

SQL Detective: Answer Key and Solution Guide

CONFIDENTIAL - ADMIN ACCESS ONLY

Document Type: Internal Reference

Classification: Private / Admin Only

Version: 1.0

Last Updated: January 2026

Maintainer: Development Team

WARNING: This document contains all correct solutions for the SQL Detective game. Do not share with players or expose through any public endpoint. For internal development, debugging, and interview demonstration only.

Table of Contents

1. Overview
 2. Level-wise Solution Index
 3. Level 1: The Missing Witness
 4. Level 2: The Midnight Call
 5. Level 3: The Connection
 6. Level 4: The Pattern
 7. Level 5: The Money Trail
 8. Level 6: The Movement
 9. Level 7: The Final Piece
 10. Query Validation Logic
 11. Security Considerations
 12. Admin Usage Guidelines
 13. Interview Demonstration Tips
-

1. Overview

Purpose

This document serves as the authoritative answer key for the SQL Detective game. It provides:

- Official correct queries for each level
- Alternative valid solutions

- Common incorrect attempts with explanations
- Edge case handling guidelines
- Validation logic reference

Integration with Game Logic

The solutions in this document correspond directly to the expected queries defined in:

`backend/levels/level_config.py`

The `LevelChecker` service in `backend/services/level_checker.py` executes both the player query and the expected query, then compares result sets.

Usage Guidelines

Use Case	Permitted
Debugging incorrect unlocks	Yes
Adding new levels	Yes
Interview demonstrations	Yes
Sharing with players	No
Public documentation	No
Frontend exposure	No

2. Level-wise Solution Index

Level	Title	SQL Concepts	Tables Used
1	The Missing Witness	SELECT, WHERE, AND	suspects
2	The Midnight Call	BETWEEN, ORDER BY, LIMIT	phone_records
3	The Connection	INNER JOIN	suspects, cctv_logs, locations
4	The Pattern	GROUP BY, HAVING, COUNT	phone_records
5	The Money Trail	Subqueries, AVG	bank_transactions
6	The Movement	CTEs (WITH clause)	cctv_logs, locations
7	The Final Piece	Window Functions	bank_transactions, suspects

3. Level 1: The Missing Witness

A. Level Summary

Objective: Find all suspects over 30 years old with prior criminal records.

SQL Concepts Evaluated: - Basic SELECT statement - WHERE clause filtering - AND logical operator - Numeric and boolean comparisons

B. Expected Output

Description: Returns suspect records where age exceeds 30 and criminal_record equals 1.

Expected Row Count: 2 rows

Expected Records: - Viktor Petrov (age 42, criminal_record = 1) - James Wilson (age 45, criminal_record = 1)

C. Primary Correct SQL Solution

```
SELECT * FROM suspects  
WHERE age > 30 AND criminal_record = 1;
```

D. Alternate Valid Solutions

Alternative 1: Using explicit column selection

```
SELECT id, name, age, occupation, criminal_record  
FROM suspects  
WHERE age > 30 AND criminal_record = 1;
```

Valid because: Returns the same logical result set with specified columns.

Alternative 2: Using greater-than-or-equal with different threshold

```
SELECT * FROM suspects  
WHERE age >= 31 AND criminal_record = 1;
```

Valid because: Produces identical results given the data distribution.

E. Common Incorrect Queries

Incorrect 1: Missing AND operator

```
SELECT * FROM suspects WHERE age > 30;
```

Failure reason: Returns all suspects over 30 regardless of criminal record, including those without prior convictions.

Incorrect 2: Wrong comparison operator for criminal_record

```
SELECT * FROM suspects WHERE age > 30 AND criminal_record > 0;
```

Failure reason: While logically equivalent for this dataset, using > 0 instead of = 1 may be flagged depending on validation strictness.

Incorrect 3: Using OR instead of AND

```
SELECT * FROM suspects WHERE age > 30 OR criminal_record = 1;
```

Failure reason: Returns suspects meeting either condition, not both. Includes younger criminals and older non-criminals.

F. Edge Cases

Edge Case	Handling
Row order difference	Ignored (order_matters = False)
NULL values	No NULLs in age or criminal_record columns
Duplicate rows	Not possible with primary key

4. Level 2: The Midnight Call

A. Level Summary

Objective: Find the 5 most recent phone calls made between 11 PM (March 15) and 2 AM (March 16, 2024).

