqe) and shuffle partitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Adaptive Query Execution (AQE) and Shuffle Partitions (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

Enable AQE and tune shuffle partitions for better task balance.

```
spark.conf.set("spark.sql.adaptive.enabled", "true")
spark.conf.set("spark.sql.shuffle.partitions", "200")
```

```
dfj = fact.join(F.broadcast(dim), on="customer_id", how="left")
```

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.

213. Broadcast Joins and Hints: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `customer_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 broadcast joins and hints problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Broadcast Joins and Hints (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

Broadcast small side tables to avoid shuffles.

```
from pyspark.sql import functions as F
joined = fact.hint("broadcast").join(dim, on="customer_id", how="left")
```

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.

214. Skew Join Salting Techniques: Advanced Task on `metrics`

Question

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

Task. Using PySpark, implement a robust solution to solve a skew join salting techniques problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Skew Join Salting Techniques (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

General advanced PySpark pattern.

```
pass
```

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.

215. Aggregations with Complex Grouping Sets: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `device_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 aggregations with complex grouping sets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Aggregations with Complex Grouping Sets (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

Use cube/rollup for multi-level aggregations.

```
from pyspark.sql import functions as F
cube = (df.cube("device_id", "sku").agg(F.sum("score").alias("sum_score")))
```

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.

216. Explode + Window Hybrids: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `customer_id`, `created_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 explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (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

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("customer_id", "created_at", F.explode("items").alias("it"))
w = W.partitionBy("customer_id", "it").orderBy("created_at")
```

```
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

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.

217. Sessionization (clickstreams): Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` 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 sessionization (clickstreams) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Sessionization (clickstreams) (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

Derive sessions from gaps between events.

```
from pyspark.sql import functions as F, Window as W

w = W.partitionBy("order_id").orderBy("updated_at")
df2 = (df
       .withColumn("prev_ts", F.lag("updated_at").over(w))
       .withColumn("gap_sec", F.coalesce(F.col("updated_at").cast("long") - F.col("prev_
ts").cast("long"),
    F.lit(0)))
       .withColumn("new_sess", (F.col("gap_sec") > 1800).cast("int"))
       .withColumn("session_seq", F.sum("new_sess").over(w.rowsBetween(Window.unboundedP
receding, 0)))
      )
```

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.

218. Time-series Gaps & Islands: Advanced Task on `orders`

Question

Scenario. You have a large `orders` dataset with columns like `account_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 time-series gaps & islands problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Time-series Gaps & Islands (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

Identify contiguous ranges (islands) using row-number differences.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("account_id").orderBy("ts")
df2 = df.withColumn("rn", F.row_number().over(w))
df3 = df2.withColumn("grp", F.expr("rn - row_number() over (partition by account_id orde
r by ts)"))
```

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.

219. Surrogate Keys & Deduplication: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` 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 surrogate keys & deduplication problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Surrogate Keys & Deduplication (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

Deduplicate by stable ordering and build surrogate keys via hashes.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("session_id").orderBy(F.desc("ts"))
dedup = (df.withColumn("rn", F.row_number().over(w)).filter("rn = 1").drop("rn"))
with_id = dedup.withColumn("surrogate_id", F.sha2(F.concat_ws("||", *dedup.columns), 256
))
```

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.

220. SCD Type 2 with MERGE logic (Delta/Parquet): Advanced Task on `clicks`

Question

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

Task. Using PySpark, implement a robust solution to solve a scd type 2 with merge logic (delta/parquet) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to SCD Type 2 with MERGE logic (Delta/Parquet) (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

Maintain history via effective_from/to and is_current flags; build updates and closures.

```
# See MERGE example; or implement DataFrame-based SCD2 staging logic.
```

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.

221. Advanced Window: Last non-null forward-fill: 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 advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (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

Forward-fill values using `last(..., ignorenulls=True)`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("customer_id").orderBy("created_at").rowsBetween(Window.unboundedPrece
ding, 0)
ff = df.withColumn("ff_val", F.last("latency_ms", ignorenulls=True).over(w))
```

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.

222. Top-K per Group at Scale: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `device_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 top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (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

Rank items per group and filter to K.

```python
from pyspark.sql import functions as F, Window as W
K = 3
w = W.partitionBy("device_id").orderBy(F.desc("duration_ms"))
topk = df.withColumn("r", F.row_number().over(w)).filter(F.col("r") <= K).drop("r")
```

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.

223. Rolling Distinct Counts (HLL sketch concept): Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `order_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 rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (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

Approximate distinct counts per rolling window with `approx_count_distinct`.

