Complete SQL/MySQL Reference Guide for Analytics Interviews

📚 Part 1: Comprehensive MySQL Syntax Reference

1. Basic Query Structure

```sql
-- Basic SELECT syntax
SELECT column1, column2, ...
FROM table_name
WHERE condition
GROUP BY column1, column2
HAVING aggregate_condition
ORDER BY column1 [ASC|DESC]
LIMIT number;
```

2. Data Types in MySQL

```sql
-- Numeric Types
INT, BIGINT, DECIMAL(M,D), FLOAT, DOUBLE

-- String Types
VARCHAR(n), CHAR(n), TEXT, ENUM('value1','value2')

-- Date/Time Types
DATE, TIME, DATETIME, TIMESTAMP, YEAR

-- Common Conversions
CAST(column AS datatype)
CONVERT(column, datatype)
```

3. Operators & Conditions

```sql
```

```
-- Comparison Operators
=, !=, <>, <, >, <=, >=
BETWEEN value1 AND value2
IN (value1, value2, ...)
LIKE 'pattern%'
REGEXP 'pattern'
IS NULL, IS NOT NULL

-- Logical Operators
AND, OR, NOT
EXISTS, NOT EXISTS

-- Arithmetic Operators
+, -, *, /, % (modulo), DIV (integer division)

-- Bitwise Operators
&, |, ^, ~, <<, >>
```

4. String Functions

sql

```
-- Manipulation
CONCAT(str1, str2, ...)
CONCAT_WS(separator, str1, str2, ...)
SUBSTRING(str, pos, length)
LEFT(str, length), RIGHT(str, length)
TRIM(), LTRIM(), RTRIM()
UPPER(), LOWER()
REPLACE(str, from_str, to_str)
REVERSE(str)

-- Analysis
LENGTH(str), CHAR_LENGTH(str)
INSTR(str, substr) -- position of substring
LOCATE(substr, str, pos)
STRCMP(str1, str2)

-- Formatting
FORMAT(number, decimals)
LPAD(str, length, padstr), RPAD(str, length, padstr)
```

5. Date & Time Functions

```sql
-- Current Date/Time
NOW(), CURDATE(), CURTIME()
CURRENT_TIMESTAMP()

-- Extraction
YEAR(date), MONTH(date), DAY(date)
HOUR(time), MINUTE(time), SECOND(time)
DAYNAME(date), MONTHNAME(date)
DAYOFWEEK(date), DAYOFYEAR(date)
WEEK(date), QUARTER(date)

-- Manipulation
DATE_ADD(date, INTERVAL value unit)
DATE_SUB(date, INTERVAL value unit)
DATEDIFF(date1, date2)
TIMESTAMPDIFF(unit, datetime1, datetime2)
DATE_FORMAT(date, format)

-- Common Units: MICROSECOND, SECOND, MINUTE, HOUR, DAY, WEEK, MONTH, QUARTER, YEAR
```

6. Aggregate Functions

```sql
-- Basic Aggregates
COUNT(*), COUNT(column), COUNT(DISTINCT column)
SUM(column), AVG(column)
MIN(column), MAX(column)
GROUP_CONCAT(column SEPARATOR ',')

-- Statistical
STD(column), STDDEV(column)
VARIANCE(column)

-- Conditional Aggregation
COUNT(CASE WHEN condition THEN 1 END)
SUM(CASE WHEN condition THEN column END)
```

