# MySQL and Query Optimization Cheat Sheet

## 📊 Database Architecture & Core Concepts

### MySQL Engine Types

| Engine | Transactions | Locking Level | Use Case |
|---|---|---|---|
| **InnoDB** | ✅ Yes | Row-level | Default, ACID compliance, high concurrency |
| **MyISAM** | ❌ No | Table-level | Read-heavy, full-text search (legacy) |
| **Memory** | ❌ No | Table-level | Temporary tables, fast lookups |
| **Archive** | ❌ No | Row-level | Logging, high-insert, rarely read |

```
-- Check table engine
SHOW TABLE STATUS WHERE Name = 'users';

-- Change table engine
ALTER TABLE users ENGINE = InnoDB;
```

### Transaction Isolation Levels

| Isolation Level | Dirty Reads | Non-Repeatable Reads | Phantom Reads | Performance |
|---|---|---|---|---|
| **READ UNCOMMITTED** | Possible | Possible | Possible | Highest |
| **READ COMMITTED** | Prevented | Possible | Possible | High |
| **REPEATABLE READ** | Prevented | Prevented | Possible *(InnoDB prevents)* | Medium |
| **SERIALIZABLE** | Prevented | Prevented | Prevented | Lowest |

```
-- Check current isolation level
SELECT @@transaction_isolation;

-- Set session isolation level
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

### Storage Architecture

- **Buffer Pool**: In-memory data and index caching (key tuning parameter)
- **Redo Log**: Write-ahead log for crash recovery (durability)
- **Undo Logs**: Enables transaction rollback and MVCC
- **Double Write Buffer**: Prevents partial page writes
- **Change Buffer**: Caches secondary index changes for non-unique indexes

## 🔍 Indexing Strategies

### Index Types

| Type | Description | Best For |
|---|---|---|
| **B-Tree** | Default balanced tree structure | Most queries, equality, ranges |
| **Hash** | Fast key lookups | Only equality comparisons |
| **Full-Text** | Text search capabilities | Natural language search |
| **Spatial** | Geospatial data | Geographic data |
| **Composite** | Multiple columns | Queries filtering on multiple fields |

### Index Selection Guidelines

1. **Selectivity**: High selectivity = better index performance

   ```
   -- Check index selectivity
   SELECT COUNT(DISTINCT column_name)/COUNT(*) FROM table_name;
   -- Higher ratio = better selectivity
   ```

2. **ESR Rule**: Equality, Sort, Range (order of columns in composite indexes)

```
-- Good: Index matches ESR rule
CREATE INDEX idx_users_status_created ON users(status, created_at);
-- Query using status (E) and sorting by created_at (S)
SELECT * FROM users WHERE status = 'active' ORDER BY created_at D
ESC LIMIT 10;
```

3. **Covering Index**: Include all columns needed by query

```
-- Covering index example
CREATE INDEX idx_users_email_name ON users(email, name);
-- Query covered entirely by index
SELECT name FROM users WHERE email = 'user@example.com';
```

4. **Prefix Index**: Index partial string (saves space)

```
-- Index just first 10 chars of description
CREATE INDEX idx_products_desc ON products(description(10));
```

5. **Functional Index**: Index based on expression (MySQL 8.0+)

```
-- Index on lowercase email
CREATE INDEX idx_users_email_lower ON users((LOWER(email)));
-- Now optimized
SELECT * FROM users WHERE LOWER(email) = 'user@example.com';
```

## Index Maintenance

```
-- Show indexes on table
SHOW INDEX FROM users;

-- Add index
CREATE INDEX idx_users_email ON users(email);

-- Add composite index
CREATE INDEX idx_users_status_created ON users(status, created_at);

-- Remove index
```

```
DROP INDEX idx_users_email ON users;

-- Find unused indexes
SELECT * FROM schema_unused_indexes;
-- Requires MySQL Enterprise Monitor or custom monitoring

-- Find missing indexes
SELECT
    table_name,
    COUNT(*) AS rows_scanned,
    COUNT(*) * AVG_ROW_LENGTH AS bytes_scanned
FROM information_schema.TABLES
JOIN information_schema.COLUMNS USING (TABLE_SCHEMA, TABLE_NA
ME)
WHERE TABLE_SCHEMA = 'your_database'
  AND NOT EXISTS (
    SELECT 1
    FROM information_schema.STATISTICS
    WHERE TABLE_SCHEMA = 'your_database'
      AND TABLE_NAME = information_schema.TABLES.TABLE_NAME
      AND COLUMN_NAME = information_schema.COLUMNS.COLUMN_NAM
E
  )
GROUP BY table_name
ORDER BY rows_scanned DESC;
```

## 🚀 Query Optimization Techniques

### EXPLAIN and Execution Plan Analysis

```
-- Basic EXPLAIN
EXPLAIN SELECT * FROM users WHERE email = 'user@example.com';

