Midterm Report-Phase2

1. ER-to-Relational **(4 pt)***:* Transform the ER diagram into relational model using SQL data definition language (DDL)

Turn boxes into tables (with simple primary keys).
Every entity in the ER diagram became its own table: customer, employee, supplier, product, and "order". Each of these have their own primary key according to the ER diagram that I used for the primary key in the creation of the tables.

Map attributes straight to columns (and split composites).
Regular attributes just become columns. The only "special" case was the Customer Address, which is composite in ER-land. In a relational table you can't keep a nested structure, so I flattened it into street, city, state, and zip to keep the first normal form. I also added basic constraints that match the data's reality: non-negative stock and prices, valid order statuses, and a default order_date = NOW() so inserts don't have to set a timestamp every time. I made customer.email and supplier.supplier_name unique to cut down on duplicates.

Convert one-to-many relationships into foreign keys.
The "many" side carries the pointer to the "one" side:

- product.supplier_id → supplier.supplier_id
- "order".customer_id → customer.customer_id
- "order".employee_id → employee.employee_id (nullable, since an order can exist even if we don't track the handler)
  I chose referential actions that match how the system should behave: deleting a supplier is restricted if products still reference it; deleting an employee sets null on existing orders (history stays intact); key updates cascade so FKs don't break.

Replace many-to-many with a join table.
Order ↔ Product is M:N in the ER model, so I introduced order_line(order_id, product_id, quantity, line_price) with a composite primary key (order_id, product_id). That prevents duplicate lines for the same product in the same order. The FK to "order" is ON DELETE CASCADE (lines go away with the order), and the FK to product is RESTRICT (we don't

want to delete a product that appears in old orders). Capturing quantity and line_price at the line level also “freezes” the transaction price.

Keep it normalized and usable.
Overall the schema lands in 3NF: each table is about one thing, attributes depend on the key, and repeating groups live in order_line. The keys, FKs, checks, and defaults carry over the ER diagram’s structure and rules, but in a way that’s practical for loading data and writing queries in PostgreSQL.

```sql
DROP SCHEMA IF EXISTS shop CASCADE;
CREATE SCHEMA shop;
SET search_path = shop;

CREATE TABLE customer (
  customer_id      INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  first_name       VARCHAR(50) NOT NULL,
  last_name        VARCHAR(50) NOT NULL,
  email            VARCHAR(255) UNIQUE,
  phone_number     VARCHAR(25),
  -- composite Address attribute flattened
  street           VARCHAR(120),
  city             VARCHAR(80),
  state            VARCHAR(50),
  zip              VARCHAR(15)
);

CREATE TABLE employee (
  employee_id      INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  first_name       VARCHAR(50) NOT NULL,
  last_name        VARCHAR(50) NOT NULL,
  position         VARCHAR(80),
  hire_date        DATE NOT NULL
);

CREATE TABLE supplier (
  supplier_id      INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  supplier_name    VARCHAR(120) NOT NULL,
  phone_number     VARCHAR(25),
  CONSTRAINT uq_supplier_name UNIQUE (supplier_name)
);

CREATE TABLE product (
  product_id       INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  product_name     VARCHAR(120) NOT NULL,
  category         VARCHAR(80),
  stock_quantity   INT NOT NULL DEFAULT 0 CHECK (stock_quantity >= 0),
  cost             NUMERIC(12,2) NOT NULL CHECK (cost >= 0),
  supplier_id      INT NOT NULL,
  CONSTRAINT fk_product_supplier
    FOREIGN KEY (supplier_id)
    REFERENCES supplier(supplier_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT
);


CREATE TABLE "order" (
  order_id         INT GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
  order_date       TIMESTAMP NOT NULL DEFAULT NOW(),
  status           TEXT NOT NULL CHECK (status IN ('NEW','PAID','SHIPPED','CANCELLED')) DEFAULT
'NEW',
  payment_method   TEXT CHECK (payment_method IN ('CASH','CARD','TRANSFER','OTHER')),
  total_amount     NUMERIC(12,2) NOT NULL DEFAULT 0 CHECK (total_amount >= 0),
  customer_id      INT NOT NULL,
  employee_id      INT,
  CONSTRAINT fk_order_customer
    FOREIGN KEY (customer_id)
    REFERENCES customer(customer_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT,
  CONSTRAINT fk_order_employee
    FOREIGN KEY (employee_id)
    REFERENCES employee(employee_id)
    ON UPDATE CASCADE
    ON DELETE SET NULL
);

CREATE TABLE order_line (
  order_id         INT NOT NULL,
  product_id       INT NOT NULL,
  quantity         INT NOT NULL CHECK (quantity > 0),
  line_price       NUMERIC(12,2) NOT NULL CHECK (line_price >= 0),
  -- Composite PK = (order_id, product_id)
  PRIMARY KEY (order_id, product_id),
  CONSTRAINT fk_ol_order
    FOREIGN KEY (order_id)
    REFERENCES "order"(order_id)
    ON UPDATE CASCADE
    ON DELETE CASCADE,
  CONSTRAINT fk_ol_product
    FOREIGN KEY (product_id)
    REFERENCES product(product_id)
    ON UPDATE CASCADE
    ON DELETE RESTRICT
);
```

2. Fill in the database with data (synthetically generated, scraped from web data or integrated from open data) **(4 pt)***:* . That should be done through a sequence of SQL Insert/Copy statements.