7. Window Functions (MySQL 8.0+)

```sql
-- Syntax
function_name() OVER (
    [PARTITION BY column1, column2, ...]
    [ORDER BY column1 [ASC|DESC], ...]
    [ROWS/RANGE frame_clause]
)

-- Ranking Functions
ROW_NUMBER() OVER (...)
RANK() OVER (...)
DENSE_RANK() OVER (...)
PERCENT_RANK() OVER (...)
NTILE(n) OVER (...)

-- Value Functions
FIRST_VALUE(column) OVER (...)
LAST_VALUE(column) OVER (...)
NTH_VALUE(column, n) OVER (...)
LAG(column, offset, default) OVER (...)
LEAD(column, offset, default) OVER (...)

-- Aggregate Window Functions
SUM(column) OVER (...)
AVG(column) OVER (...)
COUNT(column) OVER (...)
MAX(column) OVER (...)
MIN(column) OVER (...)

-- Frame Clauses
ROWS BETWEEN UNBOUNDED PRECEDING AND CURRENT ROW
ROWS BETWEEN n PRECEDING AND n FOLLOWING
ROWS BETWEEN CURRENT ROW AND UNBOUNDED FOLLOWING
RANGE BETWEEN INTERVAL n DAY PRECEDING AND CURRENT ROW
```

8. Joins

```sql
```

```sql
-- Inner Join
SELECT * FROM table1
INNER JOIN table2 ON table1.id = table2.id;

-- Left Join (Left Outer Join)
SELECT * FROM table1
LEFT JOIN table2 ON table1.id = table2.id;

-- Right Join (Right Outer Join)
SELECT * FROM table1
RIGHT JOIN table2 ON table1.id = table2.id;

-- Full Outer Join (MySQL workaround using UNION)
SELECT * FROM table1 LEFT JOIN table2 ON table1.id = table2.id
UNION
SELECT * FROM table1 RIGHT JOIN table2 ON table1.id = table2.id;

-- Cross Join (Cartesian Product)
SELECT * FROM table1
CROSS JOIN table2;

-- Self Join
SELECT a.column, b.column
FROM table1 a
JOIN table1 b ON a.id = b.parent_id;

-- Multiple Joins
SELECT *
FROM table1 t1
JOIN table2 t2 ON t1.id = t2.t1_id
JOIN table3 t3 ON t2.id = t3.t2_id;
```

9. Subqueries

sql

```sql
-- Scalar Subquery (returns single value)
SELECT column1,
    (SELECT MAX(column2) FROM table2) as max_value
FROM table1;

-- Column Subquery (returns single column)
SELECT * FROM table1
WHERE id IN (SELECT id FROM table2 WHERE condition);

-- Row Subquery (returns single row)
SELECT * FROM table1
WHERE (column1, column2) = (SELECT col1, col2 FROM table2 LIMIT 1);

-- Table Subquery (returns table)
SELECT * FROM (
  SELECT column1, COUNT(*) as cnt
  FROM table1
  GROUP BY column1
) AS derived_table
WHERE cnt > 10;

-- Correlated Subquery
SELECT * FROM table1 t1
WHERE column1 > (
  SELECT AVG(column1)
  FROM table1 t2
  WHERE t2.category = t1.category
);
```

10. Common Table Expressions (CTEs)

sql

```sql
-- Single CTE
WITH cte_name AS (
    SELECT column1, column2
    FROM table1
    WHERE condition
)
SELECT * FROM cte_name;

-- Multiple CTEs
WITH
cte1 AS (
    SELECT * FROM table1
),
cte2 AS (
    SELECT * FROM cte1 WHERE condition
)
SELECT * FROM cte2;

-- Recursive CTE
WITH RECURSIVE cte_name AS (
    -- Anchor member
    SELECT initial_query
    UNION ALL
    -- Recursive member
    SELECT recursive_query
    FROM cte_name
    WHERE termination_condition
)
SELECT * FROM cte_name;
```

11. Set Operations

sql

```sql
-- UNION (removes duplicates)
SELECT column1 FROM table1
UNION
SELECT column1 FROM table2;

-- UNION ALL (keeps duplicates)
SELECT column1 FROM table1
UNION ALL
SELECT column1 FROM table2;

-- INTERSECT (MySQL workaround)
SELECT DISTINCT column1 FROM table1
WHERE column1 IN (SELECT column1 FROM table2);

-- EXCEPT/MINUS (MySQL workaround)
SELECT DISTINCT column1 FROM table1
WHERE column1 NOT IN (SELECT column1 FROM table2);
```