SQL Concepts Evaluated: - BETWEEN operator for range filtering - ORDER BY for sorting - DESC for descending order - LIMIT for result restriction

B. Expected Output

Description: Returns phone records within the crime time window, sorted by timestamp descending, limited to 5 rows.

Expected Row Count: 5 rows

C. Primary Correct SQL Solution

```
SELECT * FROM phone_records
WHERE timestamp BETWEEN '2024-03-15 23:00:00' AND '2024-03-16 02:00:00'
ORDER BY timestamp DESC
LIMIT 5;
```

D. Alternate Valid Solutions

Alternative 1: Using comparison operators

```
SELECT * FROM phone_records
WHERE timestamp >= '2024-03-15 23:00:00'
  AND timestamp <= '2024-03-16 02:00:00'
ORDER BY timestamp DESC
LIMIT 5;
```

Valid because: Equivalent to BETWEEN with explicit operators.

Alternative 2: Different column selection

```
SELECT id, caller_id, receiver_id, timestamp, duration
FROM phone_records
```

```

WHERE timestamp BETWEEN '2024-03-15 23:00:00' AND '2024-03-16 02:00:00'
ORDER BY timestamp DESC
LIMIT 5;

```

Valid because: Same logical result with explicit columns.

E. Common Incorrect Queries

Incorrect 1: Missing ORDER BY

```

SELECT * FROM phone_records
WHERE timestamp BETWEEN '2024-03-15 23:00:00' AND '2024-03-16 02:00:00'
LIMIT 5;

```

Failure reason: Returns arbitrary 5 rows, not the most recent ones.

Incorrect 2: Ascending order instead of descending

```

SELECT * FROM phone_records
WHERE timestamp BETWEEN '2024-03-15 23:00:00' AND '2024-03-16 02:00:00'
ORDER BY timestamp ASC
LIMIT 5;

```

Failure reason: Returns the oldest 5 calls, not the most recent.

Incorrect 3: Wrong time range

```

SELECT * FROM phone_records
WHERE timestamp BETWEEN '2024-03-15 11:00:00' AND '2024-03-16 02:00:00'
ORDER BY timestamp DESC
LIMIT 5;

```

Failure reason: Uses 11 AM instead of 11 PM (23:00), capturing wrong time window.

F. Edge Cases

Edge Case	Handling
Row order difference	Matters (order_matters = True)
Exact boundary times	BETWEEN is inclusive
Missing LIMIT	Returns all matching rows

5. Level 3: The Connection

A. Level Summary

Objective: Find suspects captured on CCTV at the Downtown Bank location. Show name, occupation, location, and timestamp.

SQL Concepts Evaluated: - INNER JOIN across multiple tables - Foreign key relationships - Table aliases - Column aliasing with AS

B. Expected Output

Description: Returns suspects seen at Downtown Bank with their details and sighting timestamps.

Expected Row Count: 5 rows

Required Columns: name, occupation, location, timestamp

C. Primary Correct SQL Solution

```
SELECT s.name, s.occupation, l.name AS location, c.timestamp
FROM suspects s
INNER JOIN cctv_logs c ON s.id = c.person_id
INNER JOIN locations l ON c.location_id = l.id
WHERE l.name = 'Downtown Bank';
```

D. Alternate Valid Solutions

Alternative 1: Using JOIN without INNER keyword

```
SELECT s.name, s.occupation, l.name AS location, c.timestamp
FROM suspects s
JOIN cctv_logs c ON s.id = c.person_id
JOIN locations l ON c.location_id = l.id
WHERE l.name = 'Downtown Bank';
```

Valid because: JOIN defaults to INNER JOIN in SQL.

Alternative 2: Different alias names

```
SELECT suspects.name, suspects.occupation, locations.name AS location, cctv_logs.timestamp
FROM suspects
INNER JOIN cctv_logs ON suspects.id = cctv_logs.person_id
INNER JOIN locations ON cctv_logs.location_id = locations.id
WHERE locations.name = 'Downtown Bank';
```

Valid because: Full table names produce same result.

