```
#Section 1
# 1) Ingestion

from pyspark.sql import SparkSession

spark = (SparkSession.builder
         .appName("Orders-Ingestion")
         .config("spark.sql.session.timeZone", "UTC")
         .getOrCreate())

input_path = "orders_raw.csv"  # change to your path if needed

df_raw = (spark.read
          .option("header", True)        # first row is header
          .option("inferSchema", False)  # read everything as strings
          .option("mode", "PERMISSIVE")  # do not fail on malformed rows
          .csv(input_path))

# Quick peek (optional)
df_raw.show(5, truncate=False)
```

```
+-----------+-----------+-----------+-----------+-----------+-------+----------+---------+
|order_id   |customer_id|city       |category   |product    |amount |order_date|status   |
+-----------+-----------+-----------+-----------+-----------+-------+----------+---------+
|ORD00000000|C000000    | hyderabad | grocery   |Oil        |invalid|01/01/2024|Cancelled|
|ORD00000001|C000001    |Pune       |Grocery    |Sugar      |35430  |2024-01-02|Completed|
|ORD00000002|C000002    |Pune       |Electronics|Mobile     |65358  |2024-01-03|Completed|
|ORD00000003|C000003    |Bangalore  |Electronics|Laptop     |5558   |2024-01-04|Completed|
|ORD00000004|C000004    |Pune       |Home       |AirPurifier|33659  |2024-01-05|Completed|
+-----------+-----------+-----------+-----------+-----------+-------+----------+---------+
only showing top 5 rows
```

```
# 3) Print schema & total record count

df_raw.printSchema()

total_count = df_raw.count()
print(f"Total records (raw): {total_count}")
```

```
root
 |-- order_id: string (nullable = true)
 |-- customer_id: string (nullable = true)
 |-- city: string (nullable = true)
 |-- category: string (nullable = true)
 |-- product: string (nullable = true)
 |-- amount: string (nullable = true)
 |-- order_date: string (nullable = true)
 |-- status: string (nullable = true)

Total records (raw): 300000
```

```
# 2.1) Trim spaces on all columns (safe for strings)
from pyspark.sql.functions import col, trim

df_trimmed = df_raw.select([trim(col(c)).alias(c) for c in df_raw.columns])

# Quick check
df_trimmed.show(5, truncate=False)
```

```
+-----------+-----------+---------+-----------+-----------+-------+----------+---------+
|order_id   |customer_id|city     |category   |product    |amount |order_date|status   |
+-----------+-----------+---------+-----------+-----------+-------+----------+---------+
|ORD00000000|C000000    |hyderabad|grocery    |Oil        |invalid|01/01/2024|Cancelled|
|ORD00000001|C000001    |Pune     |Grocery    |Sugar      |35430  |2024-01-02|Completed|
|ORD00000002|C000002    |Pune     |Electronics|Mobile     |65358  |2024-01-03|Completed|
|ORD00000003|C000003    |Bangalore|Electronics|Laptop     |5558   |2024-01-04|Completed|
|ORD00000004|C000004    |Pune     |Home       |AirPurifier|33659  |2024-01-05|Completed|
+-----------+-----------+---------+-----------+-----------+-------+----------+---------+
only showing top 5 rows
```

```
#4
from pyspark.sql.functions import col, trim

df_trimmed = df_raw.select([trim(col(c)).alias(c) for c in df_raw.columns])
```

```
# 5
from pyspark.sql.functions import lower, initcap

df_std = (df_trimmed
          .withColumn("city_std", initcap(lower(col("city"))))
          .withColumn("category_std", lower(col("category")))
          .withColumn("product_std", lower(col("product"))))


df_std.select("city", "city_std", "category", "category_std", "product", "product_std").show(10, truncate=False)
```

```
+---------+---------+-----------+------------+-----------+-----------+
|city     |city_std |category   |category_std|product    |product_std|
+---------+---------+-----------+------------+-----------+-----------+
|hyderabad|Hyderabad|grocery    |grocery     |Oil        |oil        |
