Data Architecture & Advanced SQL

JOIN • CTE • Window Functions • Time Travel

Hands-on: Event Pipeline (orders, sessions, payments)

Data Architecture & Advanced SQL

Course: Business Analytics

Topics: JOIN • CTE • Window Functions • Time Travel

Hands-on: Event Pipeline (orders, sessions, payments)

Agenda

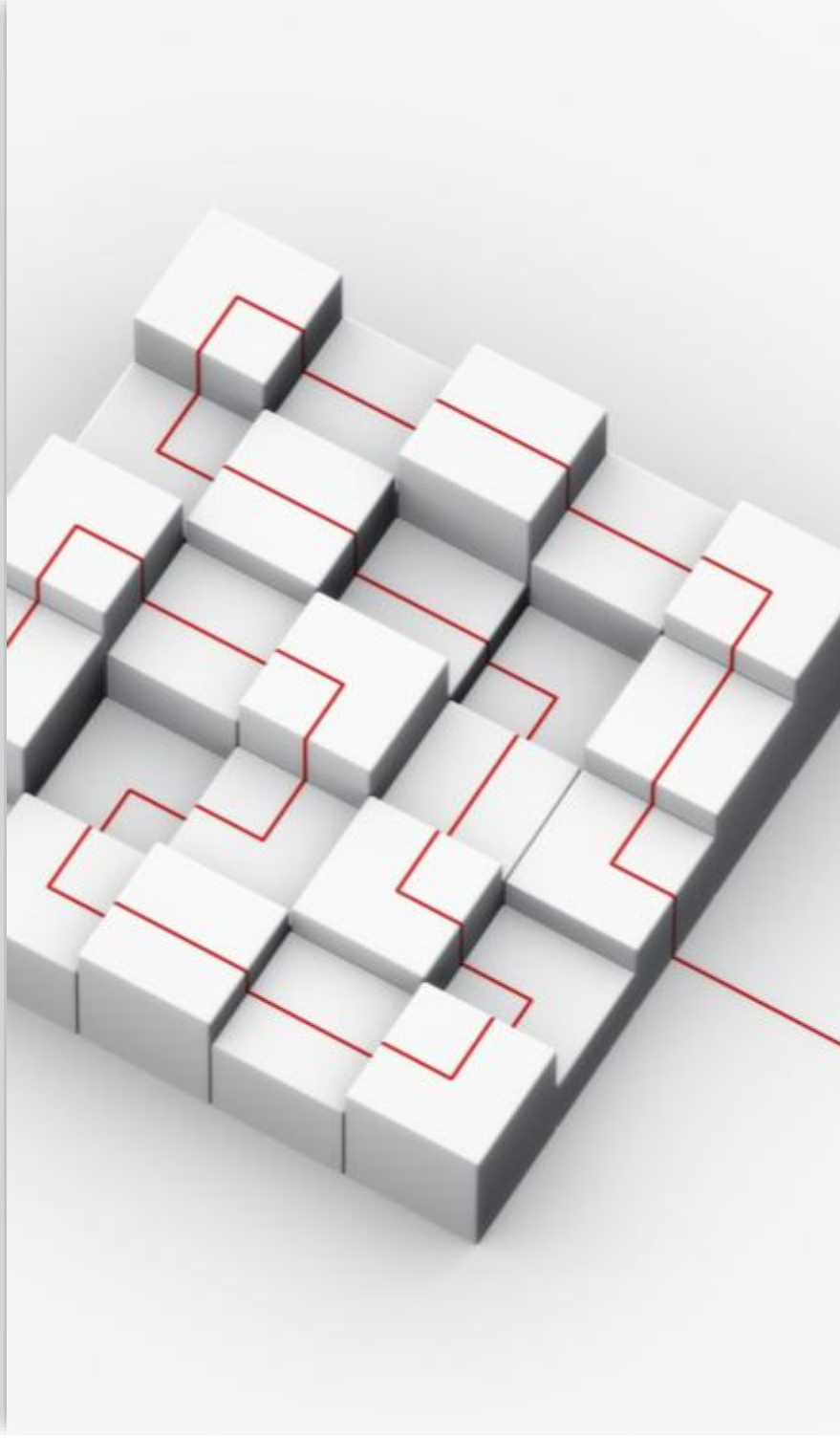
1) Data architecture for analytics: OLTP vs OLAP, Star/Snowflake, Lakehouse, Medallion
2) Advanced JOINs: inner/outer, cross, semi/anti, join strategies
3) CTE & Recursive CTE: readability, reuse, graph/tree examples
4) Window functions: ranking, moving average, frames, sessionization
5) Time Travel: query historical data (Snowflake, BigQuery, Delta Lake)
6) Hands-on: Event pipeline queries (orders, sessions, payments)