Data Source

All data were **synthetically generated** to simulate a realistic small-business scenario:

- suppliers.csv - list of product suppliers
- products.csv - inventory items linked to suppliers
- customers.csv - customer profiles with addresses
- employees.csv - store staff and hire dates
- orders.csv - customer orders referencing both customers and employees
- order_lines.csv - line-items associating products with orders

Import Method

We populated the schema via a sequence of SQL COPY statements executed in psql.

Example command sequence:

```
SET search_path TO shop;

COPY shop.supplier   (supplier_name, phone_number)
FROM '/Users/abdullahalghabban/Desktop/SQLMidtermReport/suppliers.csv'
WITH (FORMAT csv, HEADER true);

COPY shop.product    (product_name, category, stock_quantity, cost, supplier_id)
FROM '/Users/abdullahalghabban/Desktop/SQLMidtermReport/products.csv'
WITH (FORMAT csv, HEADER true);

COPY shop.customer   (first_name, last_name, email, phone_number, street, city, state,
zip)
FROM '/Users/abdullahalghabban/Desktop/SQLMidtermReport/customers.csv'
WITH (FORMAT csv, HEADER true);

COPY shop.employee   (first_name, last_name, position, hire_date)
```

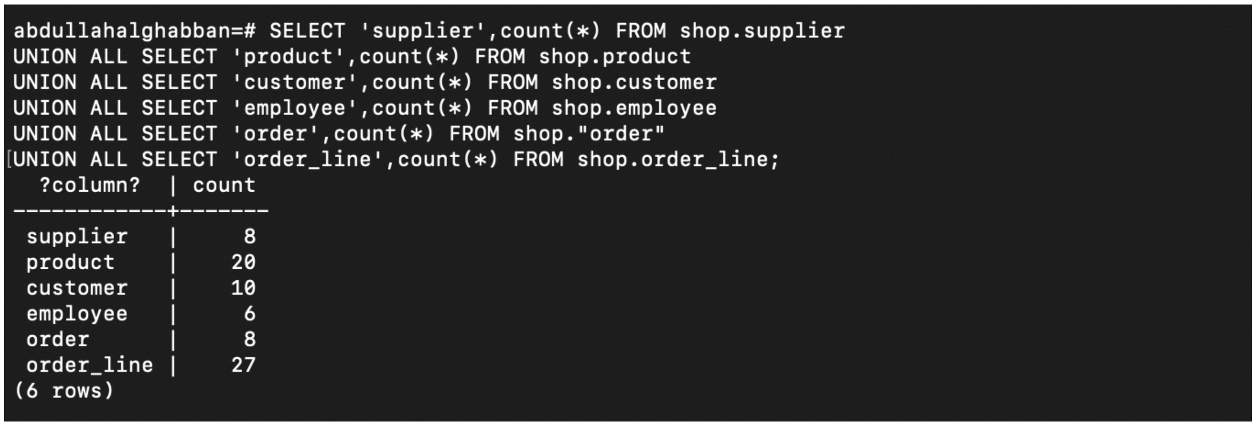
FROM '/Users/abdullahalghabban/Desktop/SQLMidtermReport/employees.csv'
WITH (FORMAT csv, HEADER true);

COPY shop."order"   (order_date, status, payment_method, total_amount, customer_id, employee_id)
FROM '/Users/abdullahalghabban/Desktop/SQLMidtermReport/orders.csv'
WITH (FORMAT csv, HEADER true);

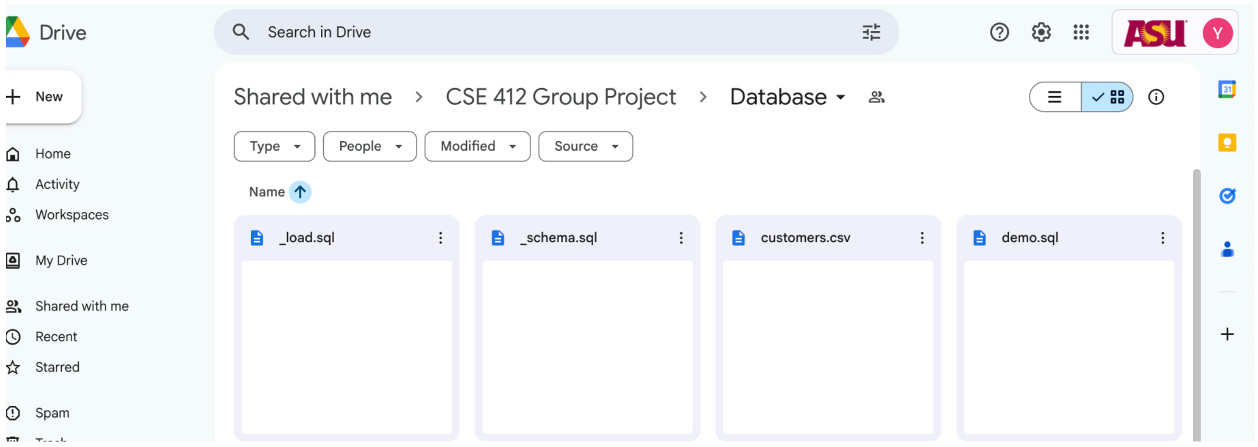
COPY shop.order_line (order_id, product_id, quantity, line_price)
FROM '/Users/abdullahalghabban/Desktop/SQLMidtermReport/order_lines.csv'
WITH (FORMAT csv, HEADER true);