```
from pyspark.sql import functions as F, Window as W
w = W.partitionBy("order_id").orderBy("updated_at").rowsBetween(-10, 0)
roll = df.withColumn("approx_dc", F.approx_count_distinct("score").over(w))
```

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.

224. Cross-file Schema Evolution: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `session_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 cross-file schema evolution problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Cross-file Schema Evolution (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

225. Dynamic File Pruning: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` 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 dynamic file pruning problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Dynamic File Pruning (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

Partition by time and filter by partition columns for pruning.

```
pruned = spark.read.parquet("/out/transactions").filter(F.col("event_time") >= "2025-01-01")
```

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.

226. Data Quality Checks & Expectations: Advanced Task on `logs`

Question

Scenario. You have a large `logs` 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 data quality checks & expectations problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Data Quality Checks & Expectations (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

Build rule-based validations; quarantine failures with reasons.

```
from pyspark.sql import functions as F

rules = [
    ("not_null_key", F.col("session_id").isNotNull()),
```

```
    ("val_non_negative", F.col("amount") >= 0)
]

def apply_rules(df):
    for rule_name, cond in rules:
        df = df.withColumn("rule_" + rule_name, cond)
    return df

scored = apply_rules(df)
bad = scored.filter("NOT (rule_not_null_key AND rule_val_non_negative)").withColumn("rea
son",
    F.lit("dq_failed"))
good = scored.filter("rule_not_null_key AND
    rule_val_non_negative").drop("rule_not_null_key","rule_val_non_negative")
```

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.

227. Unit Testing with pytest & chispa: Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `customer_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 unit testing with pytest & chispa problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Unit Testing with pytest & chispa (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

Use pytest + chispa to assert DataFrame equality; isolate pure transforms.

```
# pip install chispa
from chispa import assert_df_equality

def transform(df):
    return df.filter("amount > 0")

def test_transform(spark):
    input_df = spark.createDataFrame([(1, -1.0), (2, 3.0)], ["id","amount"])
    exp_df = spark.createDataFrame([(2, 3.0)], ["id","amount"])
    assert_df_equality(transform(input_df), exp_df, ignore_column_order=True)
```

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.

228. Performance Debugging with UI & Query Plans: Advanced Task on `sessions`

Question

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

Task. Using PySpark, implement a robust solution to solve a performance debugging with ui & query plans problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Performance Debugging with UI & Query Plans (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

Inspect query plans and the Spark UI; avoid Python UDFs and skew.

```
df_explain = df.select("session_id", "value").groupBy("session_id").agg(F.sum("value"))
print(df_explain._jdf.queryExecution().toString())
```

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.

230. Reusable Jobs & Parameterized Notebooks: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` 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 reusable jobs & parameterized notebooks problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reusable Jobs & Parameterized Notebooks (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

General advanced PySpark pattern.

```
    pass
```

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.

231. DataFrame <-> Spark SQL Interop: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `user_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 dataframe <-> spark sql interop problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to DataFrame <-> Spark SQL Interop (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

Register temp views to use Spark SQL alongside DataFrame API.

```
df.createOrReplaceTempView("v")
sql_df = spark.sql("select user_id, sum(score) as s from v group by user_id")
```

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.

232. Pivot/Unpivot Large Datasets: Advanced Task on `payments`

Question

Scenario. You have a large `payments` dataset with columns like `account_id`, `created_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 pivot/unpivot large datasets problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Pivot/Unpivot Large Datasets (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

Pivot after pre-aggregating to avoid explosion.

```
from pyspark.sql import functions as F
piv = df.groupBy("account_id").pivot("sku").agg(F.sum("quantity"))
```

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.

233. Joins over Ranges (temporal joins): Advanced Task on `transactions`

Question

Scenario. You have a large `transactions` dataset with columns like `customer_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 joins over ranges (temporal joins) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Joins over Ranges (temporal joins) (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

Join facts to dimensions where timestamp falls within validity range.

```
joined = fact.join(dim, (fact["created_at"].between(dim["start"], dim["end"])) &
    (fact["customer_id"]==dim["customer_id"]), "left")
```

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.

234. Windowed UDAFs (via Pandas UDFs): Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `account_id`, `updated_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 windowed udafs (via pandas udfs) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Windowed UDAFs (via Pandas UDFs) (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

Prefer Pandas UDFs for vectorized operations over Python UDFs for performance. Use type hints and avoid heavy per-row Python code. Ensure Arrow is enabled. Demonstrate a scalar Pandas

UDF.

```
from pyspark.sql import functions as F, types as T
import pandas as pd

@F.pandas_udf("double")
def zscore(col: pd.Series) -> pd.Series:
    mu = col.mean()
    sig = col.std(ddof=0) or 1.0
    return (col - mu) / sig

df2 = df.withColumn("latency_ms", F.col("latency_ms").cast("double"))
out = df2.withColumn("z_latency_ms", zscore(F.col("latency_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.