E. Common Incorrect Queries

Incorrect 1: Missing one JOIN

```
SELECT s.name, s.occupation, c.timestamp
FROM suspects s
INNER JOIN cctv_logs c ON s.id = c.person_id
WHERE c.location_id = 1;
```

Failure reason: Missing location name column; uses hardcoded ID instead of name filter.

Incorrect 2: Wrong JOIN condition

```
SELECT s.name, s.occupation, l.name AS location, c.timestamp
FROM suspects s
INNER JOIN cctv_logs c ON s.id = c.location_id
INNER JOIN locations l ON c.location_id = l.id
WHERE l.name = 'Downtown Bank';
```

Failure reason: Joining suspects to cctv_logs on wrong column (location_id instead of person_id).

Incorrect 3: Using LEFT JOIN when only INNER needed

```
SELECT s.name, s.occupation, l.name AS location, c.timestamp
FROM suspects s
LEFT JOIN cctv_logs c ON s.id = c.person_id
LEFT JOIN locations l ON c.location_id = l.id
WHERE l.name = 'Downtown Bank';
```

Failure reason: While this may work, it could include NULL matches not desired for this level.

F. Edge Cases

Edge Case	Handling
Row order difference	Ignored (order_matters = False)
Column alias differences	Must match expected columns
Multiple sightings per suspect	All sightings returned

6. Level 4: The Pattern

A. Level Summary

Objective: Find callers who made more than 5 calls on March 15, 2024.

SQL Concepts Evaluated: - GROUP BY clause - HAVING for group filtering
- COUNT aggregate function - DATE() function for date extraction

B. Expected Output

Description: Returns caller IDs and their call counts for those exceeding 5 calls.

Expected Row Count: 1 row

Expected Record: - caller_id: 3 (Viktor Petrov)

C. Primary Correct SQL Solution

```
SELECT caller_id, COUNT(*) AS call_count
FROM phone_records
WHERE DATE(timestamp) = '2024-03-15'
GROUP BY caller_id
HAVING COUNT(*) > 5;
```

D. Alternate Valid Solutions

Alternative 1: Using timestamp range instead of DATE()

```
SELECT caller_id, COUNT(*) AS call_count
FROM phone_records
WHERE timestamp >= '2024-03-15 00:00:00'
    AND timestamp < '2024-03-16 00:00:00'
GROUP BY caller_id
HAVING COUNT(*) > 5;
```

Valid because: Equivalent date filtering with explicit range.

Alternative 2: Different column alias

```
SELECT caller_id, COUNT(*) AS total_calls
FROM phone_records
WHERE DATE(timestamp) = '2024-03-15'
GROUP BY caller_id
HAVING COUNT(*) > 5;
```

Valid because: Alias name does not affect result set comparison.

E. Common Incorrect Queries

Incorrect 1: Using WHERE instead of HAVING

```
SELECT caller_id, COUNT(*) AS call_count
FROM phone_records
WHERE DATE(timestamp) = '2024-03-15' AND COUNT(*) > 5
GROUP BY caller_id;
```

Failure reason: Cannot use aggregate functions in WHERE clause; syntax error.

Incorrect 2: Missing GROUP BY

```
SELECT caller_id, COUNT(*) AS call_count
FROM phone_records
WHERE DATE(timestamp) = '2024-03-15'
HAVING COUNT(*) > 5;
```

Failure reason: Aggregation without GROUP BY returns single row with total count.

Incorrect 3: Wrong date

```
SELECT caller_id, COUNT(*) AS call_count
FROM phone_records
WHERE DATE(timestamp) = '2024-03-16'
GROUP BY caller_id
HAVING COUNT(*) > 5;
```

Failure reason: Filtering wrong date misses the crime day activity.

F. Edge Cases

Edge Case	Handling
Row order difference	Ignored (order_matters = False)
Callers with exactly 5 calls	Not included (> 5 required)
Alias naming	Ignored in comparison

7. Level 5: The Money Trail

A. Level Summary

Objective: Find transactions where amount exceeds the average for the crime week (March 10-17, 2024).

SQL Concepts Evaluated: - Subqueries in WHERE clause - AVG aggregate function - Comparison with subquery result

B. Expected Output

Description: Returns all transactions above the calculated average amount.