Verification

SELECT 'supplier',count(*) FROM shop.supplier
UNION ALL SELECT 'product',count(*) FROM shop.product
UNION ALL SELECT 'customer',count(*) FROM shop.customer
UNION ALL SELECT 'employee',count(*) FROM shop.employee
UNION ALL SELECT 'order',count(*) FROM shop."order"
UNION ALL SELECT 'order_line',count(*) FROM shop.order_line;

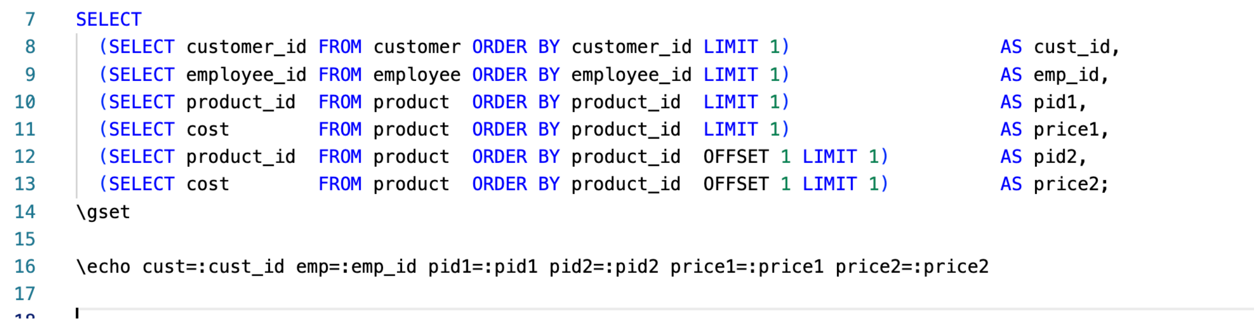

```
abdullahalghabban=# SELECT 'supplier',count(*) FROM shop.supplier
UNION ALL SELECT 'product',count(*) FROM shop.product
UNION ALL SELECT 'customer',count(*) FROM shop.customer
UNION ALL SELECT 'employee',count(*) FROM shop.employee
UNION ALL SELECT 'order',count(*) FROM shop."order"
UNION ALL SELECT 'order_line',count(*) FROM shop.order_line;
  ?column?  | count
------------+-------
 supplier   |     8
 product    |    20
 customer   |    10
 employee   |     6
 order      |     8
 order_line |    27
(6 rows)
```

3.*3. SQL Queries* **(4 pt)**: Prepare examples of SQL queries that cover the application description (as described by the proposal submitted). The SQL queries may include SELECT queries or INSERT/UPDATE/DELETE queries.



HOW TO RUN:
psql -d "$USER" -v ON_ERROR_STOP=1 -f _schema.sql
psql -d "$USER" -v ON_ERROR_STOP=1 -f _load.sql
psql -d "$USER" -v ON_ERROR_STOP=1 -f demo.sql

Code And Explain:

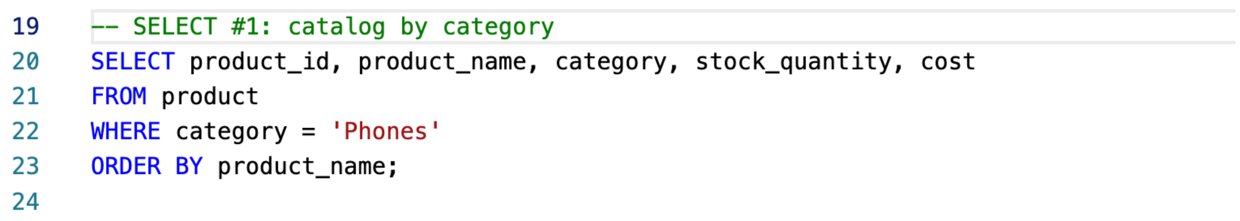


```
SELECT
  (SELECT customer_id FROM customer ORDER BY customer_id LIMIT 1)              AS cust_id,
  (SELECT employee_id FROM employee ORDER BY employee_id LIMIT 1)              AS emp_id,
  (SELECT product_id  FROM product  ORDER BY product_id  LIMIT 1)              AS pid1,
  (SELECT cost        FROM product  ORDER BY product_id  LIMIT 1)              AS price1,
  (SELECT product_id  FROM product  ORDER BY product_id  OFFSET 1 LIMIT 1)     AS pid2,
  (SELECT cost        FROM product  ORDER BY product_id  OFFSET 1 LIMIT 1)     AS price2;
\gset

\echo cust=:cust_id emp=:emp_id pid1=:pid1 pid2=:pid2 price1=:price1 price2=:price2
```

Sets the schema, then runs one SELECT that grabs a sample customer_id, employee_id, and two products with their current cost. \gset saves those columns into psql session variables (:cust_id, :emp_id, :pid1, :price1, :pid2, :price2). \echo prints the variables to confirm they're set.

```
SET
 cust_id | emp_id | pid1 | price1 | pid2 | price2
---------+--------+------+--------+------+--------
       1 |      1 |    1 |  79.00 |    2 | 129.00
(1 row)
cust=1 emp=1 pid1=1 pid2=2 price1=79.00 price2=129.00
```

The query returned one row and \gset created the variables. \echo shows: cust=1, emp=1, pid1=1, pid2=2, price1=79.00, price2=129.00. This means the variables are correctly initialized and ready to be used in later statements (e.g., INSERT ... VALUES (:cust_id, :emp_id, ...)).

