1️⃣ Top 3 products by total sales per region in the last 6 months (handle ties)

Approach:

- Filter sales in the last 6 months
- Aggregate by `region, product_id`
- Use `DENSE_RANK()` to handle ties
- Pick rank ≤ 3

```
WITH sales_6m AS (
    SELECT region,
           product_id,
           SUM(sales_amount) AS total_sales
    FROM sales
    WHERE sale_date >= DATEADD(MONTH, -6, CURRENT_DATE)
    GROUP BY region, product_id
),
ranked AS (
    SELECT region,
           product_id,
           total_sales,
           DENSE_RANK() OVER (
                PARTITION BY region
                ORDER BY total_sales DESC
           ) AS rnk
    FROM sales_6m
)
SELECT region, product_id, total_sales
FROM ranked
WHERE rnk <= 3;
```

2️⃣ Orders where shipping took longer than the month's average

Approach:

- Compute shipping duration per order
- Compute average duration per month
- Compare each order's duration with the monthly average

```
WITH order_durations AS (
    SELECT order_id,
           EXTRACT(YEAR FROM order_date) AS order_year,
           EXTRACT(MONTH FROM order_date) AS order_month,
           DATE_PART('day', shipped_date - order_date) AS ship_days
    FROM orders
),
month_avg AS (
    SELECT order_year,
           order_month,
           AVG(ship_days) AS avg_ship_days
    FROM order_durations
    GROUP BY order_year, order_month
)
SELECT o.order_id,
       o.ship_days,
       m.avg_ship_days
FROM order_durations o
JOIN month_avg m
  ON o.order_year = m.order_year
 AND o.order_month = m.order_month
WHERE o.ship_days > m.avg_ship_days;
```

3 Recursive query: list all subordinates for a given manager

Approach:

- Table `employees(emp_id, emp_name, manager_id)`
- Anchor: direct reports of manager X
- Recursive step: employees whose `manager_id` = one of previously found emp_ids

```
WITH RECURSIVE subordinates AS (
    -- anchor: direct reports
```

```
    SELECT emp_id, emp_name, manager_id
    FROM employees
    WHERE manager_id = :manager_id   -- pass manager here

    UNION ALL

    -- recursion: indirect reports
    SELECT e.emp_id, e.emp_name, e.manager_id
    FROM employees e
    INNER JOIN subordinates s
        ON e.manager_id = s.emp_id
)
SELECT * FROM subordinates;
```

4️⃣ Deduplicate a huge table while keeping the most recent record

Approach:

- Use `ROW_NUMBER()` partitioned by the natural key
- Order by `updated_at DESC` to get the latest record
- Pick `row_number = 1`

```
WITH ranked AS (
    SELECT *,
           ROW_NUMBER() OVER (
             PARTITION BY natural_key_column
             ORDER BY updated_at DESC
           ) AS rn
    FROM big_table
)
SELECT *
FROM ranked
WHERE rn = 1;
```

(Replace `natural_key_column` with the column(s) that define duplicates.)

◇ Python

5️⃣ Efficiently read/transform/write a 20GB CSV daily to a database

Approach:

- 20 GB is too big for memory; process in **chunks** using `pandas.read_csv(chunksize=…)` or plain `csv` reader.
- Apply transformation logic chunk by chunk.
- Use DB bulk-insert (e.g., `to_sql(if_exists='append')` with `method='multi'` or `executemany`).
- Wrap in a streaming pipeline.

```
import pandas as pd
from sqlalchemy import create_engine

engine = create_engine("postgresql://user:pass@host/db")

chunksize = 500_000  # tune this
for chunk in pd.read_csv("daily_file.csv", chunksize=chunksize):
    # transform
    chunk["sale_date"] = pd.to_datetime(chunk["sale_date"])
    chunk["amount"] = chunk["amount"].fillna(0)

    # load to DB
    chunk.to_sql("sales", engine, if_exists="append",
                 index=False, method="multi")
```

(For ultra-high throughput, use COPY / bulk-load utilities.)

6️⃣ Detect anomalies in sales data (3 std-dev spikes/drops)

Approach:

- Compute mean & std for your metric.
- Flag values beyond ±3 σ as anomalies.
- Vectorised with NumPy for speed.

```
import numpy as np
import pandas as pd

def detect_anomalies(df, col="sales_amount"):
    mu = df[col].mean()
    sigma = df[col].std()
    df["anomaly_flag"] = np.where(
        (df[col] > mu + 3*sigma) | (df[col] < mu - 3*sigma),
        True, False
    )
    return df[df["anomaly_flag"]]

# usage:
# anomalies = detect_anomalies(sales_df, "daily_sales")
```

(In streaming, maintain rolling mean & std via Welford's algorithm.)