Learning Outcomes

- Explain core data architecture patterns and trade-offs
- Write correct and performant JOINs (including semi/anti)
- Use CTEs and recursive CTEs to structure complex logic
- Apply window functions for rolling metrics and cohort analysis
- Query historical states using time travel features
- Build an end-to-end SQL pipeline over events data

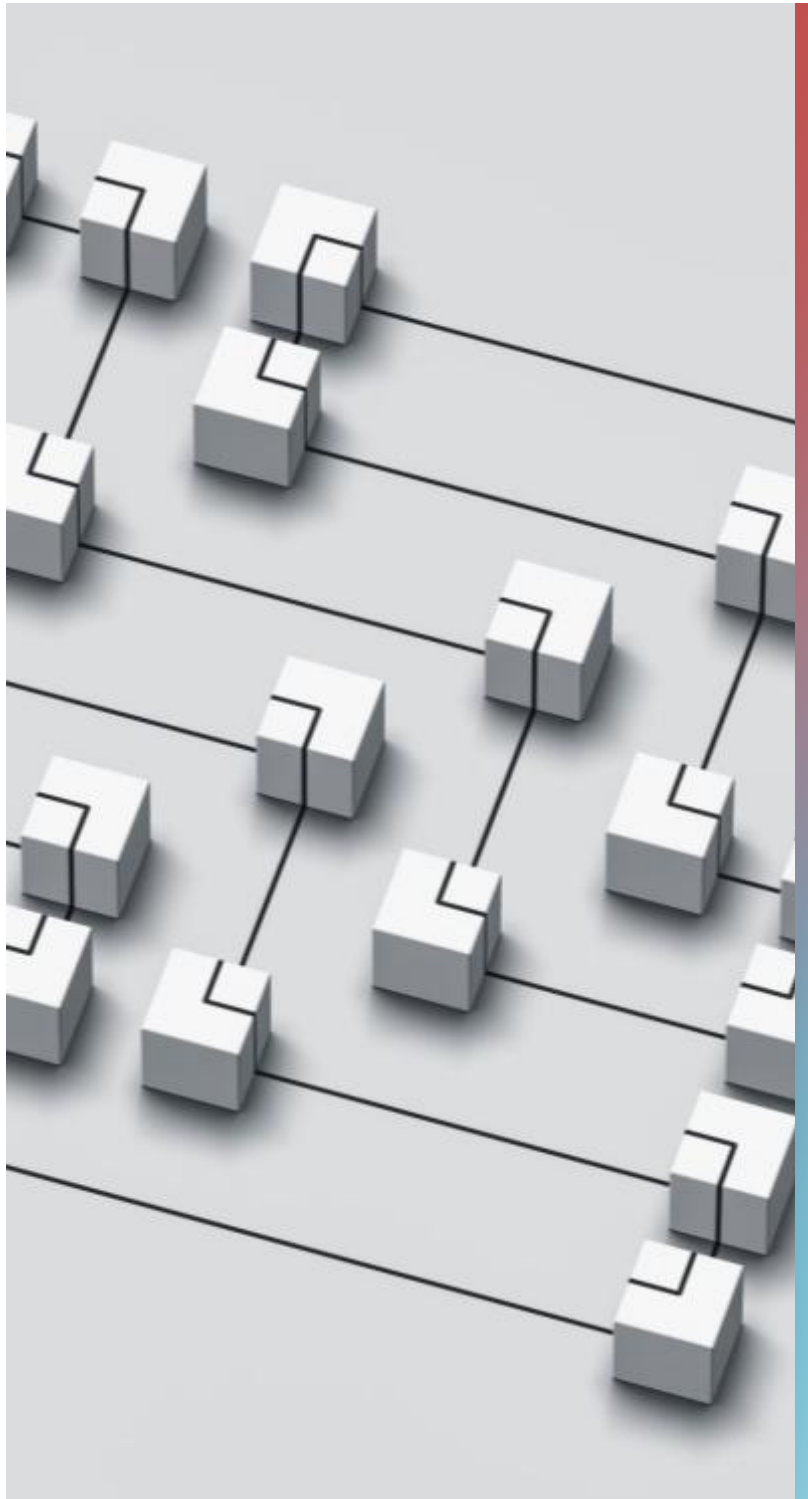
Part I — Data Architecture for Analytics