235. Binary Files & Image Ingestion: Advanced Task on `events`

Question

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

Task. Using PySpark, implement a robust solution to solve a binary files & image ingestion problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Binary Files & Image Ingestion (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

Read binary files for feature extraction or ML preprocessing.

```
images = spark.read.format("binaryFile").load("/data/images/*")
```

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.

236. Graph-Style Problems without GraphFrames: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `account_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 graph-style problems without graphframes problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Graph-Style Problems without GraphFrames (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

Approximate graph analytics via SQL/DF ops (degree counts, simple traversals).

```
edges = spark.createDataFrame([("a","b"),("b","c")], ["src","dst"])
deg = (edges.select("src").union(edges.select("dst"))
          .groupBy("src").count())
```

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.

237. MLlib Pipelines with Custom Transformers: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `user_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 mllib pipelines with custom transformers problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to MLlib Pipelines with Custom Transformers (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

Build ML pipelines; ensure proper vectorization and column roles.

```
from pyspark.ml.feature import VectorAssembler
from pyspark.ml.classification import LogisticRegression
from pyspark.ml import Pipeline

va = VectorAssembler(inputCols=["f1","f2","f3"], outputCol="features")
lr = LogisticRegression(featuresCol="features", labelCol="label")
pipe = Pipeline(stages=[va, lr]).fit(train_df)
```

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.

242. File Compaction Job: Advanced Task on `events`

Question

Scenario. You have a large `events` dataset with columns like `order_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 file compaction job problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to File Compaction Job (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/events")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/events_compacted"))
```

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.

243. Small-file Problem Mitigation: Advanced Task on `clicks`

Question

Scenario. You have a large `clicks` dataset with columns like `account_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 small-file problem mitigation problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Small-file Problem Mitigation (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

Coalesce many small files into fewer large ones to improve read performance.

```
(spark.read.parquet("/bronze/clicks")
      .repartition(64)
      .write.mode("overwrite").parquet("/silver/clicks_compacted"))
```

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.

244. Reading from Hive Metastore & External Tables: Advanced Task on `payments`

Question

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

Task. Using PySpark, implement a robust solution to solve a reading from hive metastore & external tables problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading from Hive Metastore & External Tables (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

Integrate with Hive catalog; repair partitions; manage external tables.

```
spark.sql("MSCK REPAIR TABLE db.tbl")
```

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.

245. Security & PII Masking Patterns: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `user_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 security & pii masking patterns problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Security & PII Masking Patterns (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

Mask/obfuscate sensitive columns; restrict access via views/catalog controls.

```
from pyspark.sql import functions as F
masked = df.withColumn("email_masked", F.sha2(F.col("email"), 256))
```

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.

246. Column-level Encryption (conceptual + UDF demo): Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `session_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 column-level encryption (conceptual + udf demo) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Column-level Encryption (conceptual + UDF demo) (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

Demo-only: emulate encryption via hashing; real systems should use KMS.

```
from pyspark.sql import functions as F
KEY = F.lit("demo-key")
```

```
enc = df.withColumn("enc", F.sha2(F.concat_ws(":", "session_id", KEY), 256))
```

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.

247. Debugging Serialization / Pickling issues: Advanced Task on `impressions`

Question

Scenario. You have a large `impressions` dataset with columns like `device_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 debugging serialization / pickling issues problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Debugging Serialization / Pickling issues (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

Avoid shipping large objects to executors; use broadcast variables.

```
bc = spark.sparkContext.broadcast({"a":1,"b":2})
```

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.

248. Handling Very Wide Schemas: Advanced Task on `metrics`

Question

Scenario. You have a large `metrics` dataset with columns like `customer_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 handling very wide schemas problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Handling Very Wide Schemas (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

Enable mergeSchema and align columns across writes.

```
df.write.option("mergeSchema","true").mode("append").parquet("/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.

249. Reading Multi-line JSON & Corrupt Records: Advanced Task on `metrics`

Question

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

Task. Using PySpark, implement a robust solution to solve a reading multi-line json & corrupt records problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Reading Multi-line JSON & Corrupt Records (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

Enable multiLine mode and capture corrupt records for analysis.

```
df = (spark.read.option("multiLine", True)
                .option("mode","PERMISSIVE")
                .json("/data/mljson"))
```

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.

250. Optimizing fromRDD / mapPartitions: Advanced Task on `sessions`

Question

Scenario. You have a large `sessions` dataset with columns like `device_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 optimizing fromrdd / mappartitions problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Optimizing fromRDD / mapPartitions (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

Use mapPartitions to amortize per-connection overhead for external I/O.

```
rdd = df.rdd.mapPartitions(lambda it: (x for x in it))
df2 = spark.createDataFrame(rdd, df.schema)
```

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.