|Pune     |Pune     |Grocery    |grocery     |Sugar      |sugar      |
|Pune     |Pune     |Electronics|electronics |Mobile     |mobile     |
|Bangalore|Bangalore|Electronics|electronics |Laptop     |laptop     |
|Pune     |Pune     |Home       |home        |AirPurifier|airpurifier|
|Delhi    |Delhi    |Fashion    |fashion     |Jeans      |jeans      |
|Delhi    |Delhi    |Grocery    |grocery     |Sugar      |sugar      |
|Pune     |Pune     |Grocery    |grocery     |Rice       |rice       |
|Bangalore|Bangalore|Fashion    |fashion     |Jeans      |jeans      |
|Kolkata  |Kolkata  |Electronics|electronics |Laptop     |laptop     |
+---------+---------+-----------+------------+-----------+-----------+
only showing top 10 rows
```

```
#6
from pyspark.sql.functions import regexp_replace, when

df_amt = (df_std
          .withColumn("amount_clean", regexp_replace(col("amount"), r"[^0-9]", ""))

          .withColumn("amount_int", when(col("amount_clean") == "", None).otherwise(col("amount_clean").cast("int"))))

df_amt.select("amount", "amount_clean", "amount_int").show(15, truncate=False)
```

```
+-------+------------+----------+
|amount |amount_clean|amount_int|
+-------+------------+----------+
|invalid|            |NULL      |
|35430  |35430       |35430     |
|65358  |65358       |65358     |
|5558   |5558        |5558      |
|33659  |33659       |33659     |
|8521   |8521        |8521      |
|42383  |42383       |42383     |
|45362  |45362       |45362     |
|10563  |10563       |10563     |
|63715  |63715       |63715     |
|66576  |66576       |66576     |
|50318  |50318       |50318     |
|84768  |84768       |84768     |
|79121  |79121       |79121     |
|79469  |79469       |79469     |
+-------+------------+----------+
only showing top 15 rows
```

```
# 7
from pyspark.sql.functions import to_date, coalesce

df_dates = df_amt.withColumn(
    "order_date_parsed",
    coalesce(
```

```
        to_date(col("order_date"), "yyyy-MM-dd"),
        to_date(col("order_date"), "dd/MM/yyyy"),
        to_date(col("order_date"), "yyyy/MM/dd"),
        to_date(col("order_date"), "dd-MM-yyyy")  # safety if any show up
    )
)

df_dates.select("order_date", "order_date_parsed").show(20, truncate=False)
```

```
---------------------------------------------------------------------------
DateTimeException                         Traceback (most recent call last)
/tmp/ipython-input-2038369039.py in <cell line: 0>()
     12 )
     13
---> 14 df_dates.select("order_date", "order_date_parsed").show(20, truncate=False)

 ──────────────────────────────── 3 frames ────────────────────────────────
/usr/local/lib/python3.12/dist-packages/pyspark/sql/classic/dataframe.py in show(self, n, truncate, vertical)
    283
    284     def show(self, n: int = 20, truncate: Union[bool, int] = True, vertical: bool = False) -> None:
--> 285         print(self._show_string(n, truncate, vertical))
    286
    287     def _show_string(

/usr/local/lib/python3.12/dist-packages/pyspark/sql/classic/dataframe.py in _show_string(self, n, truncate, vertical)
    314                 )
    315
--> 316             return self._jdf.showString(n, int_truncate, vertical)
    317
    318     def __repr__(self) -> str:

/usr/local/lib/python3.12/dist-packages/py4j/java_gateway.py in __call__(self, *args)
   1360
   1361         answer = self.gateway_client.send_command(command)
-> 1362         return_value = get_return_value(
   1363             answer, self.gateway_client, self.target_id, self.name)
   1364

/usr/local/lib/python3.12/dist-packages/pyspark/errors/exceptions/captured.py in deco(*a, **kw)
    286                 # Hide where the exception came from that shows a non-Pythonic
    287                 # JVM exception message.