12. Conditional Logic

sql

```sql
-- CASE Statement
CASE
    WHEN condition1 THEN result1
    WHEN condition2 THEN result2
    ELSE default_result
END

-- Simple CASE
CASE column
    WHEN value1 THEN result1
    WHEN value2 THEN result2
    ELSE default_result
END

-- IF Function
IF(condition, true_value, false_value)

-- IFNULL/COALESCE
IFNULL(column, default_value)
COALESCE(column1, column2, ..., default_value)

-- NULLIF
NULLIF(expression1, expression2) -- returns NULL if equal
```

13. Data Modification

sql

```sql
-- INSERT
INSERT INTO table_name (column1, column2)
VALUES (value1, value2);

INSERT INTO table_name
SELECT * FROM other_table;

-- UPDATE
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;

-- DELETE
DELETE FROM table_name
WHERE condition;

-- UPSERT (INSERT ON DUPLICATE KEY UPDATE)
INSERT INTO table_name (id, column1)
VALUES (1, 'value')
ON DUPLICATE KEY UPDATE column1 = VALUES(column1);

-- REPLACE (DELETE + INSERT)
REPLACE INTO table_name (id, column1)
VALUES (1, 'value');
```

🎯 Part 2: Most Common Interview Patterns & Combinations

Pattern 1: Funnel Analysis

Use Case: Track user conversion through multiple steps

sql

```sql
-- E-commerce Purchase Funnel
WITH funnel_events AS (
    SELECT
        user_id,
        MAX(CASE WHEN event_type = 'page_view' THEN 1 ELSE 0 END) as viewed,
        MAX(CASE WHEN event_type = 'add_to_cart' THEN 1 ELSE 0 END) as added_cart,
        MAX(CASE WHEN event_type = 'checkout' THEN 1 ELSE 0 END) as checked_out,
        MAX(CASE WHEN event_type = 'purchase' THEN 1 ELSE 0 END) as purchased
    FROM user_events
    WHERE event_date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY)
    GROUP BY user_id
)
SELECT
    COUNT(*) as total_users,
    SUM(viewed) as viewed_product,
    SUM(added_cart) as added_to_cart,
    SUM(checked_out) as reached_checkout,
    SUM(purchased) as completed_purchase,
    -- Conversion Rates
    ROUND(100.0 * SUM(added_cart) / NULLIF(SUM(viewed), 0), 2) as view_to_cart_rate,
    ROUND(100.0 * SUM(checked_out) / NULLIF(SUM(added_cart), 0), 2) as cart_to_checkout_rate,
    ROUND(100.0 * SUM(purchased) / NULLIF(SUM(checked_out), 0), 2) as checkout_to_purchase_rate,
    ROUND(100.0 * SUM(purchased) / NULLIF(SUM(viewed), 0), 2) as overall_conversion_rate
FROM funnel_events;
```

Pattern 2: Cohort Retention Analysis

Use Case: Track user retention over time

```sql
```

```sql
-- Monthly Cohort Retention
WITH cohort_data AS (
    SELECT
        user_id,
        DATE_FORMAT(MIN(created_date), '%Y-%m') as cohort_month,
        MIN(created_date) as first_date
    FROM users
    GROUP BY user_id
),
user_activities AS (
    SELECT
        u.user_id,
        c.cohort_month,
        c.first_date,
        DATE_FORMAT(u.activity_date, '%Y-%m') as activity_month,
        TIMESTAMPDIFF(MONTH, c.first_date, u.activity_date) as months_since_signup
    FROM user_activity u
    JOIN cohort_data c ON u.user_id = c.user_id
)
SELECT
    cohort_month,
    months_since_signup,
    COUNT(DISTINCT user_id) as active_users,
    ROUND(100.0 * COUNT(DISTINCT user_id) /
        FIRST_VALUE(COUNT(DISTINCT user_id)) OVER (PARTITION BY cohort_month ORDER BY months_since_
FROM user_activities
GROUP BY cohort_month, months_since_signup
ORDER BY cohort_month, months_since_signup;
```

