Solution

1) Sessionization (30-min gaps)

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

gap = F.when(F.col("ts") -
F.lag("ts").over(W.partitionBy("user_id").orderBy("ts")) >
F.expr("INTERVAL 30 MINUTES"), 1).otherwise(0)
df1 = (df.orderBy("user_id","ts")
         .withColumn("new_sess", F.coalesce(gap, F.lit(1)))
         .withColumn("session_id",
F.sum("new_sess").over(W.partitionBy("user_id").orderBy("ts")
                                  .rowsBetween(W.unboundedPreceding,
0))))
out = (df1.groupBy("user_id","session_id")
          .agg(F.min("ts").alias("session_start"),
               F.max("ts").alias("session_end"),
               F.count("*").alias("page_count")))
```

---

2) Top-N per group with ties

```
w = W.partitionBy("cust_id").orderBy(F.col("amount").desc())
df_top = (sales
  .withColumn("rnk", F.dense_rank().over(w))
  .filter(F.col("rnk") <= 3))
```

---

3) SCD Type-2 (Delta, upserts at scale)

```
from delta.tables import DeltaTable
from pyspark.sql import functions as F

src = source.withColumn("effective_from",
F.col("ingest_ts")).withColumn("effective_to",
F.lit(None).cast("timestamp")).withColumn("is_current", F.lit(True))
```

```
tgt = DeltaTable.forName(spark, "dim_customers")

# Close rows that changed
updates = (src.alias("s").join(tgt.toDF().alias("t"), "customer_id")
  .where(F.sha2(F.to_json("s"),256) != F.sha2(F.to_json("t"),256))  #
fast change detect on business cols
  .select("t.customer_id"))

tgt.alias("t").merge(
  src.alias("s"), "t.customer_id = s.customer_id AND t.is_current =
true"
).whenMatchedUpdate(
  condition="sha2(to_json(s.*),256) <> sha2(to_json(t.*),256)",
  set={"effective_to":"s.effective_from","is_current":"false"}
).whenNotMatchedInsertAll().execute()
```

Tip: Use a **hash of business columns** to avoid expensive row-by-row comparisons.

---

4) Incremental de-dup (exactly-once)

Batch:

```
# keep only first-seen tx_id across all batches
existing = spark.read.parquet(target_path).select("tx_id").distinct()
new_uniques = incoming.join(existing, "tx_id", "left_anti")
(new_uniques.write.mode("append").parquet(target_path))
```

Streaming:

```
(sdf
 .withWatermark("ts","2 hours")
 .dropDuplicates(["tx_id"])              # stateful, exactly-once
semantics per tx_id+watermark
 .writeStream.format("delta").option("checkpointLocation",
ckpt).start(target_tbl))
```

---

5) Skew-safe join

Broadcast when safe (<= ~10–50MB):

```
small = F.broadcast(ads.select("ad_id","attr"))
joined = clicks.join(small, "ad_id", "left")
```

Salting for hot keys:

```
from pyspark.sql.functions import rand, floor
salt_n = 20
clicks_s = clicks.withColumn("salt", floor(rand()*salt_n))
ads_s = (ads
  .withColumn("salt", F.explode(F.sequence(F.lit(0),
F.lit(salt_n-1)))))
joined = clicks_s.join(ads_s, ["ad_id","salt"])
```

Also enable **AQE**: `spark.conf.set("spark.sql.adaptive.enabled","true")`.

---

6) Explode nested JSON (schema-safe)

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

def flatten(df):
    complex_fields = [(f.name, f.dataType) for f in df.schema.fields
                      if isinstance(f.dataType, (T.ArrayType,
T.StructType))]
    while complex_fields:
        name, dtype = complex_fields.pop(0)
        if isinstance(dtype, T.StructType):
            expanded = [F.col(f"{name}.{k}").alias(f"{name}_{k}") for
k in [n.name for n in dtype]]
            df = df.select("*", *expanded).drop(name)
        elif isinstance(dtype, T.ArrayType) and
isinstance(dtype.elementType, T.StructType):
            df = df.withColumn(name, F.explode_outer(name))
        complex_fields = [(f.name, f.dataType) for f in
df.schema.fields
```

```
                        if isinstance(f.dataType, (T.ArrayType,
T.StructType))]
    return df

# enforce provided schema to avoid _corrupt_record
df = spark.read.schema(provided_schema).json(input_path)
flat = flatten(df)
```