- OLTP vs OLAP workloads: read/write patterns
- Normalization vs denormalization
- Dimensional modeling (Star & Snowflake)
- Lakehouse & Medallion architecture
- Batch vs streaming; CDC & event-driven pipelines

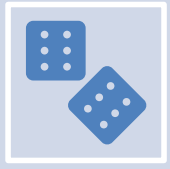


OLTP vs OLAP

- OLTP: many small writes, normalized schema, point lookups
- OLAP: fewer large reads, denormalized schema, columnar storage
- Implication: separate systems, or lakehouse with serving layers



Dimensional Modeling: Star Schema



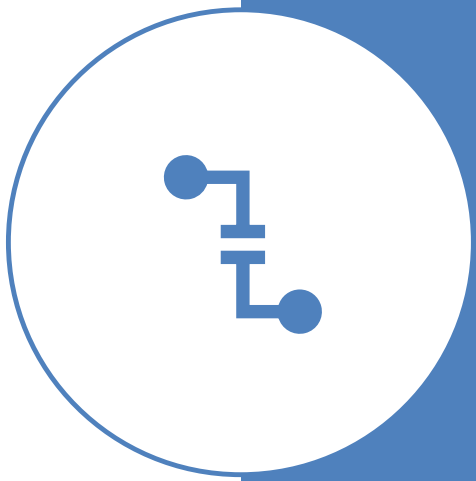
Fact table: transactional grain (e.g., orders, payments)

Dimension tables: customers, products, time, channel

Benefits: simplicity, predictable JOINs, good for BI

Dimensional Modeling: Snowflake Schema

- Normalized dimensions (e.g., product → category → department)
- Pros: reduced redundancy; Cons: more joins and complexity
- Use when dimension updates are frequent and large



SCD Type 2
(Slowly Changing Dimensions)

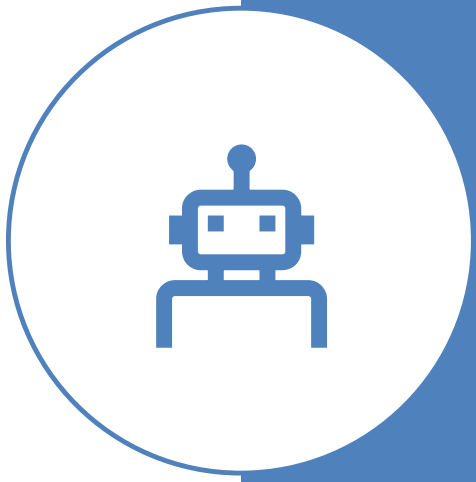
- Maintain history via versioned rows with validity ranges
- Columns: valid_from, valid_to, is_current
- Join fact → dimension using time-sensitive key + range

SCD Type 2 (Slowly Changing Dimensions) — Code

```
-- Example SCD2 join (PostgreSQL-ish)
SELECT f.order_id, d.customer_tier
FROM fact_orders f
JOIN dim_customer_scd d
  ON f.customer_id = d.customer_id
 AND f.order_ts >= d.valid_from
 AND (d.valid_to IS NULL OR f.order_ts < d.valid_to);
```

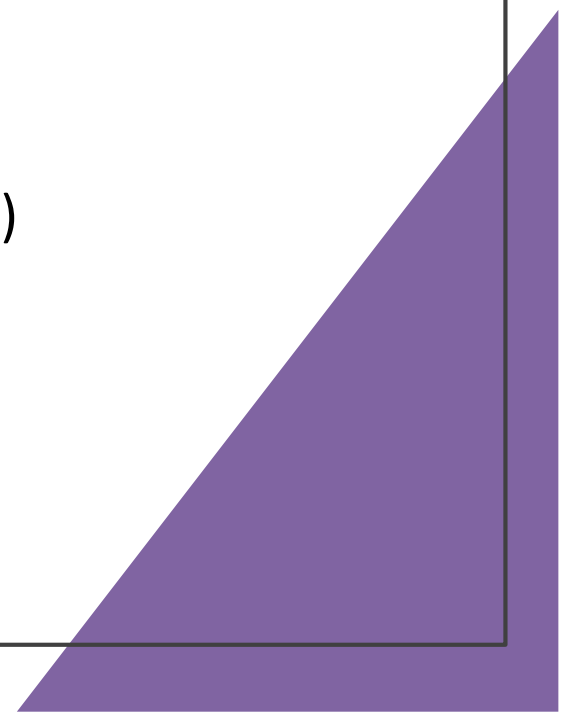
Lakehouse & Medallion

- Bronze: raw ingests (append-only, minimal transforms)
- Silver: cleaned & conformed (dedupe, schema), ready for JOINs
- Gold: curated marts for BI/ML (star schemas, aggregates)