--> 288                 raise converted from None
    289             else:
    290                 raise

DateTimeException: [CANNOT_PARSE_TIMESTAMP] Text '01/01/2024' could not be parsed at index 0. Use `try_to_timestamp` to
tolerate invalid input string and return NULL instead. SQLSTATE: 22007
```

Next steps: Explain error

```
#8
from pyspark.sql.functions import lower, array, array_union, lit, size

df_dq = (df_dates
         .withColumn("status_std", lower(trim(col("status"))))
         .withColumn("issues", array())  # start empty issues array
         .withColumn("issues",
                     when(col("order_id").isNull() | (trim(col("order_id")) == ""),
                          array_union(col("issues"), array(lit("missing_order_id"))))
                     .otherwise(col("issues")))
         .withColumn("issues",
                     when(col("amount_int").isNull(),
                          array_union(col("issues"), array(lit("invalid_amount"))))
                     .otherwise(col("issues")))
         .withColumn("issues",
                     when(col("order_date_parsed").isNull(),
                          array_union(col("issues"), array(lit("invalid_order_date"))))
                     .otherwise(col("issues")))
         .withColumn("issues_count", size(col("issues")))
)

# Split datasets
curated_df = df_dq.filter(col("issues_count") == 0)
invalid_df = df_dq.filter(col("issues_count") > 0)
```

```
print(f"Curated (clean) records: {curated_df.count()}")
print(f"Quarantined (invalid) records: {invalid_df.count()}")

# Optional: inspect a few invalid rows to understand common problems
invalid_df.select("order_id", "amount", "amount_int", "order_date", "order_date_parsed", "issues").show(20, truncate=False)
```

```
---------------------------------------------------------------------------
DateTimeException                         Traceback (most recent call last)
/tmp/ipython-input-2400963030.py in <cell line: 0>()
     24 invalid_df = df_dq.filter(col("issues_count") > 0)
     25
---> 26 print(f"Curated (clean) records: {curated_df.count()}")
     27 print(f"Quarantined (invalid) records: {invalid_df.count()}")
     28

 2 frames
/usr/local/lib/python3.12/dist-packages/pyspark/errors/exceptions/captured.py in deco(*a, **kw)
    286                 # Hide where the exception came from that shows a non-Pythonic
    287                 # JVM exception message.
--> 288                 raise converted from None
    289             else:
    290                 raise

DateTimeException: [CANNOT_PARSE_TIMESTAMP] Text '01/01/2024' could not be parsed at index 0. Use `try_to_timestamp` to
tolerate invalid input string and return NULL instead. SQLSTATE: 22007
```

Next steps: Explain error

```
from pyspark.sql.functions import to_date, coalesce, col, trim

df_tmp = df_raw.withColumn("order_date", trim(col("order_date")))

formats = [
    "yyyy-MM-dd",      # 2024-12-31
    "MM/dd/yyyy",      # 12/31/2024
    "dd/MM/yyyy",      # 31/12/2024
    "dd-MMM-yyyy",     # 31-Dec-2024
    "yyyy/MM/dd",      # 2024/12/31
]

parsed_exprs = [to_date(col("order_date"), f) for f in formats]

df_parsed = df_tmp.withColumn("order_date_parsed", coalesce(*parsed_exprs))

df_parsed.select("order_id", "order_date", "order_date_parsed").show(10, truncate=False)
df_parsed.printSchema()
```

```
---------------------------------------------------------------------------
DateTimeException                         Traceback (most recent call last)
/tmp/ipython-input-3415670636.py in <cell line: 0>()
     15 df_parsed = df_tmp.withColumn("order_date_parsed", coalesce(*parsed_exprs))
     16
---> 17 df_parsed.select("order_id", "order_date", "order_date_parsed").show(10, truncate=False)
     18 df_parsed.printSchema()

 3 frames
/usr/local/lib/python3.12/dist-packages/pyspark/errors/exceptions/captured.py in deco(*a, **kw)
    286                 # Hide where the exception came from that shows a non-Pythonic
    287                 # JVM exception message.