---

7) Event-time aggregation with late data

```
(sdf
 .withWatermark("event_time","2 hours")
 .groupBy(F.window("event_time","1 hour").alias("win"))
 .agg(F.countDistinct("user_id").alias("distinct_users"))
 .select("win.start","win.end","distinct_users"))
```

---

8) Rolling 1-hour avg/stddev with reset (>15-min gap)

```
w_order = W.partitionBy("device_id").orderBy("ts")
gap = (F.col("ts") - F.lag("ts").over(w_order)) > F.expr("INTERVAL 15
MINUTES")
df2 = (df.withColumn("reset", F.when(gap, 1).otherwise(0))
         .withColumn("grp",
F.sum("reset").over(w_order.rowsBetween(W.unboundedPreceding, 0))))

w_roll = (W.partitionBy("device_id","grp")
            .orderBy(F.col("ts").cast("long"))
            .rangeBetween(-3600, 0))  # 1 hour in seconds (ts cast to
long)

out = df2.select(
    "device_id","ts","reading",
    F.avg("reading").over(w_roll).alias("avg_1h"),
    F.stddev_samp("reading").over(w_roll).alias("std_1h")
)
```

---

9) File compaction (target ~128MB)

```
spark.conf.set("spark.sql.files.maxRecordsPerFile", 0)  # let size
drive splits
# Estimate target partitions by total size / 128MB
input_df = spark.read.parquet(src_path)
target_parts = max(1, int(input_df.rdd.getNumPartitions() / 4))  #
quick heuristic
(input_df
   .repartition("dt")            # keep partitioning
   .sortWithinPartitions("dt")   # better packing
   .write.mode("overwrite").partitionBy("dt").parquet(dst_path))
```

On Delta: prefer `OPTIMIZE table ZORDER BY (...)` for ongoing compaction.

---

10) CDC merge with deletes (Delta)

```
from delta.tables import DeltaTable
cdc = cdc_df.select("id","op","op_ts", "cols...")  # op in
('I','U','D')

tgt = DeltaTable.forName(spark, "sales")
tgt.alias("t").merge(
  cdc.alias("c"), "t.id = c.id"
).whenMatchedUpdate(
  condition="c.op = 'U' AND c.op_ts >= t.op_ts",
  set={"col1":"c.col1","op_ts":"c.op_ts"}
).whenMatchedDelete(condition="c.op = 'D' AND c.op_ts >= t.op_ts"
).whenNotMatchedInsert(
  condition="c.op = 'I'",
  values={"id":"c.id","col1":"c.col1","op_ts":"c.op_ts"}
).execute()
```

Deduplicate CDC first on `(id, op_ts)` with `row_number` to keep the latest per id.

---

11) Consecutive daily streaks

```
from pyspark.sql import functions as F, Window as W
d = (logins.select("user_id",
F.to_date("login_date").alias("d")).distinct())
w = W.partitionBy("user_id").orderBy("d")
grp = (F.datediff("d", F.lag("d").over(w)).alias("diff"))
g = (d.withColumn("diff", grp)
      .withColumn("grp",
          F.sum(F.when((F.col("diff") != 1) | F.col("diff").isNull(),
1).otherwise(0))
            .over(w.rowsBetween(W.unboundedPreceding, 0))))
streaks = (g.groupBy("user_id","grp")
            .agg(F.min("d").alias("start"), F.max("d").alias("end"),
                 F.count("*").alias("len")))
result = (streaks.withColumn("r",

F.row_number().over(W.partitionBy("user_id").orderBy(F.col("len").desc
(),F.col("start"))))
          .filter("r=1").drop("r"))
```