Batch vs Streaming

- Batch: simpler orchestration, higher latency
- Streaming: lower latency, complexity (exactly-once, upserts)
- Micro-batch (e.g., Spark Structured Streaming) bridges both



CDC (Change Data Capture) & Event-Driven

- Capture inserts/updates/deletes from OLTP (e.g., Debezium)
- Publish to topics; process into lakehouse layers
- Upsert into silver; maintain gold aggregates

Event Data Model (orders, sessions, payments)

- Events: event_time, event_type, user_id/session_id
- orders(order_id, user_id, session_id, amount, status, created_at)
- payments(payment_id, order_id, amount, method, status, paid_at)
- sessions(session_id, user_id, start_at, end_at, channel)

Part II — Advanced JOINs

- Inner/Left/Right/Full/Cross
- Semi/Anti joins (EXISTS / NOT EXISTS)
- Join keys, NULL handling, duplicates, cardinality
- Join order & execution strategies

Equi-Join vs Non-Equi Join

Equi-join on keys: common for dimensions

Non-equi: range joins (e.g., SCD2 validity windows)

Inner vs Outer Joins

INNER: rows that match on both sides

LEFT/RIGHT: preserve one side with NULLs for missing matches

FULL: preserve both sides

Inner vs Outer Joins — Code

```
-- Outer join example: orders without payments (left anti via WHERE)
SELECT o.order_id, o.amount
FROM orders o
LEFT JOIN payments p ON p.order_id = o.order_id
WHERE p.order_id IS NULL;
```

Semi & Anti Joins

SEMI: rows from A with at least one match in B (EXISTS)

ANTI: rows from A with no matches in B (NOT EXISTS)

Useful for integrity checks and funnels

Semi & Anti Joins — Code

```
-- Semi join: orders that have payments
SELECT o.order_id
FROM orders o
WHERE EXISTS (SELECT 1 FROM payments p WHERE p.order_id = o.order_id);

-- Anti join: orders lacking payments
SELECT o.order_id
FROM orders o
WHERE NOT EXISTS (SELECT 1 FROM payments p WHERE p.order_id = o.order_id);
```

Join Gotchas & NULLs

NULLs do not equal anything (even NULL) in equality joins

Coalesce or surrogate keys for null-safe comparisons

Beware duplicated keys → row explosion

Join Gotchas & NULLs — Code

```
-- Null-safe equality (BigQuery)
SELECT *
FROM a
LEFT JOIN b
ON IFNULL(a.key, '__NULL__') = IFNULL(b.key, '__NULL__');
```

Join Strategies & Performance

- Broadcast/small table join vs shuffle join
- Hash join vs sort-merge join
- Pushdown filters, pre-aggregate, selective projections

Part III — CTEs & Recursive CTEs

- CTE: improves readability; can aid reuse
- Recursive CTE: trees/graphs, hierarchies, sequences
- Materialization behavior varies by engine (cost-based)

CTE for Organization

Break complex logic into named steps (bronze → silver → gold)

Avoid repeated subqueries; annotate transformations

CTE for Organization — Code

```
WITH clean_orders AS (
  SELECT order_id, user_id, amount, created_at::date AS order_date
  FROM raw_orders
  WHERE status = 'confirmed'
), daily_rev AS (
  SELECT order_date, SUM(amount) AS revenue
  FROM clean_orders GROUP BY 1
)
SELECT * FROM daily_rev ORDER BY order_date;
```

Recursive CTE — Sequence & Tree

Generate series; walk parent-child relations

Add a termination condition to avoid infinite loops

Recursive CTE — Sequence & Tree — Code

```
-- Tree traversal (PostgreSQL-style)
WITH RECURSIVE org AS (
  SELECT id, parent_id, name, 0 AS depth
  FROM dept WHERE parent_id IS NULL
  UNION ALL
  SELECT d.id, d.parent_id, d.name, o.depth + 1
  FROM dept d
  JOIN org o ON d.parent_id = o.id
)
SELECT * FROM org ORDER BY depth, name;
```