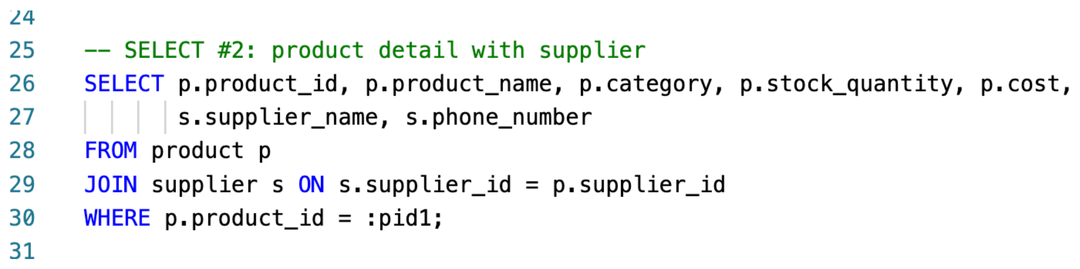


```sql
-- SELECT #1: catalog by category
SELECT product_id, product_name, category, stock_quantity, cost
FROM product
WHERE category = 'Phones'
ORDER BY product_name;
```

```
 product_id |  product_name  | category | stock_quantity |  cost
------------+----------------+----------+----------------+---------
          1 | Acme Phone X   | Phones   |            228 |   79.00
          2 | Acme Phone XL  | Phones   |            107 |  129.00
         13 | Acme Tablet 10 | Phones   |             68 |   19.99
         15 | Stark Keyboard | Phones   |             64 | 1099.00
(4 rows)
```

Selects all products in the 'Phones' category and returns key fields—product_id, name, category, current stock_quantity, and cost—ordered by product_name. This implements the catalog “filter by category + show stock” behavior.

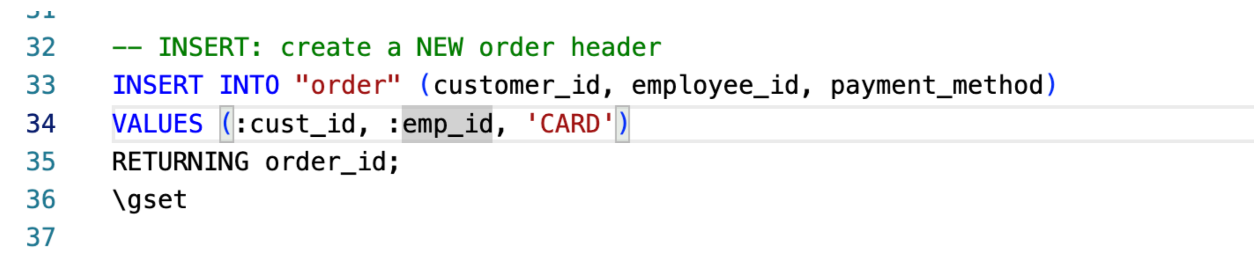
The output lists four phone items (IDs 1, 2, 13, 15) with their on-hand quantities and unit costs—for example, “Acme Phone X” (stock 228, cost 79.00) and “Acme Phone XL” (stock 107, cost 129.00). This confirms the filter works and displays real inventory and pricing for the category.



```sql
-- SELECT #2: product detail with supplier
SELECT p.product_id, p.product_name, p.category, p.stock_quantity, p.cost,
       s.supplier_name, s.phone_number
FROM product p
JOIN supplier s ON s.supplier_id = p.supplier_id
WHERE p.product_id = :pid1;
```

```
 product_id | product_name | category | stock_quantity | cost  |  supplier_name   | phone_number
------------+--------------+----------+----------------+-------+------------------+--------------
          1 | Acme Phone X | Phones   |            228 | 79.00 | Acme Electronics | 555-7001
(1 row)
```

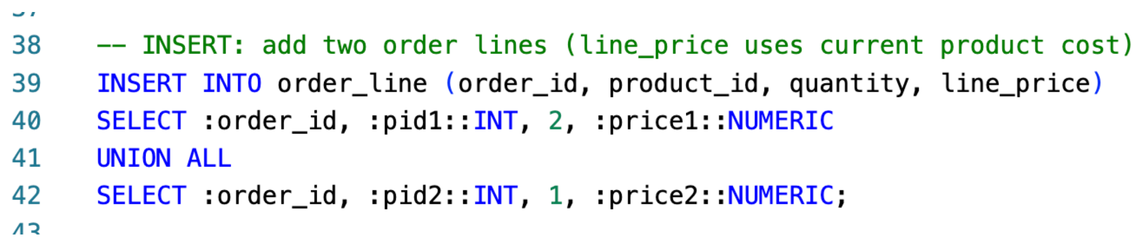
Joins product to its supplier to show a single product’s full detail (id, name, category, stock, cost) plus the supplier’s name and phone. The WHERE p.product_id = :pid1 filter uses the variable you set earlier, so it fetches the chosen product.
The result shows product 1 (“Acme Phone X”) in category “Phones,” with 228 units in stock at cost 79.00, supplied by “Acme Electronics” (phone 555-7001). This confirms the join works and the product-to-supplier relationship is correctly populated.



```
-- INSERT: create a NEW order header
INSERT INTO "order" (customer_id, employee_id, payment_method)
VALUES (:cust_id, :emp_id, 'CARD')
RETURNING order_id;
\gset
```

```
 order_id
----------
        9
(1 row)
INSERT 0 1
INSERT 0 1
```