7 Group transactions by customer → return top N by purchase frequency (no Pandas)

Approach:

- Use built-in `collections.Counter` or `defaultdict(int)`.
- Count per customer in one pass.
- Use `heapq.nlargest` to get top-N.

```
from collections import Counter
import heapq

def top_n_customers(transactions, n=5):
    # transactions is an iterable of (cust_id, amount)
    counts = Counter(cust_id for cust_id, _ in transactions)
    return heapq.nlargest(n, counts.items(), key=lambda x: x[1])
```

```
# Example:
# top5 = top_n_customers(transactions_list, n=5)
```

(Memory-light, streaming friendly.)

◇ PySpark / Big Data

8️⃣ Solve the small file problem in ADLS/S3

Problem: Many tiny CSV/Parquet files cause slow Spark jobs.

Approach:

- Use Spark to **coalesce/repartition** into fewer large files after read.
- Ingest using `autoMerge` or copy into staging with combine.
- Adjust your upstream writers to produce fewer files.

```
df = spark.read.csv("abfss://container@account.dfs.core.windows.net/path")

# Combine small files into bigger ones (e.g., 256MB)
df.repartition(20).write.mode("overwrite").parquet("/combined/path")
```

(Use `spark.sql.files.maxPartitionBytes` to tune reading.)

9️⃣ Design a PySpark job to join billions of rows efficiently

Approach:

- Make sure both DataFrames are partitioned & bucketed on join key.
- If one side is small enough, **broadcast join**.
- Otherwise, increase shuffle parallelism & use sorted bucket join.

```
from pyspark.sql.functions import broadcast

# large_df (billions), small_df (lookup)
result = large_df.join(broadcast(small_df), "key_column", "inner")
```

(Or bucket both tables by key and `spark.sql.bucketing.enabled=true`.)

🔟 Handle key skew in a Spark join

Problem: One key dominates → one reducer overloaded.

Approach:

- **Salting**: Add a random number to the key on the skewed side.
- Pre-replicate small DF across salts.
- Join on (key,salt) then remove salt after join.
- Or use `skewHint` in Spark 3.2+.

```
from pyspark.sql.functions import col, lit, rand

# add salt to skewed df
salted_big = big_df.withColumn("salt", (rand()*10).cast("int"))
replicated_small = small_df.crossJoin(
    spark.range(10).withColumnRenamed("id", "salt")
)

joined = salted_big.join(
    replicated_small,
    on=[col("big_df.key")==col("small_df.key"), "salt"],
    how="inner"
).drop("salt")
```

(Also consider using adaptive execution & AQE's skew join handling.)

◇ Data Modeling & Architecture

1️⃣1️⃣ Q: Design a data model for a global supply chain supporting both historical analytics and near-real-time processing.

Approach (high-level goals):

- Support OLTP for operational systems + OLAP for analytics.
- Capture history (inventory, shipments, orders) and stream real-time events (shipments, IoT sensors).
- Make model queryable for KPIs (lead time, fill rate, stockouts).
- Ensure partitioning, time-series friendliness, and easy joins across domains.

Key components / tables & formats:

- **Raw Events / Ingest** (event store in object storage / Kafka): `events(topic, event_time, payload_json, source, partition_key)`
- **Staging / Bronze** (append-only Delta/Parquet): store raw, untransformed events with metadata.
- **Silver (Enriched / Conformed)**: normalized dimension & fact tables:
  - `dim_product(product_id, sku, category, attributes, effective_from, effective_to)`
  - `dim_location(location_id, country, region, type)`
  - `dim_supplier(supplier_id, ...)`
  - `fact_orders(order_id, product_id, qty, order_ts, order_status, source_system)`
  - `fact_shipments(shipment_id, order_id, ship_ts, delivery_ts, carrier, status)`
  - `inventory_snapshots(store_id, product_id, qty_on_hand, snapshot_ts)` (time-partitioned)
- **Gold / Aggregates:** daily/weekly rollups, SLA tables, ML feature tables.

Streaming + Historical pattern:

- Ingest real-time events via Kafka → land in Bronze Delta (append) → streaming jobs (Spark Structured Streaming) enrich & upsert to Silver (Delta) → materialize Gold aggregates periodically or via incremental streaming.