Pattern 3: Moving Averages & Trends

Use Case: Smooth out daily fluctuations to see trends

```sql
```

```sql
-- 7-day and 30-day Moving Averages
WITH daily_metrics AS (
  SELECT
    DATE(created_at) as date,
    COUNT(*) as daily_orders,
    SUM(order_value) as daily_revenue
  FROM orders
  WHERE created_at >= DATE_SUB(CURDATE(), INTERVAL 90 DAY)
  GROUP BY DATE(created_at)
)
SELECT
  date,
  daily_orders,
  daily_revenue,
  -- 7-day moving average
  AVG(daily_orders) OVER (
    ORDER BY date
    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) as ma7_orders,
  AVG(daily_revenue) OVER (
    ORDER BY date
    ROWS BETWEEN 6 PRECEDING AND CURRENT ROW
  ) as ma7_revenue,
  -- 30-day moving average
  AVG(daily_orders) OVER (
    ORDER BY date
    ROWS BETWEEN 29 PRECEDING AND CURRENT ROW
  ) as ma30_orders,
  -- Week-over-week growth
  (daily_revenue - LAG(daily_revenue, 7) OVER (ORDER BY date)) /
  NULLIF(LAG(daily_revenue, 7) OVER (ORDER BY date), 0) * 100 as wow_growth
FROM daily_metrics
ORDER BY date DESC;
```

Pattern 4: Top N per Group

Use Case: Find top performers in each category

```sql
```

```sql
-- Top 3 Products per Category by Revenue
WITH product_revenue AS (
    SELECT
        p.category,
        p.product_id,
        p.product_name,
        SUM(o.quantity * o.price) as total_revenue,
        ROW_NUMBER() OVER (
            PARTITION BY p.category
            ORDER BY SUM(o.quantity * o.price) DESC
        ) as revenue_rank
    FROM products p
    JOIN order_items o ON p.product_id = o.product_id
    WHERE o.created_at >= DATE_SUB(CURDATE(), INTERVAL 30 DAY)
    GROUP BY p.category, p.product_id, p.product_name
)
SELECT
    category,
    product_id,
    product_name,
    total_revenue,
    revenue_rank
FROM product_revenue
WHERE revenue_rank <= 3
ORDER BY category, revenue_rank;
```

Pattern 5: User Segmentation

Use Case: Classify users based on behavior

sql

```sql
-- RFM (Recency, Frequency, Monetary) Segmentation
WITH user_metrics AS (
  SELECT
    user_id,
    DATEDIFF(CURDATE(), MAX(order_date)) as recency_days,
    COUNT(DISTINCT order_id) as frequency,
    SUM(order_value) as monetary_value
  FROM orders
  WHERE order_date >= DATE_SUB(CURDATE(), INTERVAL 365 DAY)
  GROUP BY user_id
),
user_scores AS (
  SELECT
    user_id,
    recency_days,
    frequency,
    monetary_value,
    NTILE(5) OVER (ORDER BY recency_days DESC) as recency_score,
    NTILE(5) OVER (ORDER BY frequency) as frequency_score,
    NTILE(5) OVER (ORDER BY monetary_value) as monetary_score
  FROM user_metrics
)
SELECT
  user_id,
  recency_days,
  frequency,
  monetary_value,
  CONCAT(recency_score, frequency_score, monetary_score) as rfm_score,
  CASE
    WHEN recency_score >= 4 AND frequency_score >= 4 AND monetary_score >= 4 THEN 'Champions'
    WHEN recency_score >= 3 AND frequency_score >= 3 AND monetary_score >= 3 THEN 'Loyal Customers'
    WHEN recency_score >= 3 AND frequency_score <= 2 THEN 'New Customers'
    WHEN recency_score <= 2 AND frequency_score >= 3 THEN 'At Risk'
    WHEN recency_score <= 2 AND frequency_score <= 2 AND monetary_score >= 3 THEN 'Cant Lose Them'
    ELSE 'Others'
  END as customer_segment
FROM user_scores;
```

