SQL Quick Reference Guide

For Kernel Module Detective CTF

■ Table of Contents

1. Basic SELECT Queries

Retrieve data from a single table:

Select all columns:

```
SELECT * FROM table_name;
```

Select specific columns:

```
SELECT column1, column2, column3
FROM table_name;
```

Select with column aliases:

```
SELECT column1 AS alias1, column2 AS alias2
FROM table_name;
```

Select distinct values:

```
SELECT DISTINCT column_name
FROM table_name;
```

2. Filtering with WHERE

Filter rows based on conditions:

Basic comparison operators:

```
SELECT * FROM table_name
WHERE column1 = 'value';

-- Operators: =, !=, <, >, <=, >=, <> (not equal)
```

Multiple conditions (AND, OR):

```
SELECT * FROM table_name
WHERE condition1 AND condition2;

SELECT * FROM table_name
WHERE condition1 OR condition2;
```

IN operator:

```
SELECT * FROM table_name
WHERE column_name IN ('value1', 'value2', 'value3');
```

BETWEEN operator:

```
SELECT * FROM table_name
WHERE column_name BETWEEN 100 AND 200;
```

LIKE operator (pattern matching):

```
SELECT * FROM table_name
WHERE column_name LIKE 'pattern%';

-- % matches any sequence
-- _ matches single character
```

NULL checks:

```
SELECT * FROM table_name
WHERE column_name IS NULL;

SELECT * FROM table_name
WHERE column_name IS NOT NULL;
```

3. Sorting and Limiting Results

ORDER BY (ascending):

```
SELECT * FROM table_name
ORDER BY column1 ASC;
```

ORDER BY (descending):

```
SELECT * FROM table_name
ORDER BY column1 DESC;
```

Multiple sort columns:

```
SELECT * FROM table_name
ORDER BY column1 DESC, column2 ASC;
```

LIMIT results:

```
SELECT * FROM table_name
LIMIT 10;
```

LIMIT with OFFSET:

```
SELECT * FROM table_name
LIMIT 10 OFFSET 20; -- Skip first 20, return next 10
```

4. Aggregate Functions

Perform calculations on sets of rows:

Common aggregate functions:

```
SELECT COUNT(*) FROM table_name;
SELECT COUNT(DISTINCT column) FROM table_name;

SELECT SUM(column_name) FROM table_name;
SELECT AVG(column_name) FROM table_name;
SELECT MIN(column_name) FROM table_name;
SELECT MAX(column_name) FROM table_name;
```

Combining aggregates:

```
SELECT
COUNT(*) AS total_rows,
AVG(column1) AS average_value,
MAX(column2) AS max_value
FROM table_name;
```

5. GROUP BY and HAVING

Basic GROUP BY:

```
SELECT column1, COUNT(*)
FROM table_name
GROUP BY column1;
```

Multiple grouping columns:

```
SELECT column1, column2, COUNT(*)
FROM table_name
GROUP BY column1, column2;
```

HAVING clause (filter after grouping):

```
SELECT column1, COUNT(*) AS count
FROM table_name
GROUP BY column1
HAVING COUNT(*) > 5;
```

HAVING vs WHERE:

- WHERE filters rows BEFORE grouping
- HAVING filters groups AFTER aggregation

Example combining WHERE and HAVING:

```
SELECT module_name, COUNT(*) AS failures
FROM module_events
WHERE status = 'FAILED'
GROUP BY module_name
HAVING COUNT(*) > 3
ORDER BY failures DESC;
```

6. JOIN Operations

Combine data from multiple tables:

INNER JOIN (matching rows only):

```
SELECT t1.column1, t2.column2
FROM table1 AS t1
INNER JOIN table2 AS t2
ON t1.id = t2.foreign_id;
```

LEFT JOIN (all rows from left table):

```
SELECT t1.column1, t2.column2
FROM table1 AS t1
LEFT JOIN table2 AS t2
ON t1.id = t2.foreign_id;
```

Multiple JOINs:

```
SELECT t1.col1, t2.col2, t3.col3
FROM table1 AS t1
INNER JOIN table2 AS t2 ON t1.id = t2.id
INNER JOIN table3 AS t3 ON t2.id = t3.id
WHERE t1.status = 'active';
```

Self JOIN:

```
SELECT a.column1, b.column2
FROM table AS a
INNER JOIN table AS b
ON a.id = b.parent_id;
```

JOIN with aggregation:

```
SELECT t1.module_name, COUNT(t2.error_id) AS error_count
FROM module_events AS t1
LEFT JOIN error_codes AS t2
ON t1.module_name = t2.affected_module
GROUP BY t1.module_name;
```

7. Subqueries

Subquery in WHERE clause:

```
SELECT * FROM table1
WHERE column1 IN (
SELECT column2 FROM table2
WHERE condition
);
```

Subquery in FROM clause:

```
SELECT subq.col1, subq.count
FROM (
SELECT column1 AS col1, COUNT(*) AS count
FROM table_name
GROUP BY column1
) AS subq
WHERE subq.count > 10;
```

Correlated subquery:

```
SELECT *
FROM table1 AS t1
WHERE EXISTS (
SELECT 1 FROM table2 AS t2
WHERE t2.foreign_id = t1.id
AND t2.status = 'FAILED'
);
```

Subquery with aggregates:

```
SELECT *
FROM table1
WHERE value > (
SELECT AVG(value) FROM table1
);
```

8. Common Table Expressions (CTEs)

Named temporary result sets for complex queries:

Basic CTE:

```
WITH cte_name AS (
SELECT column1, COUNT(*) AS count
FROM table_name
GROUP BY column1
)
SELECT * FROM cte_name
WHERE count > 5;
```

Multiple CTEs:

```
WITH
failed_modules AS (
SELECT module_name, COUNT(*) AS failures
FROM module_events
WHERE status = 'FAILED'
GROUP BY module_name
),
critical_errors AS (
SELECT affected_module, COUNT(*) AS errors
FROM error_codes
WHERE severity = 'CRITICAL'
GROUP BY affected_module
)
SELECT
fm.module_name,
fm.failures,
ce.errors
FROM failed_modules AS fm
INNER JOIN critical_errors AS ce
ON fm.module_name = ce.affected_module;
```

Recursive CTE (advanced):

```
WITH RECURSIVE counter(n) AS (
SELECT 1
UNION ALL
SELECT n + 1 FROM counter WHERE n < 10
)
SELECT * FROM counter;
```

9. Set Operations

UNION (combine, remove duplicates):

```
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;
```

UNION ALL (combine, keep duplicates):

```
SELECT column1 FROM table1
UNION ALL
SELECT column1 FROM table2;
```

INTERSECT (common rows):

```
SELECT module_name FROM module_events
WHERE status = 'FAILED'
INTERSECT
SELECT affected_module FROM error_codes
WHERE severity = 'CRITICAL';
```

EXCEPT (rows in first but not second):

```
SELECT module_name FROM module_events
EXCEPT
SELECT module_name FROM module_events
WHERE status = 'SUCCESS';
```

■■ Note: Column count and types must match in all queries

10. Window Functions

Perform calculations across rows without collapsing results:

ROW_NUMBER:

```
SELECT
module_name,
failures,
ROW_NUMBER() OVER (ORDER BY failures DESC) AS rank
FROM module_summary;
```

RANK and DENSE_RANK:

```
SELECT
module_name,
error_count,
RANK() OVER (ORDER BY error_count DESC) AS rank,
DENSE_RANK() OVER (ORDER BY error_count DESC) AS dense_rank
FROM error_summary;
```

Partition by (grouping within windows):

```
SELECT
boot_session,
module_name,
COUNT(*) AS errors,
ROW_NUMBER() OVER (
PARTITION BY boot_session
ORDER BY COUNT(*) DESC
) AS rank_in_session
FROM error_codes
GROUP BY boot_session, module_name;
```

Aggregate window functions:

```
SELECT
timestamp,
value,
AVG(value) OVER (
ORDER BY timestamp
ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
) AS moving_avg
FROM measurements;
```

11. Operator Precedence

Understanding how SQL evaluates complex conditions:

Precedence order (highest to lowest):

1	Parentheses ()	Force evaluation order
2	Comparison	=, !=, <, >, <=, >=, IN, BETWEEN, LIKE
3	NOT	Logical NOT
4	AND	Logical AND
5	OR	Logical OR

Example WITHOUT parentheses:

```
-- Evaluates as: status = 'FAILED' AND (severity = 'HIGH' OR severity = 'CRITICAL')
SELECT * FROM errors
WHERE status = 'FAILED' AND severity = 'HIGH' OR severity = 'CRITICAL';
```

Example WITH parentheses (recommended):

```
-- Explicit grouping makes intent clear
SELECT * FROM errors
WHERE status = 'FAILED'
AND (severity = 'HIGH' OR severity = 'CRITICAL');
```

Complex example:

```
SELECT * FROM events
WHERE (type = 'ERROR' OR type = 'WARNING')
AND module_name IN ('mod1', 'mod2')
AND (timestamp > 1000 OR priority = 'HIGH')
AND NOT ignored = 1;
```

12. SQLite-Specific Features

Database metadata:

```
-- List all tables
SELECT name FROM sqlite_master
WHERE type='table';

-- Get table schema
PRAGMA table_info(table_name);
```

Type conversion (CAST):

```
SELECT CAST(column AS INTEGER) FROM table;
SELECT CAST(column AS REAL) FROM table;
SELECT CAST(column AS TEXT) FROM table;
```

String functions:

```
SELECT LENGTH(column) FROM table;
SELECT UPPER(column) FROM table;
SELECT LOWER(column) FROM table;
SELECT SUBSTR(column, 1, 5) FROM table;
SELECT REPLACE(column, 'old', 'new') FROM table;
```

Math functions:

```
SELECT ABS(column) FROM table;
SELECT ROUND(column, 2) FROM table;
SELECT MAX(col1, col2) FROM table;
SELECT MIN(col1, col2) FROM table;
```

Date/time functions:

```
SELECT datetime(timestamp, 'unixepoch') FROM table;
SELECT strftime('%Y-%m-%d', timestamp, 'unixepoch') FROM table;
```

EXPLAIN QUERY PLAN (analyze performance):

```
EXPLAIN QUERY PLAN
SELECT * FROM table WHERE column = 'value';
```

■ Quick Tips & Best Practices

Use aliases: Make complex queries readable with table and column aliases

Indent properly: Format multi-line queries with proper indentation

Test incrementally: Build complex queries step by step

Use EXPLAIN: Check query execution plans for performance

Comment your code: Use -- for single line or /* */ for blocks

Be explicit: Use parentheses in complex WHERE clauses

Choose right JOIN: INNER for matches, LEFT for optional relationships

GROUP before FILTER: WHERE filters rows, HAVING filters groups

Name your CTEs: Use descriptive names for Common Table Expressions

Watch for NULL: Use IS NULL, not = NULL

Good luck with the CTF! ■