Design considerations & details:

- **SCD for dimensions** (Type 2) to preserve history.
- **Partitioning**: time (date) + region to optimize scans.
- **Time-series tables**: use Delta time travel for audits.
- **Late data handling**: watermarking, window grace period, event-time processing.
- **Idempotency & dedupe**: use unique event IDs and upsert patterns.
- **Governance**: catalog (Glue/Unity), lineage, RBAC.

Example ingestion flow (pseudo):
Kafka → Bronze Delta (`/bronze/events/yyyy=.../`) → Spark job: parse JSON, enrich (lookup dims), write to Silver with upsert keys → downstream BI reads Gold.

1️⃣2️⃣ Q: Implement SCD Type 2 in a data warehouse (concept + code).

Approach (concept):
SCD Type 2 = preserve history by creating new rows for changed attributes while marking previous versions as inactive.

Key columns: `natural_key`, `surrogate_key`, `attribute_cols...`, `effective_from`, `effective_to`, `is_current` (or `current_flag`).

Implementation patterns: (a) SQL MERGE (preferred in DWs that support MERGE), (b) PySpark upsert to Delta.

SQL MERGE example (pseudo, Snowflake/SQL Server style):

```
MERGE INTO dim_customer tgt
USING staging_customer src
  ON tgt.customer_nk = src.customer_nk
  AND tgt.is_current = 1
WHEN MATCHED AND (
     tgt.name <> src.name OR tgt.address <> src.address
) THEN
  UPDATE SET is_current = 0, effective_to = current_date
WHEN NOT MATCHED OR (MATCHED BUT CHANGED) THEN
  INSERT (surrogate_key, customer_nk, name, address, effective_from,
effective_to, is_current)
  VALUES (NEXTVAL('dim_customer_seq'), src.customer_nk, src.name,
```

```
src.address, current_date, '9999-12-31', 1);
```

PySpark / Delta pattern (idempotent):

1. Read `staging` (new data) and `dim` (current records).
2. Join staging → current on natural key.
3. For changed records: update current row `is_current=0, effective_to=now`, and insert new row with `is_current=1`.
4. Use Delta `merge` (Databricks):

```
from delta.tables import DeltaTable
delta_dim = DeltaTable.forPath(spark, "/delta/dim_customer")
src_df = spark.read.format("parquet").load("/staging/customers")

delta_dim.alias("tgt").merge(
    src_df.alias("src"),
    "tgt.customer_nk = src.customer_nk"
).whenMatchedUpdate(
    condition = "tgt.is_current = 1 AND (tgt.name <> src.name OR
tgt.address <> src.address)",
    set = { "is_current": "0", "effective_to": "current_date()" }
).whenNotMatchedInsertAll().execute()
# then insert new rows for changed ones
```

Notes & best practices:

- Ensure transactional upserts (Delta/Iceberg/Hudi).
- Keep `effective_from` on inserted rows and `effective_to=9999-12-31` on active rows.
- Maintain surrogate keys for joins.
- Test backfills and idempotency.

1️⃣3️⃣ Q: How do you unify schemas from multiple ERP systems for reporting?

Approach (goal): build a canonical/conformed schema & mapping layer so reports use uniform fields.

Steps / Pattern:

1. **Inventory sources**: list ERP systems, export schemas, field meanings.
2. **Define canonical model**: data contract (standard fields, types, units, enumerations). Example: `order_id, order_date_utc, customer_id, product_sku, qty, currency, amount_usd, org_unit, source_system`.
3. **Field mapping & lineage**: maintain a mapping table `mapping(source_system, source_field, canonical_field, transform_expr)`; document differences.
4. **Schema registry & validation**: use JSON schema or Avro for events; validate incoming data and store raw.
5. **Transformation layer**: ETL/ELT jobs that apply mapping & transformations (unit conversions, currency conversion, code mapping). Encapsulate transformations in reusable functions or dbt models.
6. **Conformance rules**: standardize enums (e.g., status codes), normalize dates to UTC, reconcile missing fields.
7. **Testing & QA**: add data quality checks for nulls, cardinality, value ranges.
8. **Versioning & evolution**: track mapping changes over time; support schema evolution.

Technical choices:

- Use a **data lakehouse** (Delta/Iceberg) as canonical store; DBT or Spark for transformations; a schema registry for streaming (Confluent/Avro).
- Implement a **lookup table** for code translations (ERP1_status → CANON_STATUS).

Example mapping transform (pseudocode):

```
SELECT
  coalesce(e1.order_id, e2.order_ref) AS order_id,
  to_utc(e.order_ts, 'source_tz') AS order_date_utc,
  map_status(e.status_code) AS order_status,
  convert_currency(e.amount, e.currency, 'USD') AS amount_usd
```

```
FROM raw_orders e
```

Governance: central docs, SLAs, automated tests, and a data steward owning canonical model changes.

