Advanced PySpark — Performance

Included questions:

2. Complex Joins & Skew Handling: Advanced Task on `orders`
11. Bucketing, Partitioning & Writer Jobs: Advanced Task on `logs`
12. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `impressions`
13. Broadcast Joins and Hints: Advanced Task on `transactions`
14. Skew Join Salting Techniques: Advanced Task on `logs`
25. Dynamic File Pruning: Advanced Task on `logs`
28. Performance Debugging with UI & Query Plans: Advanced Task on `payments`
29. Caching vs Checkpointing vs Persist: Advanced Task on `orders`
42. File Compaction Job: Advanced Task on `logs`
43. Small-file Problem Mitigation: Advanced Task on `orders`
52. Complex Joins & Skew Handling: Advanced Task on `transactions`
61. Bucketing, Partitioning & Writer Jobs: Advanced Task on `logs`
62. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `orders`
63. Broadcast Joins and Hints: Advanced Task on `payments`
64. Skew Join Salting Techniques: Advanced Task on `orders`
75. Dynamic File Pruning: Advanced Task on `transactions`
78. Performance Debugging with UI & Query Plans: Advanced Task on `events`
79. Caching vs Checkpointing vs Persist: Advanced Task on `metrics`
92. File Compaction Job: Advanced Task on `impressions`
93. Small-file Problem Mitigation: Advanced Task on `payments`
102. Complex Joins & Skew Handling: Advanced Task on `clicks`
111. Bucketing, Partitioning & Writer Jobs: Advanced Task on `metrics`
112. Adaptive Query Execution (AQE) and Shuffle Partitions: Advanced Task on `sessions`
113. Broadcast Joins and Hints: Advanced Task on `transactions`
114. Skew Join Salting Techniques: Advanced Task on `logs`
125. Dynamic File Pruning: Advanced Task on `orders`
128. Performance Debugging with UI & Query Plans: Advanced Task on `transactions`
129. Caching vs Checkpointing vs Persist: Advanced Task on `orders`
142. File Compaction Job: Advanced Task on `orders`
143. Small-file Problem Mitigation: Advanced Task on `sessions`
152. Complex Joins & Skew Handling: Advanced Task on `metrics`

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

242. File Compaction Job: Advanced Task on `events`

243. Small-file Problem Mitigation: Advanced Task on `clicks`

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.

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.

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.

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.

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.

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.

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.

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.

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("dur
ation_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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.