Advanced PySpark — Delta_Cdc

Included questions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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.

```
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.

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.

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.

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.

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.

```
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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.

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.