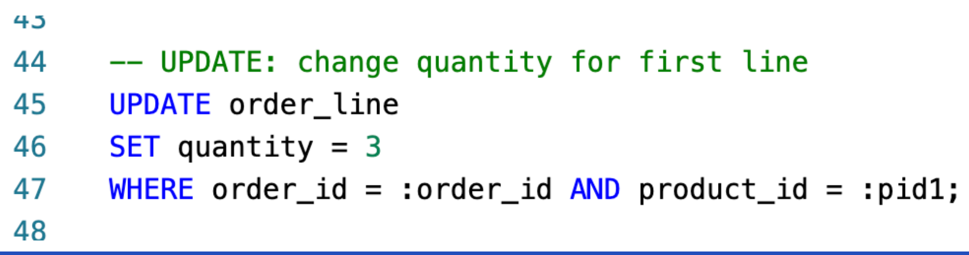
Creates a new order header using the previously set variables :cust_id and :emp_id, with payment_method = 'CARD'. The RETURNING order_id clause outputs the newly created ID, and \gset stores it into the psql variable :order_id for later statements.
The output shows one row returned with order_id = 9, and the INSERT 0 1 lines confirm exactly one row was inserted. Your session now has :order_id set to 9, ready for adding order lines and subsequent updates.



```
-- INSERT: add two order lines (line_price uses current product cost)
INSERT INTO order_line (order_id, product_id, quantity, line_price)
SELECT :order_id, :pid1::INT, 2, :price1::NUMERIC
UNION ALL
SELECT :order_id, :pid2::INT, 1, :price2::NUMERIC;
```

```
INSERT 0 2
```

Inserts two order-line rows for the current :order_id: one for :pid1 with quantity 2 and unit price :price1, and one for :pid2 with quantity 1 and unit price :price2. The two SELECT statements are combined with UNION ALL, so both rows are inserted in a single command.The message INSERT 0 2 confirms that exactly two rows were inserted into order_line, meaning both line items were added successfully for this order.

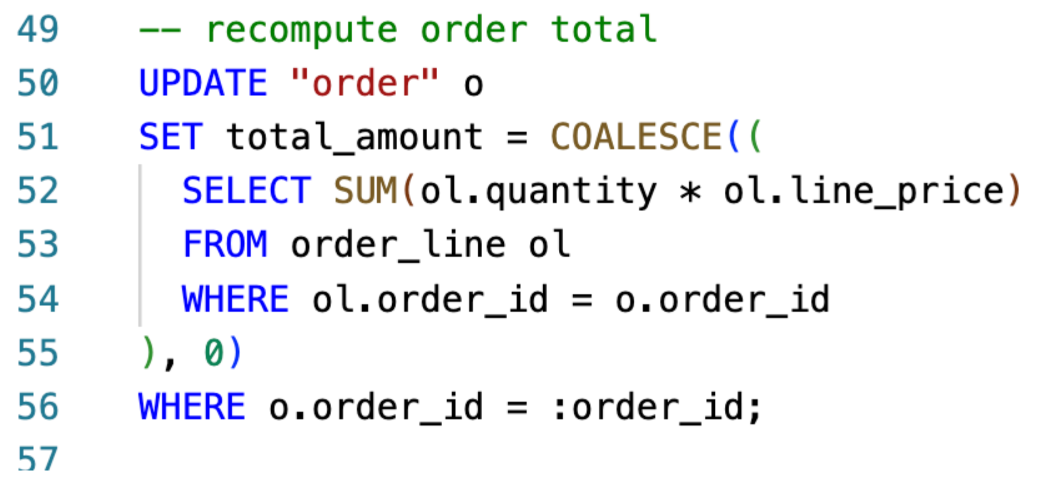


```
-- UPDATE: change quantity for first line
UPDATE order_line
SET quantity = 3
WHERE order_id = :order_id AND product_id = :pid1;
```

```
UPDATE 1
```

Updates the existing line item for the current :order_id and product :pid1, setting its quantity to 3. This simulates a cart edit where the user increases the quantity of the first product.

UPDATE 1 means exactly one row matched the filter and was modified, so the quantity change took effect for that line item.

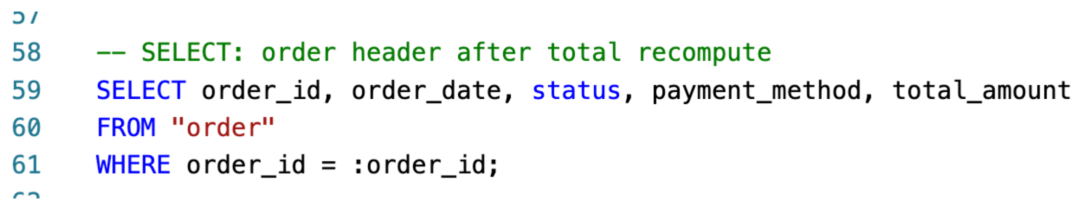


```sql
-- recompute order total
UPDATE "order" o
SET total_amount = COALESCE((
    SELECT SUM(ol.quantity * ol.line_price)
    FROM order_line ol
    WHERE ol.order_id = o.order_id
), 0)
WHERE o.order_id = :order_id;
```

```
UPDATE 1
```

Recalculates the order header's total_amount as the sum of quantity * line_price across all its order_line rows, using COALESCE(..., 0) to set it to zero if there are no lines. The filter WHERE o.order_id = :order_id ensures only the current order is updated.