Expected Row Count: 6 rows (approximate, depends on average calculation)

C. Primary Correct SQL Solution

```
SELECT * FROM bank_transactions
WHERE amount > (
    SELECT AVG(amount) FROM bank_transactions
    WHERE timestamp BETWEEN '2024-03-10' AND '2024-03-17 23:59:59'
);
```

D. Alternate Valid Solutions

Alternative 1: Using date range with explicit times

```
SELECT * FROM bank_transactions
WHERE amount > (
    SELECT AVG(amount) FROM bank_transactions
    WHERE timestamp >= '2024-03-10 00:00:00'
        AND timestamp <= '2024-03-17 23:59:59'
);
```

Valid because: Equivalent date range specification.

Alternative 2: Using DATE() function

```
SELECT * FROM bank_transactions
WHERE amount > (
    SELECT AVG(amount) FROM bank_transactions
    WHERE DATE(timestamp) BETWEEN '2024-03-10' AND '2024-03-17'
);
```

Valid because: DATE extraction achieves same filtering.

E. Common Incorrect Queries

Incorrect 1: Hardcoded average value

```
SELECT * FROM bank_transactions
WHERE amount > 5000;
```

Failure reason: Uses assumed average instead of calculated value; not dynamic.

Incorrect 2: Missing date filter in subquery

```
SELECT * FROM bank_transactions
WHERE amount > (
    SELECT AVG(amount) FROM bank_transactions
);
```

Failure reason: Calculates average of all transactions, not just crime week.

Incorrect 3: Using \geq instead of $>$

```
SELECT * FROM bank_transactions
WHERE amount >=
    (SELECT AVG(amount) FROM bank_transactions
    WHERE timestamp BETWEEN '2024-03-10' AND '2024-03-17 23:59:59'
);
```

Failure reason: Includes transactions equal to average; may return additional rows.

F. Edge Cases

Edge Case	Handling
Row order difference	Ignored (order_matters = False)
Floating point comparison	Handled by SQLite
Date boundary inclusion	BETWEEN is inclusive

8. Level 6: The Movement

A. Level Summary

Objective: Create a timeline of suspect #3's movements using CCTV data.

SQL Concepts Evaluated: - Common Table Expressions (CTEs) - WITH clause syntax - JOIN within CTE - Ordering by timestamp

B. Expected Output

Description: Returns chronological list of locations visited by suspect ID 3.

Expected Row Count: 10 rows (all Viktor's CCTV sightings)

Required Columns: timestamp, location

C. Primary Correct SQL Solution

```
WITH suspect_movements AS (
    SELECT c.timestamp, l.name AS location
    FROM cctv_logs c
    JOIN locations l ON c.location_id = l.id
    WHERE c.person_id = 3
)
SELECT * FROM suspect_movements
ORDER BY timestamp;
```

D. Alternate Valid Solutions

Alternative 1: Without CTE (direct query)

```
SELECT c.timestamp, l.name AS location
FROM cctv_logs c
JOIN locations l ON c.location_id = l.id
WHERE c.person_id = 3
ORDER BY timestamp;
```

Valid because: Produces identical result without CTE wrapper. May be accepted depending on validation strictness.

Alternative 2: CTE with different alias

```
WITH movements AS (
    SELECT c.timestamp, l.name AS location
    FROM cctv_logs c
    JOIN locations l ON c.location_id = l.id
    WHERE c.person_id = 3
)
SELECT timestamp, location FROM movements
ORDER BY timestamp;
```

Valid because: CTE name and explicit column selection do not affect result.

E. Common Incorrect Queries

Incorrect 1: Missing ORDER BY

```
WITH suspect_movements AS (
    SELECT c.timestamp, l.name AS location
    FROM cctv_logs c
    JOIN locations l ON c.location_id = l.id
    WHERE c.person_id = 3
)
SELECT * FROM suspect_movements;
```

Failure reason: Timeline requires chronological ordering.