-- EXPLAIN with execution information
EXPLAIN ANALYZE SELECT * FROM users WHERE email = 'user@example.
com';

-- Format with JSON for more details
```

```
EXPLAIN FORMAT=JSON SELECT * FROM users JOIN orders ON users.id =
orders.user_id;
```

## Key EXPLAIN Output Fields

| Field | Meaning | Good Values | Bad Values |
|---|---|---|---|
| **type** | Access method used | `const`, `ref`, `range` | `ALL`, `index` (full scan) |
| **key** | Index used | Any named index | `NULL` (no index used) |
| **rows** | Estimated rows to examine | Lower numbers | High numbers |
| **filtered** | % of rows filtered | Higher % (closer to 100%) | Low % |
| **Extra** | Additional execution info | `Using index`, `Using where` | `Using filesort`, `Using temporary` |

## Common Query Anti-patterns

## Common Query Anti-patterns

1. **SELECT * instead of specific columns**

```
-- Bad
SELECT * FROM users JOIN orders ON users.id = orders.user_id;

-- Good
SELECT u.id, u.email, o.total
FROM users u JOIN orders o ON u.id = o.user_id;
```

2. **Implicit type conversion breaking indexes**

```
-- Bad (id is INT, string comparison breaks index)
SELECT * FROM users WHERE id = '10';

-- Good
SELECT * FROM users WHERE id = 10;
```

3. **LIKE with leading wildcard**

```
-- Bad (can't use index efficiently)
SELECT * FROM users WHERE email LIKE '%gmail.com';

-- Better
SELECT * FROM users WHERE email LIKE 'john%';

-- Best for searching anywhere
CREATE FULLTEXT INDEX idx_products_desc ON products(description);
SELECT * FROM products WHERE MATCH(description) AGAINST('keyword');
```

4. **OR conditions with mixed indexed/non-indexed columns**

```
-- Bad (won't use index)
SELECT * FROM users WHERE email = 'test@example.com' OR address LIKE '%Main St%';

-- Better (union approach)
SELECT * FROM users WHERE email = 'test@example.com'
UNION
SELECT * FROM users WHERE address LIKE '%Main St%';
```

5. **COUNT(*) on large tables**

```
-- Bad for large tables
SELECT COUNT(*) FROM huge_table;

-- Better
SELECT table_rows
FROM information_schema.tables
WHERE table_schema = 'your_database' AND table_name = 'huge_table';
-- Note: This is an approximation after ANALYZE TABLE
```

6. **Redundant JOINs**

```
-- Bad (extra join not needed)
SELECT o.* FROM orders o
```

```
JOIN users u ON o.user_id = u.id
WHERE o.status = 'completed';

-- Good
SELECT * FROM orders WHERE status = 'completed';
```

### N+1 Query Problem (Critical for ORM users)

```
-- Bad (N+1 problem in application code)
-- Pseudocode:
users = SELECT * FROM users LIMIT 10;
for each user:
   orders = SELECT * FROM orders WHERE user_id = user.id;

-- Good (single query with JOIN)
SELECT u.*, o.*
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE u.id IN (SELECT id FROM users LIMIT 10);

-- Better (with pagination)
SELECT u.*,
       GROUP_CONCAT(o.id) as order_ids,
       GROUP_CONCAT(o.total) as order_totals
FROM users u
LEFT JOIN orders o ON u.id = o.user_id
WHERE u.id IN (SELECT id FROM users LIMIT 10)
GROUP BY u.id;
```

## 📊 Advanced SQL Techniques

### Window Functions (MySQL 8.0+)

```
-- Row numbering
SELECT
   id,
   category,
    price,
    ROW_NUMBER() OVER(PARTITION BY category ORDER BY price DESC) a
s price_rank
FROM products;

-- Running totals
SELECT
    date,
    amount,
    SUM(amount) OVER(ORDER BY date) as running_total
FROM transactions;

-- Moving averages
SELECT
    date,
    price,
    AVG(price) OVER(ORDER BY date ROWS BETWEEN 6 PRECEDING AND C
URRENT ROW) as 7day_avg
FROM stock_prices;
```

## Common Table Expressions (CTEs)

```
-- Basic CTE
WITH active_users AS (
    SELECT * FROM users WHERE status = 'active' AND last_login > DATE_S
UB(NOW(), INTERVAL 30 DAY)
)
SELECT * FROM active_users WHERE subscription_tier = 'premium';

-- Recursive CTE (e.g., hierarchical data)
WITH RECURSIVE subordinates AS (
    -- Anchor member
    SELECT id, name, manager_id
    FROM employees
    WHERE id = 1

    UNION ALL
    -- Recursive member
    SELECT e.id, e.name, e.manager_id
    FROM employees e
    JOIN subordinates s ON e.manager_id = s.id
)
SELECT * FROM subordinates;
```

**JSON Operations (MySQL 8.0+)**