--> 288                 raise converted from None
    289             else:
    290                 raise

DateTimeException: [CANNOT_PARSE_TIMESTAMP] Text '01/01/2024' could not be parsed at index 0. Use `try_to_timestamp` to
tolerate invalid input string and return NULL instead. SQLSTATE: 22007
```

Next steps: Explain error

```
#9

df_dedup = df_parsed.dropDuplicates(["order_id"])
```

```
count_after_dedup = df_dedup.count()
print(f"Total records AFTER dedup on order_id: {count_after_dedup}")
```

```
Total records AFTER dedup on order_id: 300000
```

```
#10
df_completed = df_dedup.filter(lower(trim(col("status"))) == "completed")
count_after_filter = df_completed.count()
print(f"Total records AFTER filtering status=Completed: {count_after_filter}")
```

```
Total records AFTER filtering status=Completed: 285000
```

```
#11
count_before = df_parsed.count()
print(f"Total records BEFORE dedup/filter: {count_before}")
```

```
Total records BEFORE dedup/filter: 300000
```

```
#13


from pyspark.sql.functions import col, countDistinct

plan_df = (
    df_parsed
      .dropDuplicates(["order_id"])
      .filter(col("status").isNotNull())
      .groupBy("city")
      .agg(countDistinct("order_id").alias("orders"))
)


plan_df.explain(True)
```

```
== Parsed Logical Plan ==
'Aggregate ['city], ['city, 'count(distinct 'order_id) AS orders#451]
+- Filter isnotnull(status#132)
   +- Deduplicate [order_id#125]
      +- Project [order_id#125, customer_id#126, city#127, category#128, product#129, amount#130, order_date#351, status#132, co
         +- Project [order_id#125, customer_id#126, city#127, category#128, product#129, amount#130, trim(order_date#131, None)
            +- Relation [order_id#125,customer_id#126,city#127,category#128,product#129,amount#130,order_date#131,status#132] cs

== Analyzed Logical Plan ==
city: string, orders: bigint
Aggregate [city#127], [city#127, count(distinct order_id#125) AS orders#451L]
+- Filter isnotnull(status#132)
   +- Deduplicate [order_id#125]
      +- Project [order_id#125, customer_id#126, city#127, category#128, product#129, amount#130, order_date#351, status#132, co
         +- Project [order_id#125, customer_id#126, city#127, category#128, product#129, amount#130, trim(order_date#131, None)
            +- Relation [order_id#125,customer_id#126,city#127,category#128,product#129,amount#130,order_date#131,status#132] cs

== Optimized Logical Plan ==
Aggregate [city#465], [city#465, count(distinct order_id#125) AS orders#451L]
+- Project [order_id#125, city#465]
   +- Filter isnotnull(status#475)
      +- Aggregate [order_id#125], [order_id#125, first(city#127, false) AS city#465, first(status#132, false) AS status#475]
         +- Project [order_id#125, city#127, status#132]
            +- Relation [order_id#125,customer_id#126,city#127,category#128,product#129,amount#130,order_date#131,status#132] cs

== Physical Plan ==
AdaptiveSparkPlan isFinalPlan=false
+- HashAggregate(keys=[city#465], functions=[count(distinct order_id#125)], output=[city#465, orders#451L])
   +- Exchange hashpartitioning(city#465, 200), ENSURE_REQUIREMENTS, [plan_id=642]
      +- HashAggregate(keys=[city#465], functions=[partial_count(distinct order_id#125)], output=[city#465, count#480L])
         +- HashAggregate(keys=[city#465, order_id#125], functions=[], output=[city#465, order_id#125])
            +- HashAggregate(keys=[city#465, order_id#125], functions=[], output=[city#465, order_id#125])
               +- Project [order_id#125, city#465]
                  +- Filter isnotnull(status#475)
                     +- SortAggregate(key=[order_id#125], functions=[first(city#127, false), first(status#132, false)], output=[
                        +- Sort [order_id#125 ASC NULLS FIRST], false, 0
                           +- Exchange hashpartitioning(order_id#125, 200), ENSURE_REQUIREMENTS, [plan_id=633]
                              +- SortAggregate(key=[order_id#125], functions=[partial_first(city#127, false), partial_first(stat
                                 +- Sort [order_id#125 ASC NULLS FIRST], false, 0
                                    +- FileScan csv [order_id#125,city#127,status#132] Batched: false, DataFilters: [], Format:
```

```
from pyspark.sql.functions import sum as _sum, avg as _avg, desc
```

```
# Repartition by city for city-level aggregation (co-locate keys)
df_city_part = curated_completed.repartition(256, col("city_std")).cache()
df_city_part.count()  # materialize cache

revenue_by_city = (df_city_part
                   .groupBy("city_std")