UPDATE 1 confirms exactly one order row was updated, meaning the header total now reflects the latest line quantities and prices (including the change you just made to set the first line's quantity to 3).



```sql
-- SELECT: order header after total recompute
SELECT order_id, order_date, status, payment_method, total_amount
FROM "order"
WHERE order_id = :order_id;
```

```
 order_id |        order_date          | status | payment_method | total_amount
----------+----------------------------+--------+----------------+--------------
       10 | 2025-10-25 13:49:04.543866 | NEW    | CARD           |       366.00
(1 row)
```

Selects the single order header by :order_id to verify the recalculated fields—showing the timestamp, current status, payment_method, and the updated total_amount after the line edits.

It returns order 10, still in NEW status with payment_method = CARD, and total_amount = 366.00, which matches 3 × 79.00 + 1 × 129.00. This confirms the recompute worked and the header now reflects the cart change.

```sql
-- SELECT: order lines with line totals
SELECT ol.order_id, ol.product_id, p.product_name, ol.quantity, ol.line_price,
       (ol.quantity * ol.line_price) AS line_total
FROM order_line ol
JOIN product p ON p.product_id = ol.product_id
WHERE ol.order_id = :order_id
ORDER BY p.product_name;
```

```
 order_id | product_id | product_name  | quantity | line_price | line_total
----------+------------+---------------+----------+------------+------------
       10 |          1 | Acme Phone X  |        3 |      79.00 |     237.00
       10 |          2 | Acme Phone XL |        1 |     129.00 |     129.00
(2 rows)
```

Lists all line items for the current order by joining order_line with product to show the product name, quantity, unit price, and a computed line_total = quantity * line_price, ordered alphabetically by product name.

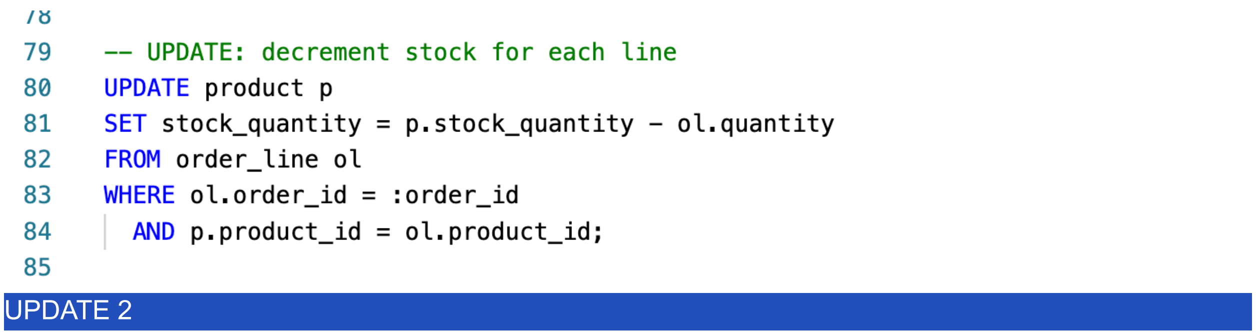
The result shows two items for order 10: “Acme Phone X” (qty 3 × 79.00 = 237.00) and “Acme Phone XL” (qty 1 × 129.00 = 129.00). The numbers add up to the header total 366.00, confirming line totals are correct.

```sql
-- SELECT #3: stock check for this order
SELECT ol.product_id, p.product_name,
       p.stock_quantity AS in_stock, ol.quantity AS needed,
       (p.stock_quantity - ol.quantity) AS remaining
FROM order_line ol
JOIN product p ON p.product_id = ol.product_id
WHERE ol.order_id = :order_id;
```

```
 product_id | product_name  | in_stock | needed | remaining
------------+---------------+----------+--------+-----------
          1 | Acme Phone X  |      228 |      3 |       225
          2 | Acme Phone XL |      107 |      1 |       106
(2 rows)
```

Selects a per-line stock snapshot for the current order by joining order_line with product. For each product it reports current on-hand units (in_stock), the order quantity (needed), and a computed remaining = in_stock - needed, so you can validate availability before payment or fulfillment.

Both items have sufficient inventory: “Acme Phone X” 228 in stock (need 3 → remaining 225) and “Acme Phone XL” 107 in stock (need 1 → remaining 106). Non-negative remaining confirms no stock shortages for this order.

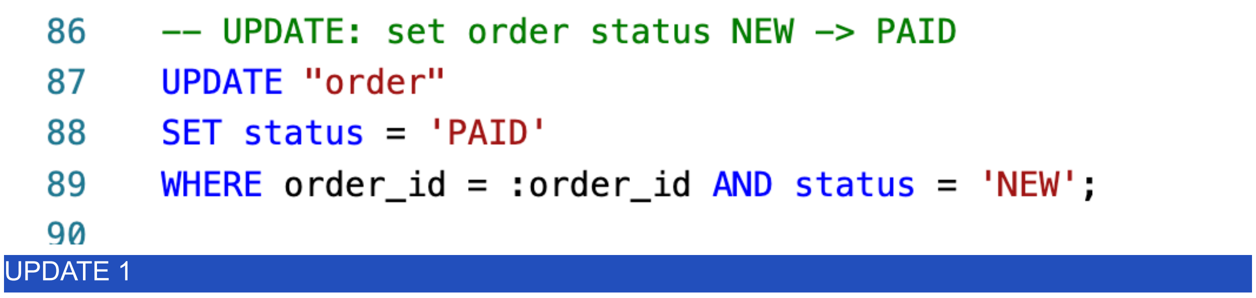


```sql
-- UPDATE: decrement stock for each line
UPDATE product p
SET stock_quantity = p.stock_quantity - ol.quantity
FROM order_line ol
WHERE ol.order_id = :order_id
  AND p.product_id = ol.product_id;
```

UPDATE 2

Updates inventory in one set-based statement by joining product to order_line for the current :order_id, subtracting each line’s quantity from the matching product’s stock_quantity. The p.product_id = ol.product_id predicate ensures only items in this order are affected.

