Advanced PySpark — Windows

Included questions:

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

221. Advanced Window: Last non-null forward-fill: Advanced Task on `events`

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

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

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

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

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("customer_id", "event_time", F.explode("items").alias("it"))
w = W.partitionBy("customer_id", "it").orderBy("event_time")
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Rank items per group and filter to K.

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Approximate distinct counts per rolling window with `approx_count_distinct`.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

Join facts to dimensions where timestamp falls within validity range.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

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

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("customer_id", "updated_at", F.explode("items").alias("it"))
w = W.partitionBy("customer_id", "it").orderBy("updated_at")
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Rank items per group and filter to K.

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Approximate distinct counts per rolling window with `approx_count_distinct`.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

Join facts to dimensions where timestamp falls within validity range.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

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

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("user_id", "ts", F.explode("items").alias("it"))
w = W.partitionBy("user_id", "it").orderBy("ts")
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Rank items per group and filter to K.

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Approximate distinct counts per rolling window with `approx_count_distinct`.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

Join facts to dimensions where timestamp falls within validity range.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

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

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("device_id", "event_time", F.explode("items").alias("it"))
w = W.partitionBy("device_id", "it").orderBy("event_time")
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Rank items per group and filter to K.

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Approximate distinct counts per rolling window with `approx_count_distinct`.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

Join facts to dimensions where timestamp falls within validity range.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

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

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a explode + window hybrids problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Explode + Window Hybrids (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Explode arrays then compute windowed metrics.

```
from pyspark.sql import functions as F, Window as W
expl = df.select("customer_id", "created_at", F.explode("items").alias("it"))
w = W.partitionBy("customer_id", "it").orderBy("created_at")
result = expl.withColumn("cnt", F.count("*").over(w.rowsBetween(-10, 0)))
```

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

221. Advanced Window: Last non-null forward-fill: Advanced Task on `events`

Question

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

Task. Using PySpark, implement a robust solution to solve a advanced window: last non-null forward-fill problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Advanced Window: Last non-null forward-fill (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

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

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a top-k per group at scale problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Top-K per Group at Scale (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Rank items per group and filter to K.

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

Validation

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

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

Question

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

Task. Using PySpark, implement a robust solution to solve a rolling distinct counts (hll sketch concept) problem: - Ingest data with proper schema handling. - Apply necessary transformations (null-safety, casting, deduplication). - Implement the core logic related to Rolling Distinct Counts (HLL sketch concept) (detailed below). - Produce an optimized output suitable for downstream consumption (partitioning/bucketing where applicable).

Why this is hard

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

Solution Outline & Explanation

Approximate distinct counts per rolling window with `approx_count_distinct`.

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

Validation

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

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

Question

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

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

Why this is hard

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

Solution Outline & Explanation

Join facts to dimensions where timestamp falls within validity range.

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

Validation

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