Incorrect 2: Wrong person_id

```
WITH suspect_movements AS (
    SELECT c.timestamp, l.name AS location
    FROM cctv_logs c
    JOIN locations l ON c.location_id = l.id
    WHERE c.person_id = 1
)
SELECT * FROM suspect_movements
ORDER BY timestamp;
```

Failure reason: Tracking wrong suspect (Marcus Chen instead of Viktor Petrov).

Incorrect 3: Missing JOIN

```
WITH suspect_movements AS (
    SELECT timestamp, location_id
    FROM cctv_logs
    WHERE person_id = 3
)
```

```
SELECT * FROM suspect_movements
ORDER BY timestamp;
```

Failure reason: Returns location IDs instead of location names.

F. Edge Cases

Edge Case	Handling
Row order difference	Matters (order_matters = True)
CTE naming	Ignored in comparison
Column alias	Must include location name

9. Level 7: The Final Piece

A. Level Summary

Objective: Analyze transaction patterns for suspects with criminal records using window functions.

SQL Concepts Evaluated: - RANK() window function - LAG() for previous row access - PARTITION BY clause - OVER() syntax - Subquery in WHERE clause

B. Expected Output

Description: Returns transactions with rankings and comparisons to previous amounts for criminal suspects.

Expected Row Count: 8 rows (transactions from suspects 3, 5, and 10)

Required Columns: account_id, amount, timestamp, amount_rank, prev_amount, amount_change

C. Primary Correct SQL Solution

```
SELECT
    account_id,
    amount,
    timestamp,
    RANK() OVER (PARTITION BY account_id ORDER BY amount DESC) AS amount_rank,
    LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS prev_amount,
    amount - LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS amount_change
FROM bank_transactions
WHERE account_id IN (SELECT id FROM suspects WHERE criminal_record = 1);
```

D. Alternate Valid Solutions

Alternative 1: Using ROW_NUMBER instead of RANK

```
SELECT
    account_id,
    amount,
    timestamp,
    ROW_NUMBER() OVER (PARTITION BY account_id ORDER BY amount DESC) AS amount_rank,
    LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS prev_amount,
    amount - LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS amount_change
FROM bank_transactions
WHERE account_id IN (SELECT id FROM suspects WHERE criminal_record = 1);
```

Valid because: For this dataset, ROW_NUMBER and RANK produce same results (no ties).

Alternative 2: Using explicit subquery

```
SELECT
    account_id,
    amount,
    timestamp,
    RANK() OVER (PARTITION BY account_id ORDER BY amount DESC) AS amount_rank,
    LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS prev_amount,
    amount - LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS amount_change
FROM bank_transactions
WHERE account_id IN (3, 5, 10);
```

Valid because: Hardcoded IDs match the criminal suspects. Not recommended for maintainability.

E. Common Incorrect Queries

Incorrect 1: Missing PARTITION BY

```
SELECT
    account_id,
    amount,
    timestamp,
    RANK() OVER (ORDER BY amount DESC) AS amount_rank,
    LAG(amount) OVER (ORDER BY timestamp) AS prev_amount,
    amount - LAG(amount) OVER (ORDER BY timestamp) AS amount_change
FROM bank_transactions
WHERE account_id IN (SELECT id FROM suspects WHERE criminal_record = 1);
```

Failure reason: Ranks across all accounts instead of per-account partitions.

Incorrect 2: Missing subquery filter

```

SELECT
    account_id,
    amount,
    timestamp,
    RANK() OVER (PARTITION BY account_id ORDER BY amount DESC) AS amount_rank,
    LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS prev_amount,
    amount - LAG(amount) OVER (PARTITION BY account_id ORDER BY timestamp) AS amount_change
FROM bank_transactions;

```

Failure reason: Includes transactions from all accounts, not just criminal suspects.

Incorrect 3: Wrong LAG ordering

```

SELECT
    account_id,
    amount,
    timestamp,
    RANK() OVER (PARTITION BY account_id ORDER BY amount DESC) AS amount_rank,
    LAG(amount) OVER (PARTITION BY account_id ORDER BY amount) AS prev_amount,
    amount - LAG(amount) OVER (PARTITION BY account_id ORDER BY amount) AS amount_change
FROM bank_transactions
WHERE account_id IN (SELECT id FROM suspects WHERE criminal_record = 1);

```

Failure reason: LAG should be ordered by timestamp for temporal comparison, not amount.