UPDATE 2 means two product rows were updated—both items in the order. Based on the prior stock check, the counts should now be reduced from 228→225 for “Acme Phone X” and 107→106 for “Acme Phone XL,” matching the ordered quantities.



```sql
-- UPDATE: set order status NEW -> PAID
UPDATE "order"
SET status = 'PAID'
WHERE order_id = :order_id AND status = 'NEW';
```

UPDATE 1

Updates the order’s status from NEW to PAID for the current :order_id, enforcing that only orders still in the NEW state are moved to PAID (prevents re-paying an already-paid/cancelled order).

UPDATE 1 indicates one order row matched the condition and was changed — the order is now marked PAID, which, together with the earlier stock decrements, represents a completed checkout for that order.

```sql
-- SELECT: verify header is PAID and show updated product stocks
SELECT order_id, status, total_amount
FROM "order"
WHERE order_id = :order_id;

SELECT p.product_id, p.product_name, p.stock_quantity
FROM product p
WHERE p.product_id IN (:pid1, :pid2)
ORDER BY p.product_id;
```

```
 order_id | status | total_amount
----------+--------+--------------
       10 | PAID   |       366.00
(1 row)
 product_id | product_name  | stock_quantity
------------+---------------+----------------
          1 | Acme Phone X  |            225
          2 | Acme Phone XL |            106
(2 rows)
```

The first query retrieves the order header (order_id, status, total_amount) for the current :order_id to confirm the order's state and amount; the second query selects product_id, product_name, and current stock_quantity for the two products in the order to verify inventory levels after checkout.

The output shows order 10 is now PAID with a total of 366.00, and the two products' stocks have been reduced to 225 and 106 respectively, confirming the checkout updated both the order status and the product inventory.

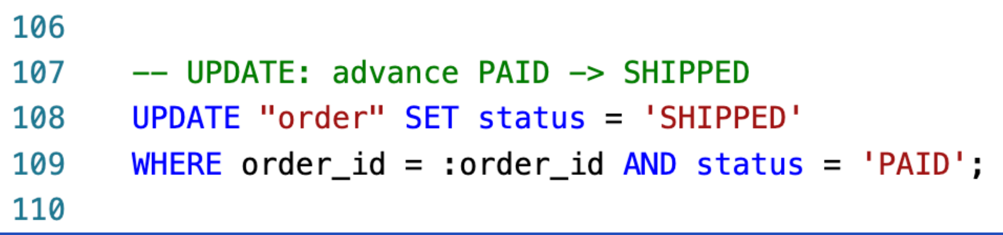
```sql
-- SELECT #4: "My Orders" for the chosen employee
SELECT order_id, order_date, status, total_amount
FROM "order"
WHERE employee_id = :emp_id
ORDER BY order_date DESC;
```

```
 order_id |         order_date         |  status   | total_amount
----------+----------------------------+-----------+--------------
       10 | 2025-10-25 13:49:04.543866 | PAID      |       366.00
        9 | 2025-10-25 13:49:04.537665 | NEW       |         0.00
        3 | 2025-09-23 12:00:00        | PAID      |         0.00
        2 | 2025-09-22 12:00:00        | CANCELLED |         0.00
```

```
(4 rows)
```

Selects all orders assigned to the chosen employee (:emp_id), returning order id, date, status, and total amount, ordered newest first — useful for an employee’s “My Orders” view to see work queue and recent activity.

The result shows four orders for that employee (most recent first): order 10 (PAID, 366.00), order 9 (NEW, 0.00), and two older orders (one PAID, one CANCELLED). This confirms the employee has both completed and pending orders to process.

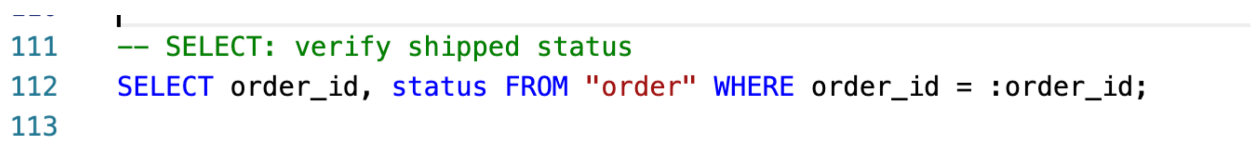


```sql

-- UPDATE: advance PAID -> SHIPPED
UPDATE "order" SET status = 'SHIPPED'
WHERE order_id = :order_id AND status = 'PAID';

```

```
UPDATE 1
```

Updates the order’s status from PAID to SHIPPED for the current :order_id, enforcing that only orders already marked PAID are advanced to the shipping stage.

UPDATE 1 means one order row matched and was changed — the order has been transitioned to SHIPPED and is now in the fulfillment/shipping stage.