Part IV — Window Functions

Compute metrics over partitions without collapsing rows

RANK / DENSE_RANK / ROW_NUMBER

SUM / AVG over frames for moving windows

Window Syntax

- function(...) OVER (PARTITION BY ... ORDER BY ... [FRAME])
- Frames: ROWS vs RANGE vs GROUPS
- Default frame varies by engine; specify explicitly

Window Syntax — Code

```
-- 7-day rolling revenue (date-grain)
SELECT order_date,
       SUM(amount) OVER (
         ORDER BY order_date
         ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
       ) AS rev_7d
FROM daily_orders;
```

Ranking Examples

Top-N per category via ROW_NUMBER()

Percentiles with NTILE() or PERCENT_RANK()

Ranking Examples — Code

```
-- Top 3 products per category
SELECT *
FROM (
  SELECT category, product_id, sales,
         ROW_NUMBER() OVER (PARTITION BY category ORDER BY sales DESC) AS rn
  FROM product_sales
) s
WHERE rn <= 3;
```

Sessionization with Windows

- Assign sessions by gaps between events
- Use LAG() to detect gap > threshold
- Some engines have SESSION windows; generic SQL shown

Sessionization with Windows — Code

```
-- Sessionize: start new session if gap > 30 minutes
WITH g AS (
  SELECT user_id, event_ts,
         CASE WHEN event_ts - LAG(event_ts) OVER (PARTITION BY user_id ORDER BY
event_ts)
                   > INTERVAL '30 minutes' OR LAG(event_ts) IS NULL
              THEN 1 ELSE 0 END AS is_new
  FROM events
),
s AS (
  SELECT user_id, event_ts,
         SUM(is_new) OVER (PARTITION BY user_id ORDER BY event_ts
                           ROWS UNBOUNDED PRECEDING) AS session_num
  FROM g
)
SELECT user_id, MIN(event_ts) AS session_start, MAX(event_ts) AS session_end,
COUNT(*) AS events
FROM s
GROUP BY user_id, session_num;
```

Cohort & Retention with Windows

- Cohort by first order month; retention by return months
- Use MIN() OVER for cohort; use DATEDIFF for offsets

Cohort & Retention with Windows — Code

```
WITH firsts AS (
  SELECT user_id,
         DATE_TRUNC('month', MIN(order_ts)) OVER (PARTITION BY user_id) AS cohort_month
         order_ts
  FROM orders
),
ret AS (
  SELECT user_id, cohort_month,
         DATE_DIFF('month', cohort_month, DATE_TRUNC('month', order_ts)) AS month_idx
  FROM firsts
)
SELECT cohort_month, month_idx, COUNT(DISTINCT user_id) AS users
FROM ret
GROUP BY 1,2 ORDER BY 1,2;
```

Part V — Time Travel

- Query historical table versions to audit or recover
- Common in Snowflake, BigQuery, Delta Lake
- Limits & retention vary by platform

Time Travel — Snowflake

- Point-in-time query with AT/BEFORE clause
- Retention: typically 1–90 days (account setting)

Time Travel — Snowflake — Code

```
-- Snowflake
SELECT * FROM sales AT (TIMESTAMP => TO_TIMESTAMP('2025-08-31 00:00:00'));
-- or by offset seconds
SELECT * FROM sales AT (OFFSET => -3600);
```

Time Travel — BigQuery

- Use FOR SYSTEM_TIME AS OF timestamp
- Default retention ~7 days for table snapshots

Time Travel — BigQuery — Code

```
-- BigQuery
SELECT * FROM `proj.ds.sales` FOR SYSTEM_TIME AS OF
TIMESTAMP_SUB(CURRENT_TIMESTAMP(), INTERVAL 1 DAY);
```

Time Travel — Delta Lake

Query by VERSION AS OF or TIMESTAMP AS OF

Inspect history via DESCRIBE HISTORY

Time Travel — Delta Lake — Code

```
-- Delta Lake
SELECT * FROM delta.`/tables/sales` VERSION AS OF 12;
-- Or:
SELECT * FROM delta.`/tables/sales` TIMESTAMP AS OF '2025-08-31T00:00:00Z';
```