◇ Real-Time & Cloud

1️⃣4️⃣ Q: Design a real-time order tracking pipeline (Kafka + Spark Streaming + Cloud DW).

Requirements: low-latency tracking of orders (status updates), ability to support dashboards & alerts, exactly-once or at-least-once semantics, handle replays/late events.

Architecture components:

- **Producers:** Order systems / microservices publish events to Kafka topics (`orders.events`).
- **Kafka (or Event Hub / Kinesis):** partition by `order_id` (for ordering), retention for replay.
- **Stream Processing:** Spark Structured Streaming (or Flink) consumes from Kafka, performs enrichment (lookup dims via broadcast state or state store), aggregates windows (e.g., status timelines), writes to sink.
- **Sink:** Delta Lake (low-latency store) plus materialized views / push to Cloud DW (Synapse / Redshift / BigQuery) for BI.
- **Downstream:** BI dashboard (Power BI / Looker), alerting (if delivery delayed), and OLTP update endpoints if needed.

Key details:

- **Event format**: `order_id, event_type, event_ts, payload, event_id` (dedupe id).
- **Stateful processing**: maintain last status per order and write upserts to a `current_order_status` table.
- **Watermarks & lateness**: handle late-arriving events with watermark and allowed lateness window.

- **Exactly-once semantics**: use Kafka offsets + idempotent writes or transactional writes in Delta (checkpointing).
- **Scaling**: partitioning by key, autoscale streaming cluster.

Simplified Spark Structured Streaming (pseudo):

```
df = spark.readStream.format("kafka").option(...).load()
events = df.selectExpr("CAST(value AS STRING) as
json").select(from_json("json", schema).alias("d")).select("d.*")

# dedupe by event_id using watermark + dropDuplicates
deduped = events.withWatermark("event_ts", "10
minutes").dropDuplicates(["event_id"])

# update current status table
deduped.writeStream.format("delta").option("checkpointLocation",
"/checkpoints/orders") \
  .foreachBatch(lambda batch_df, batch_id: upsert_to_delta(batch_df,
"/delta/current_order_status")) \
  .start()
```

Monitoring & SLOs: track consumer lag, processing time, late-event counts, and SLA alerts.

1️⃣5️⃣ Q: Migrate 10TB from on-prem SQL Server → Azure Synapse with minimal downtime.

Approach (strategy): full initial load + continuous change data capture (CDC) sync → cutover. Use Azure Data Factory (ADF) or Azure Database Migration Service (DMS) + transactional log replication to minimize downtime.

Step-by-step plan:

1. **Assessment & prep:** catalog tables, indexes, constraints, data types, compatibility issues. Identify large tables and dependency order.
2. **Schema migration:** convert schemas (types, constraints) and create target objects in Synapse. Use Data Migration Assistant to identify blockers.

3. **Initial bulk load:** perform high-throughput bulk copy:
    a. Use ADF Copy Activity with `PolyBase` (recommended for Synapse) or `COPY` for Azure Synapse dedicated SQL pool.
    b. For very large volumes, consider offline transfer (Azure Data Box) if network is limited.
4. **Enable CDC on source:** use SQL Server CDC / Transaction Log or use third-party CDC tool (Attunity/Qlik/Striim) or Azure DMS for continuous replication.
5. **Continuous sync (near real-time):** use CDC to apply changes incrementally to Synapse while business continues on source. This keeps target near-synced.
6. **Validation & parallel run:** run parallel workloads against Synapse (reports + queries) and compare results (row counts, checksums).
7. **Cutover plan:** choose a low-traffic window → briefly pause writes to source, apply final CDC changes (drain), switch applications/BI to Synapse, and monitor. Downtime limited to final drain time.
8. **Post-cutover:** monitor performance, rebuild indexes, enable tuning, and decommission source or keep read-only.

Tools & options:

- **Azure DMS**: guidance and assistance for minimal downtime migrations; supports online migrations.
- **ADF**: good for bulk + incremental copy; PolyBase for high throughput.
- **Transactional replication / CDC**: for low-latency changes.
- **Validation**: use checksums, row counts, and query result comparison.

Risks & mitigations:

- **Network throughput** → use compression, parallel copy, or physical Data Box.
- **DDL drift** → lock schemas during final sync or script schema changes carefully.
- **Large table cutover** → perform table partitioning and migrate partition-by-partition, or use shadow tables and swap.