Pattern 6: Cumulative Metrics

Use Case: Running totals and cumulative calculations

```sql
-- Cumulative Revenue and Customer Count
WITH daily_stats AS (
    SELECT
        DATE(created_at) as order_date,
        COUNT(DISTINCT customer_id) as new_customers,
        COUNT(*) as orders,
        SUM(order_value) as revenue
    FROM orders
    WHERE created_at >= DATE_FORMAT(CURDATE(), '%Y-%m-01')
    GROUP BY DATE(created_at)
)
SELECT
    order_date,
    new_customers,
    orders,
    revenue,
    SUM(new_customers) OVER (ORDER BY order_date) as cumulative_customers,
    SUM(orders) OVER (ORDER BY order_date) as cumulative_orders,
    SUM(revenue) OVER (ORDER BY order_date) as cumulative_revenue,
    AVG(revenue) OVER (ORDER BY order_date) as running_avg_revenue
FROM daily_stats
ORDER BY order_date;
```

Pattern 7: Duplicate Detection

Use Case: Find duplicate records or repeated patterns

```sql
```

```sql
-- Find Duplicate Email Addresses with User Details
WITH duplicate_emails AS (
    SELECT
        email,
        COUNT(*) as duplicate_count,
        GROUP_CONCAT(user_id ORDER BY created_at) as user_ids,
        MIN(created_at) as first_created,
        MAX(created_at) as last_created
    FROM users
    GROUP BY email
    HAVING COUNT(*) > 1
)
SELECT
    d.email,
    d.duplicate_count,
    d.user_ids,
    d.first_created,
    d.last_created,
    u.user_id,
    u.username,
    u.created_at
FROM duplicate_emails d
JOIN users u ON d.email = u.email
ORDER BY d.email, u.created_at;
```

Pattern 8: Gap Analysis

Use Case: Find missing sequences or time gaps

```sql
```

```sql
-- Find Gaps in Sequential IDs
WITH id_gaps AS (
    SELECT
        id as current_id,
        LEAD(id) OVER (ORDER BY id) as next_id,
        LEAD(id) OVER (ORDER BY id) - id as gap_size
    FROM table_name
)
SELECT
    current_id + 1 as gap_start,
    next_id - 1 as gap_end,
    gap_size - 1 as missing_count
FROM id_gaps
WHERE gap_size > 1;

-- Find Days with No Activity
WITH date_range AS (
    SELECT DATE_SUB(CURDATE(), INTERVAL n DAY) as date
    FROM (
        SELECT @row := @row + 1 as n
        FROM (SELECT 0 UNION SELECT 1 UNION SELECT 2 UNION SELECT 3) t1,
             (SELECT 0 UNION SELECT 1 UNION SELECT 2 UNION SELECT 3) t2,
             (SELECT @row := -1) t3
        LIMIT 30
    ) numbers
),
activity_dates AS (
    SELECT DISTINCT DATE(activity_time) as activity_date
    FROM user_activity
    WHERE activity_time >= DATE_SUB(CURDATE(), INTERVAL 30 DAY)
)
SELECT d.date as missing_date
FROM date_range d
LEFT JOIN activity_dates a ON d.date = a.activity_date
WHERE a.activity_date IS NULL
ORDER BY d.date;
```