F. Edge Cases

Edge Case	Handling
Row order difference	Ignored (order_matters = False)
NULL from first LAG	First row has NULL prev_amount
amount_change NULL	Expected for first row per partition

10. Query Validation Logic

Comparison Approach

The system uses result-based comparison rather than query-string comparison:

1. Execute player's query against database
2. Execute expected query against database
3. Normalize both result sets
4. Compare sets/lists based on order_matters flag

Normalization Process

```
def normalize_rows(rows):
    normalized = []
    for row in rows:
        normalized_row = []
        for val in row:
            if val is None:
                normalized_row.append(None)
            elif isinstance(val, float):
                normalized_row.append(round(val, 2))
            else:
                normalized_row.append(val)
        normalized.append(tuple(normalized_row))
    return normalized
```

Tolerance Rules

Aspect	Tolerance
Row ordering	Configurable per level
Column aliases	Ignored (only values compared)
Floating point	Rounded to 2 decimal places
NULL values	Compared as-is
Whitespace	Stripped before execution

Result Comparison

```
if level.order_matters:
    is_correct = user_data == expected_data
else:
    is_correct = set(user_data) == set(expected_data)
```

11. Security Considerations

Read-Only Enforcement

Database connections use URI mode with read-only flag:

```
sqlite3.connect('file:database.db?mode=ro', uri=True)
```

Blocked Keywords

The following SQL keywords are rejected before execution:

DROP, DELETE, UPDATE, INSERT, ALTER, CREATE, TRUNCATE,

GRANT, REVOKE, EXEC, EXECUTE, PRAGMA, ATTACH, DETACH,
VACUUM, REINDEX, REPLACE, UPSERT, MERGE

SQL Injection Mitigation

Layer	Protection
Validation	Keyword blocking
Statement Type	SELECT/WITH only
Multi-Statement	Semicolon detection
Timeout	5 second limit
Result Limit	1000 rows maximum

Pattern Blocking

Additional patterns blocked:

- INTO OUTFILE
 - INTO DUMPFILE
 - LOAD_FILE
 - BENCHMARK()
 - SLEEP()
-

12. Admin Usage Guidelines

When to Update Solutions

- Database schema changes affecting query results
- New seed data altering expected row counts
- Bug fixes in level configuration
- Validation logic refinements

How to Add New Levels

1. Define level in `backend/levels/level_config.py`
2. Add expected query to Level class instantiation
3. Set `expected_row_count` if fixed
4. Configure `order_matters` flag
5. Add tables to `tables_unlocked` list
6. Update this document with new level section

How to Debug Incorrect Unlocks

1. Check player's query in browser console
2. Execute query manually in SQLite
3. Compare result with expected query result

4. Verify row count matches
5. Check order_matters configuration
6. Review normalization for edge cases

Level Configuration Reference

```
Level(
    level_id=1,
    title="The Missing Witness",
    story="...",
    objective="...",
    hint="...",
    sql_concepts=["SELECT", "WHERE"],
    tables_unlocked=["suspects"],
    expected_query="SELECT * FROM suspects WHERE age > 30 AND criminal_record = 1",
    expected_columns=None,
    expected_row_count=2,
    order_matters=False
)
```

13. Interview Demonstration Tips

How to Explain This Document

“This answer key demonstrates several professional practices:

First, it shows I understand the difference between player-facing and admin-facing documentation. Solutions are kept separate from public materials.

Second, it documents not just correct answers but also common mistakes. This helps with debugging and shows understanding of how students typically fail.

Third, it explains the validation logic so future maintainers understand how answers are checked. This is crucial for extending the system.”

Key Points to Highlight

1. **Result-based comparison** allows multiple correct query formulations
2. **Edge case documentation** shows attention to detail
3. **Security section** demonstrates awareness of production concerns
4. **Maintenance guidelines** show long-term thinking

Sample Discussion Topics

- Why compare results instead of query strings?
- How would you add a new level?
- What happens if a query times out?

- How do you handle NULL values in comparisons?
-

Document Revision History

Version	Date	Changes
1.0	January 2026	Initial release with all 7 levels

End of Document

Classification: CONFIDENTIAL - ADMIN ACCESS ONLY