Part VI — Hands-on: Event Pipeline

Dataset: orders, payments, sessions

Tasks: funnel, anomalies, rolling metrics, time-travel checks

Environment-agnostic SQL (PostgreSQL/SparkSQL/BigQuery-style)

Create Bronze (raw → cleaned)

- Deduplicate, cast types, basic validation

Create Bronze (raw → cleaned) — Code

```
WITH o AS (
  SELECT DISTINCT order_id, user_id, session_id, amount::NUMERIC, status, created_at::timestamp AS order_ts
  FROM raw_orders
),
p AS (
  SELECT DISTINCT payment_id, order_id, amount::NUMERIC, status, paid_at::timestamp AS paid_ts
  FROM raw_payments
),
s AS (
  SELECT DISTINCT session_id, user_id, start_at::timestamp AS session_start, end_at::timestamp AS session_end, channel
  FROM raw_sessions
)
SELECT COUNT(*) FROM o; -- sanity checks
```

Silver: Conform & Joinable Tables

- Normalize status values; ensure referential integrity

Silver: Conform & Joinable Tables — Code

```
WITH clean_orders AS (
  SELECT *, LOWER(status) AS status_norm
  FROM o
  WHERE amount >= 0 AND order_ts IS NOT NULL
), clean_payments AS (
  SELECT *, LOWER(status) AS status_norm FROM p
), clean_sessions AS (
  SELECT * FROM s WHERE session_start <= COALESCE(session_end, session_start + INTERVAL '8 hours')
)
SELECT COUNT(*) FROM clean_orders;
```

Gold: Core Marts

- fact_orders, fact_payments, dim_user, dim_channel
- Daily revenue, conversion rates, AOV

Gold: Core Marts — Code

```
-- Daily revenue & orders
WITH daily AS (
  SELECT DATE_TRUNC('day', order_ts) AS d, SUM(amount) AS revenue, COUNT(*) AS orders
  FROM clean_orders WHERE status_norm='confirmed' GROUP BY 1
)
SELECT d, revenue, orders, revenue::NUMERIC / NULLIF(orders,0) AS aov
FROM daily ORDER BY d;
```

Funnel: session → order → payment

- Identify drop-offs with left/semi/anti joins

Funnel: session → order → payment — Code

```
WITH session_to_order AS (
  SELECT s.session_id, COUNT(DISTINCT o.order_id) AS orders
  FROM clean_sessions s
  LEFT JOIN clean_orders o ON o.session_id = s.session_id
  GROUP BY 1
),
order_to_payment AS (
  SELECT o.order_id, EXISTS(SELECT 1 FROM clean_payments p
WHERE p.order_id=o.order_id AND p.status_norm='succeeded') AS paid
  FROM clean_orders o
)
SELECT
  (SELECT COUNT(*) FROM clean_sessions) AS sessions,
  (SELECT COUNT(DISTINCT order_id) FROM clean_orders) AS orders,
  (SELECT COUNT(*) FROM order_to_payment WHERE paid) AS paid_orders;
```

Payment Mismatch Checks

- Amount mismatches, multiple payments per order, failed payments

Payment Mismatch Checks — Code

```
-- Mismatch report
SELECT o.order_id, o.amount AS order_amount, SUM(p.amount) AS paid_amount
FROM clean_orders o
LEFT JOIN clean_payments p ON p.order_id=o.order_id AND p.status_norm='succeeded'
GROUP BY 1, o.amount
HAVING SUM(p.amount) IS DISTINCT FROM o.amount;
```

Rolling KPIs with Windows

- 7-day revenue, 30-day conversion rate

Rolling KPIs with Windows — Code

```
-- 7-day rolling revenue & orders
WITH daily AS (
  SELECT DATE_TRUNC('day', order_ts) AS d, SUM(amount) AS revenue, COUNT(*) AS orders
  FROM clean_orders WHERE status_norm='confirmed' GROUP BY 1
)
SELECT d,
       SUM(revenue) OVER (ORDER BY d ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS rev_7d,
       SUM(orders) OVER (ORDER BY d ROWS BETWEEN 6 PRECEDING AND CURRENT ROW) AS orders_7d
FROM daily ORDER BY d;
```

