🕵️💏 The SQL Sentry: Tracking Anomalies with Database Logic

Technical Guide for Kiosk Operators (and Sleep-Deprived Developers)

In the **Scary Shawarma Kiosk**, your eyes can deceive you. Cameras glitch, shadows move, and some customers seem… statistically improbable. To survive the night shift, you must stop thinking like a cook and start thinking like a **Database Administrator**.

As Django developers know: a clean, structured database is the difference between stable deployment and complete system failure. In this kiosk, that “system failure” just happens to be **Status Erasure**.

This guide walks through a full SQL workflow that turns chaotic observations into structured anomaly detection — using joins, case logic, and a single diagnostic query.

🧑‍🌾 1. Initializing the Security Schema

First, we create a database designed to cross-reference reality from three perspectives:

- **customers** → what the operator sees
- **cctv_logs** → what cameras record
- **biometrics** → hidden physiological truth

```
-- Step 1: Create the Database
CREATE DATABASE shawarma_shop;
USE shawarma_shop;

-- Step 2: Create the Tables
CREATE TABLE customers (
    customer_id INTEGER PRIMARY KEY,
    name VARCHAR(50),
    reported_gender VARCHAR(10),
    appearance_notes TEXT
);

CREATE TABLE cctv_logs (
    log_id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    cam_1_front VARCHAR(50),
    cam_2_back VARCHAR(50),
    night_vision VARCHAR(20),
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

CREATE TABLE biometrics (
    customer_id INTEGER,
    voice_pitch VARCHAR(20),
    has_pulse BOOLEAN,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

🏋🏽‍♀️ 2. Populating the Night Shift Logs

Around 2:00 AM, patterns begin to break. The following entries illustrate normal vs anomalous behavior.

```
-- Step 3: Insert the Suspects
INSERT INTO customers VALUES
(1, 'Karate Man', 'Male', 'Practicing moves, looks normal'),
(2, 'Smiling Man', 'Male', 'Wide grin, staring intensely'),
(3, 'Business Lady', 'Female', 'Looking at watch, seems impatient'),
(4, 'The Twin', 'Female', 'Looks like a regular student');

-- Step 4: Record the CCTV Truth
INSERT INTO cctv_logs VALUES
(101, 1, 'Humanoid', 'Normal Back', 'Visible'),
(102, 2, 'Two People', 'Normal Back', 'Visible'),
(103, 3, 'Humanoid', 'Black Holes', 'Disappears'),
(104, 4, 'Humanoid', 'Normal Back', 'Visible');

-- Step 5: Log the Biometric Reality
INSERT INTO biometrics VALUES
(1, 'Normal', 1),
(2, 'High-Pitched', 1),
(3, 'None', 0),
(4, 'Ultra-Deep Bass', 1);
```

🧏🏾‍♀️ 3. The One-Command Survival Query

In production environments — and haunted kiosks — you don’t have time to run multiple checks. You need **one command** that correlates every signal and tells you whether to proceed.

The Master Diagnostic Command

```
SELECT
    c.customer_id,
    c.name,
    c.appearance_notes,
    CASE
        WHEN b.has_pulse = 0 THEN 'NO HEARTBEAT (CLASS-S)'
        WHEN l.night_vision = 'Disappears' THEN 'VANISHING ENTITY'
        WHEN l.cam_1_front = 'Two People' THEN 'BODY DOUBLE DETECTED'
        WHEN l.cam_2_back = 'Black Holes' THEN 'BACK MUTATION'
        WHEN c.reported_gender = 'Female'
             AND b.voice_pitch LIKE '%Deep%' THEN 'VOICE-GENDER MISMATCH'
        ELSE 'LIKELY HUMAN'
    END AS anomaly_type
FROM customers c
JOIN cctv_logs l ON c.customer_id = l.customer_id
JOIN biometrics b ON c.customer_id = b.customer_id
WHERE l.night_vision != 'Visible'
   OR l.cam_1_front != 'Humanoid'
   OR l.cam_2_back != 'Normal Back'
   OR b.has_pulse = 0
   OR (c.reported_gender = 'Female' AND b.voice_pitch LIKE '%Deep%');
```

Why this works

- **JOINs** unify multiple perception layers.
- **CASE** maps raw anomalies into human-readable labels.
- **WHERE** filters out normal customers, reducing cognitive load under pressure.

This turns noisy surveillance data into a clean operational decision.

👨🏻 4. Database State: The Raw Evidence

Before running diagnostics, always inspect raw tables to verify data integrity.

Table: `customers`

```
SELECT * FROM customers;
```

customer_id	name	reported_gender	appearance_notes
1	Karate Man	Male	Practicing moves, looks normal

customer_id	name	reported_gender	appearance_notes
2	Smiling Man	Male	Wide grin, staring intensely
3	Business Lady	Female	Looking at watch, seems impatient
4	The Twin	Female	Looks like a regular student

Table: `cctv_logs`

```sql
SELECT * FROM cctv_logs;
```

log_id	customer_id	cam_1_front	cam_2_back	night_vision
101	1	Humanoid	Normal Back	Visible
102	2	**Two People**	Normal Back	Visible
103	3	Humanoid	**Black Holes**	**Disappears**
104	4	Humanoid	Normal Back	Visible

Table: `biometrics`

```sql
SELECT * FROM biometrics;
```

customer_id	voice_pitch	has_pulse
1	Normal	1
2	High-Pitched	1
3	None	**0**
4	**Ultra-Deep Bass**	1

👦🏽 5. Final Diagnostic Output

Executing the master query produces the following actionable report:

customer_id	name	appearance_notes	anomaly_type
2	Smiling Man	Wide grin, staring intensely	**BODY DOUBLE DETECTED**
3	Business Lady	Looking at watch, seems impatient	**NO HEARTBEAT (CLASS-S)**
4	The Twin	Looks like a regular student	**VOICE-GENDER MISMATCH**

This report is intentionally minimal: just enough context to make a decision quickly.

👧🏻 6. Lessons for Real-World Developers

Behind the horror theme are practical database design principles:

- Model multiple sources of truth separately.
- Normalize observational data before analysis.
- Use SQL logic to transform raw signals into business decisions.
- Keep output concise for operational environments.

Whether you're running a kiosk, a security dashboard, or a Django SaaS platform, structured queries reduce uncertainty.

👨🏾‍🍼 Conclusion

Data integrity is survival.

When `biometrics.has_pulse` doesn't align with `customers.name`, you're not just debugging — you're making a safety decision.

Keep your schema clean, your joins indexed, and your CASE statements ready.

Because sometimes... the anomaly isn't in the code.

🏊‍♀️ Next Steps (Optional Enhancements)

If you want to evolve this system further, consider:

- Creating a stored procedure: `CALL Check_Customer(id);`
- Building a Python script that prints **SERVE** or **REJECT** automatically
- Streaming logs into a Django admin dashboard
- Adding triggers for real-time alerts

Stay safe. Query everything.