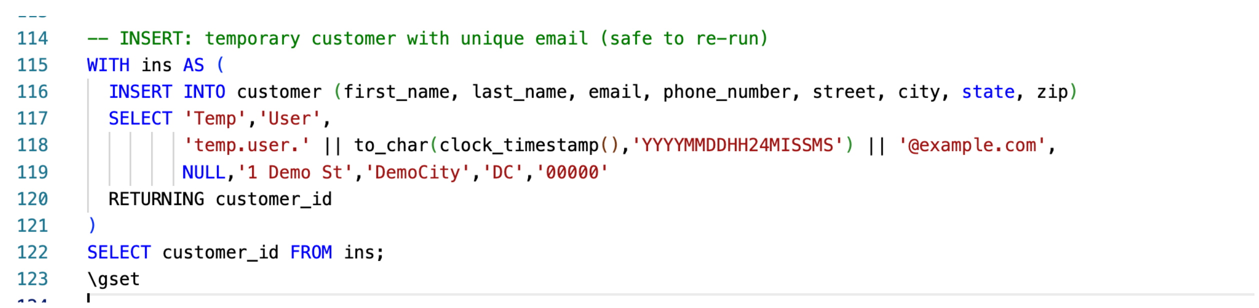


```sql
-- SELECT: verify shipped status
SELECT order_id, status FROM "order" WHERE order_id = :order_id;

```

```
 order_id | status
----------+---------
       10 | SHIPPED
(1 row)
```

Updates the order row to fetch and display its current status for the given :order_id, used to confirm the latest lifecycle state after a status transition.

The query returns one row showing order_id = 10 with status = SHIPPED, indicating the order successfully moved from PAID to the shipping stage.

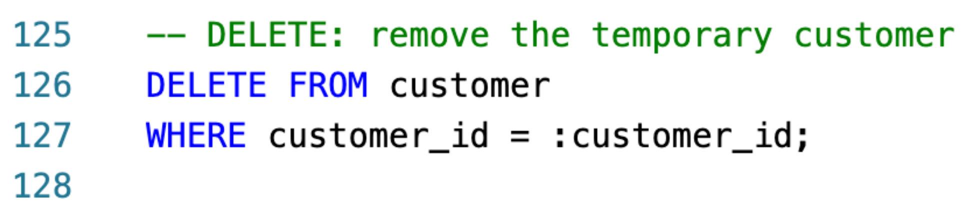


```
-- INSERT: temporary customer with unique email (safe to re-run)
WITH ins AS (
  INSERT INTO customer (first_name, last_name, email, phone_number, street, city, state, zip)
  SELECT 'Temp','User',
         'temp.user.' || to_char(clock_timestamp(),'YYYYMMDDHH24MISSMS') || '@example.com',
         NULL,'1 Demo St','DemoCity','DC','00000'
  RETURNING customer_id
)
SELECT customer_id FROM ins;
\gset
```

```
 customer_id
-------------
          11
(1 row)
```

Inserts a temporary customer with a timestamped, unique email (so repeated runs won't violate the email uniqueness constraint) using a WITH ... RETURNING CTE, and then selects the new customer_id; the following \gset stores that returned customer_id into the psql variable :customer_id for immediate use.

The output shows the new customer_id = 11 (one row returned) and confirms :customer_id is set — you can now run DELETE FROM customer WHERE customer_id = :customer_id; to remove this temporary record.

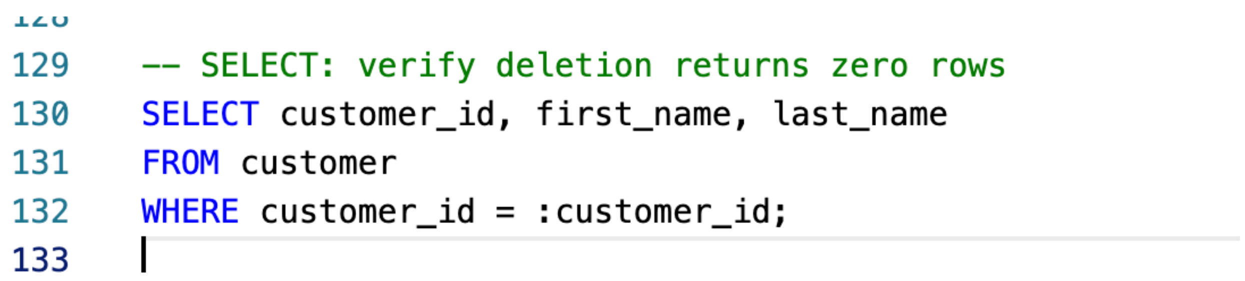


```
-- DELETE: remove the temporary customer
DELETE FROM customer
WHERE customer_id = :customer_id;
```

```
DELETE 1
```

Deletes the temporary customer row whose customer_id matches the :customer_id psql variable, removing the demo record from the customer table.

The output DELETE 1 confirms exactly one row was removed, so the temporary customer (the id returned earlier) was successfully deleted.



```sql
-- SELECT: verify deletion returns zero rows
SELECT customer_id, first_name, last_name
FROM customer
WHERE customer_id = :customer_id;
```

```
 customer_id | first_name | last_name
-------------+------------+-----------
(0 rows)
```

Selects the customer row matching the temporary :customer_id to confirm whether the deletion succeeded; this query returns any remaining record fields (customer_id, first_name, last_name) for that id.

The result shows (0 rows), meaning no record exists with that customer_id — the temporary customer was successfully removed.