                   .agg(_sum("amount_int").alias("total_revenue"),
                        _avg("amount_int").alias("avg_order_value"))
                   .orderBy(desc("total_revenue")))

revenue_by_city.explain(True)
revenue_by_city.show(50, truncate=False)
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
/tmp/ipython-input-3003942810.py in <cell line: 0>()
      2
      3 # Repartition by city for city-level aggregation (co-locate keys)
----> 4 df_city_part = curated_completed.repartition(256, col("city_std")).cache()
      5 df_city_part.count()  # materialize cache
      6

NameError: name 'curated_completed' is not defined
```

Next steps: Explain error

```
#Section 5 16
from pyspark.sql.functions import sum as _sum, desc

revenue_by_city = (curated_completed
                   .groupBy("city_std")
                   .agg(_sum("amount_int").alias("total_revenue"))
                   .orderBy(desc("total_revenue")))

revenue_by_city.show(50, truncate=False)
```

```
#17

revenue_by_category = (curated_completed
                       .groupBy("category_std")
                       .agg(_sum("amount_int").alias("total_revenue"))
                       .orderBy(desc("total_revenue")))

revenue_by_category.show(50, truncate=False)
```

```
#18
top_5_products = (curated_completed
                  .groupBy("product_std")
                  .agg(_sum("amount_int").alias("total_revenue"))
                  .orderBy(desc("total_revenue"))
                  .limit(5))

top_5_products.show(truncate=False)
```

```
#19
from pyspark.sql.functions import avg as _avg

aov_by_city = (curated_completed
               .groupBy("city_std")
               .agg(_avg("amount_int").alias("avg_order_value"))
               .orderBy(desc("avg_order_value")))

aov_by_city.show(50, truncate=False)
```

```
#Section 6 20

from pyspark.sql.functions import sum as _sum, desc
```

```
from pyspark.sql.window import Window
from pyspark.sql.functions import row_number

city_rev = (curated_completed
            .groupBy("city_std")
            .agg(_sum("amount_int").alias("total_revenue")))

w_city = Window.orderBy(desc("total_revenue"))
city_ranked = city_rev.withColumn("revenue_rank", row_number().over(w_city))

city_ranked.show(50, truncate=False)
```

```
#21

from pyspark.sql.window import Window
from pyspark.sql.functions import row_number

cat_prod_rev = (curated_completed
                .groupBy("category_std", "product_std")
                .agg(_sum("amount_int").alias("total_revenue")))

w_cat_prod = Window.partitionBy("category_std").orderBy(desc("total_revenue"))
cat_prod_ranked = cat_prod_rev.withColumn("rank_in_category", row_number().over(w_cat_prod))

cat_prod_ranked.show(50, truncate=False)
```

```
#22

top_product_per_category = cat_prod_ranked.filter("rank_in_category = 1")
top_product_per_category.show(100, truncate=False)
```

```
# section 7 23
clean_out = "out/cleaned_orders_parquet"  # adapt to HDFS/ADLS/S3 as needed
(curated_completed
 .select(
     "order_id", "customer_id",
     "city_std", "category_std", "product_std",
     "amount_int", "order_date_parsed"  # typed, cleaned fields
 )
 .write
 .mode("overwrite")
 .partitionBy("city_std")
 .parquet(clean_out))
```

```
#24
analytics_out = "out/analytics_orc"
from pyspark.sql.functions import sum as _sum, avg as _avg, desc

revenue_by_city = (curated_completed
                   .groupBy("city_std")
                   .agg(_sum("amount_int").alias("total_revenue"))
                   .orderBy(desc("total_revenue")))

revenue_by_category = (curated_completed
                       .groupBy("category_std")
                       .agg(_sum("amount_int").alias("total_revenue"))
                       .orderBy(desc("total_revenue")))

top_5_products = (curated_completed
                  .groupBy("product_std")
                  .agg(_sum("amount_int").alias("total_revenue"))
                  .orderBy(desc("total_revenue"))
                  .limit(5))

aov_by_city = (curated_completed
               .groupBy("city_std")
               .agg(_avg("amount_int").alias("avg_order_value"))
               .orderBy(desc("avg_order_value")))

(revenue_by_city.write.mode("overwrite").format("orc").save(f"{analytics_out}/revenue_by_city"))
(revenue_by_category.write.mode("overwrite").format("orc").save(f"{analytics_out}/revenue_by_category"))
```

```
(top_5_products.write.mode("overwrite").format("orc").save(f"{analytics_out}/top_5_products"))
(aov_by_city.write.mode("overwrite").format("orc").save(f"{analytics_out}/aov_by_city"))
```