Pattern 9: Percentiles & Distribution

Use Case: Understand data distribution and outliers

sql

```sql
-- Calculate Percentiles for Order Values
WITH order_percentiles AS (
    SELECT
        order_value,
        PERCENT_RANK() OVER (ORDER BY order_value) as percentile_rank
    FROM orders
    WHERE order_date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY)
)
SELECT
    MIN(CASE WHEN percentile_rank >= 0.25 THEN order_value END) as p25,
    MIN(CASE WHEN percentile_rank >= 0.50 THEN order_value END) as p50_median,
    MIN(CASE WHEN percentile_rank >= 0.75 THEN order_value END) as p75,
    MIN(CASE WHEN percentile_rank >= 0.90 THEN order_value END) as p90,
    MIN(CASE WHEN percentile_rank >= 0.95 THEN order_value END) as p95,
    MIN(CASE WHEN percentile_rank >= 0.99 THEN order_value END) as p99
FROM order_percentiles;
```

Pattern 10: Time-based Comparisons

Use Case: Compare metrics across different time periods

sql

```sql
-- Year-over-Year Comparison by Month
WITH monthly_metrics AS (
    SELECT
        YEAR(order_date) as year,
        MONTH(order_date) as month,
        COUNT(DISTINCT customer_id) as customers,
        COUNT(*) as orders,
        SUM(order_value) as revenue
    FROM orders
    WHERE order_date >= DATE_SUB(CURDATE(), INTERVAL 2 YEAR)
    GROUP BY YEAR(order_date), MONTH(order_date)
)
SELECT
    cur.year,
    cur.month,
    cur.revenue as current_revenue,
    prev.revenue as previous_year_revenue,
    cur.revenue - prev.revenue as absolute_change,
    ROUND((cur.revenue - prev.revenue) / NULLIF(prev.revenue, 0) * 100, 2) as yoy_growth_rate,
    cur.orders as current_orders,
    prev.orders as previous_year_orders,
    ROUND(cur.revenue / NULLIF(cur.orders, 0), 2) as current_aov,
    ROUND(prev.revenue / NULLIF(prev.orders, 0), 2) as previous_aov
FROM monthly_metrics cur
LEFT JOIN monthly_metrics prev
    ON cur.month = prev.month
    AND cur.year = prev.year + 1
WHERE cur.year = YEAR(CURDATE())
ORDER BY cur.month;
```

💡 Interview Pro Tips

1. **Always Start with Questions**

- What's the data schema?
- Are there NULL values to handle?
- What's the expected output format?
- What's the scale of data?

2. **Optimize for Performance**

```sql
-- Use indexes effectively
WHERE indexed_column = value  -- Good
WHERE FUNCTION(indexed_column) = value  -- Bad, can't use index

-- Limit data early
WITH filtered_data AS (
   SELECT * FROM large_table
   WHERE date >= DATE_SUB(CURDATE(), INTERVAL 30 DAY)  -- Filter early
)

-- Use EXISTS instead of IN for large datasets
WHERE EXISTS (SELECT 1 FROM table2 WHERE condition)  -- Better
WHERE column IN (SELECT column FROM table2)  -- Slower for large sets
```

3. **Handle Edge Cases**

```sql
-- Division by zero
ROUND(numerator / NULLIF(denominator, 0), 2)

-- NULL handling
COALESCE(column, 0)
IFNULL(column, 'default')

-- Empty results
LEFT JOIN to preserve all records
```

4. **Common Gotchas**

- **COUNT(*) vs COUNT(column)**: COUNT(*) counts all rows, COUNT(column) excludes NULLs
- **DISTINCT with multiple columns**: `COUNT(DISTINCT col1, col2)` treats combination as unique
- **GROUP BY with non-aggregated columns**: MySQL allows this but can give unexpected results
- **Date comparisons**: Always be explicit with time components

5. **Testing Your Query**

```sql
```

```sql
-- Test with small sample first
... LIMIT 10;

-- Check for duplicates
SELECT column, COUNT(*) as cnt
FROM table
GROUP BY column
HAVING COUNT(*) > 1;

-- Validate calculations
SELECT
  SUM(parts) as sum_of_parts,
  total,
  SUM(parts) = total as validation_check
FROM ...
```