User Cohorts & LTV

- First-order cohort, months since cohort, cumulative revenue

User Cohorts & LTV — Code

```
WITH first_order AS (
  SELECT user_id, MIN(order_ts) AS first_ts
  FROM clean_orders GROUP BY 1
),
cohort_orders AS (
  SELECT o.user_id,
         DATE_TRUNC('month', f.first_ts) AS cohort_month,
         DATE_DIFF('month', DATE_TRUNC('month', f.first_ts),
DATE_TRUNC('month', o.order_ts)) AS m,
         o.amount
  FROM clean_orders o JOIN first_order f USING(user_id)
)
SELECT cohort_month, m, SUM(amount) AS revenue
FROM cohort_orders
GROUP BY 1,2 ORDER BY 1,2;
```

Anomaly Detection (SQL-first)

- Z-score vs global mean; quick-and-dirty monitoring

Anomaly Detection (SQL-first) — Code

```
WITH daily AS (
  SELECT DATE_TRUNC('day', order_ts) AS d, SUM(amount) AS revenue
  FROM clean_orders WHERE status_norm='confirmed' GROUP BY 1
),
stats AS (
  SELECT AVG(revenue) AS mu, STDDEV_SAMP(revenue) AS sigma FROM daily
)
SELECT d, revenue, (revenue - mu)/NULLIF(sigma,0) AS z
FROM daily CROSS JOIN stats
ORDER BY d;
```

Time Travel in the Pipeline

- Audit historical states of facts/dimensions
- Compare to current to detect retroactive changes

Time Travel in the Pipeline — Code

```
-- Snowflake: yesterday's snapshot vs today
WITH y AS (
  SELECT * FROM fact_orders AT (OFFSET => -86400)
), t AS (
  SELECT * FROM fact_orders
)
SELECT COALESCE(y.order_id, t.order_id) AS order_id,
       y.amount AS amount_y, t.amount AS amount_t
FROM y FULL JOIN t USING(order_id)
WHERE y.amount IS DISTINCT FROM t.amount;
```

End-to-End Example Query

- From bronze to gold in one CTE pipeline

End-to-End Example Query — Code

```
WITH o AS (
  SELECT DISTINCT order_id, user_id, session_id, amount::NUMERIC,
LOWER(status) AS st, created_at::timestamp AS ts FROM raw_orders
),
p AS (
  SELECT DISTINCT order_id, amount::NUMERIC, LOWER(status)
AS st, paid_at::timestamp AS ts FROM raw_payments
),
s AS (
  SELECT DISTINCT session_id, user_id, start_at::timestamp AS start_ts
  FROM raw_sessions
),
confirmed AS (SELECT * FROM o WHERE st='confirmed'),
paid AS (SELECT order_id, SUM(amount) AS paid_amount FROM p
WHERE st='succeeded' GROUP BY 1),
joined AS (
  SELECT c.order_id, c.user_id, c.session_id, c.amount, c.ts
AS order_ts, COALESCE(p.paid_amount,0) AS paid_amount FROM confirmed c
LEFT JOIN paid p USING(order_id)
)
SELECT *,
       CASE WHEN paid_amount >= amount THEN 'paid' ELSE 'unpaid' END AS payment_status
FROM joined;
```

Exercises (1/2)

- Compute daily conversion rate: sessions → orders → paid
- Find orders with payment attempts > 1 and flag anomalies
- Top-3 channels by 7-day rolling revenue

Exercises (2/2)

- Build a cohort table by signup month and compute 90-day LTV
- Use time travel to compare today's gold mart vs 7 days ago
- Recursive CTE: expand a product category tree and roll up sales

Summary & What's Next

- We covered data architecture patterns for analytics
- Advanced SQL: JOINs, CTEs, windows, time travel
- Hands-on pipeline for events data
- Next: KPI Tree, Experimentation, and Causal Thinking

References

- Kimball & Ross — The Data Warehouse Toolkit
- Snowflake, BigQuery, Delta Lake official docs (time travel, windows)
- Spark SQL, PostgreSQL docs (CTE, window functions)

Extra Example — Window Frame Deep Dive

- Compare ROWS vs RANGE on date buckets
- Demonstrate GROUPS frame for ties (if supported)

Extra Example — Window Frame Deep Dive — Code

```
-- RANGE vs ROWS example (beware duplicates on ORDER BY key)
SELECT d, x,
       SUM(x) OVER (ORDER BY d ROWS BETWEEN 2 PRECEDING AND CURRENT ROW)  AS rows_win,
       SUM(x) OVER (ORDER BY d RANGE BETWEEN INTERVAL '2 day'
PRECEDING AND CURRENT ROW) AS range_win
FROM series;
```