```
-- Store JSON
CREATE TABLE events (
    id INT PRIMARY KEY AUTO_INCREMENT,
    data JSON,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert JSON
INSERT INTO events (data) VALUES ('{"user_id": 123, "action": "login", "device": "mobile"}');

-- Query JSON
SELECT id, data->>'$.user_id' AS user_id, data->>'$.action' AS action
FROM events
WHERE data->>'$.device' = 'mobile';

-- Index JSON fields (MySQL 8.0.17+)
ALTER TABLE events ADD COLUMN user_id INT GENERATED ALWAYS AS
(data->>'$.user_id') STORED;
CREATE INDEX idx_events_user_id ON events(user_id);
```

## 🔧 Performance Tuning

### Server Configuration Parameters

| Parameter | Purpose | Recommended Setting |
|---|---|---|
| **innodb_buffer_pool_size** | Data/index caching | 70–80% of server RAM |
| **innodb_buffer_pool_instances** | Reduce contention | 8 (or equal to number of CPU cores) |

| Parameter | Purpose | Recommended Setting |
|---|---|---|
| **innodb_log_file_size** | Redo log size | 1–4 GB (larger = better performance, longer recovery) |
| **max_connections** | Connection limit | 151+ (based on application needs) |
| **thread_cache_size** | Thread reuse | 8–16 |
| **query_cache_size** | Query results cache | 0 (disable in MySQL 5.7+, removed in MySQL 8.0) |
| **tmp_table_size**, **max_heap_table_size** | In-memory temporary tables | 32–64 MB |

```
-- Check current settings
SHOW VARIABLES LIKE 'innodb_buffer_pool_size';

-- Check buffer pool hit ratio (should be >95%)
SELECT (1 - (
    SELECT variable_value
    FROM performance_schema.global_status
    WHERE variable_name = 'Innodb_buffer_pool_reads'
) / (
    SELECT variable_value
    FROM performance_schema.global_status
    WHERE variable_name = 'Innodb_buffer_pool_read_requests'
)) * 100 AS buffer_pool_hit_ratio;
```

## Connection Pooling

```
// HikariCP configuration (Java example)
@Bean
public DataSource dataSource() {
    HikariConfig config = new HikariConfig();
    config.setJdbcUrl("jdbc:mysql://localhost:3306/mydb");
    config.setUsername("user");
    config.setPassword("password");

    // Connection pool settings
    config.setMaximumPoolSize(10);          // Based on connection cost vs thr
eads
    config.setMinimumIdle(5);               // Keep some connections ready
    config.setIdleTimeout(300000);          // 5 minutes
    config.setMaxLifetime(1800000);         // 30 minutes
    config.setConnectionTimeout(30000);     // 30 seconds
    config.setValidationTimeout(5000);      // 5 seconds

    // Statement caching
    config.addDataSourceProperty("cachePrepStmts", "true");
    config.addDataSourceProperty("prepStmtCacheSize", "250");
    config.addDataSourceProperty("prepStmtCacheSqlLimit", "2048");

    return new HikariDataSource(config);
}
```

## Slow Query Analysis

```
-- Enable slow query log
SET GLOBAL slow_query_log = 'ON';
SET GLOBAL long_query_time = 1;  -- Log queries taking > 1 second
SET GLOBAL slow_query_log_file = '/var/log/mysql/slow-query.log';

-- Find slow queries
SELECT * FROM performance_schema.events_statements_summary_by_dig
est
ORDER BY sum_timer_wait DESC
LIMIT 10;

-- Find queries with full table scans
SELECT * FROM performance_schema.events_statements_summary_by_dig
est
WHERE digest_text NOT LIKE '%INFORMATION_SCHEMA%'
  AND digest_text NOT LIKE '%performance_schema%'
  AND sum_no_index_used > 0
ORDER BY sum_no_index_used DESC
LIMIT 10;
```

### Database Maintenance

```
-- Optimize table (defragment)
OPTIMIZE TABLE users;

-- Update statistics
ANALYZE TABLE users;

-- Check and repair tables
CHECK TABLE users;
REPAIR TABLE users;  -- Only for MyISAM and Archive tables

-- Check for table corruption
mysqlcheck -u root -p --check your_database
```

## 📈 Scaling & High Performance Patterns

### Read/Write Splitting

```
Application
    ↓
 ┌─────────┐
 │ Proxy  │ (ProxySQL, HAProxy, MySQL Router)
 └─────────┘
  ↙   ↘
┌────────┐ ┌───────┐
│Master│→│Slave│→Additional Slaves
└────────┘ └───────┘
```

```
# Spring Data JPA configuration example
spring.datasource.write.url=jdbc:mysql://master:3306/mydb
spring.datasource.write.username=write_user
spring.datasource.write.password=password

spring.datasource.read.url=jdbc:mysql://slave:3306/mydb
spring.datasource.read.username=read_user
spring.datasource.read.password=password
```

## Sharding Strategies

1. **Vertical Sharding**: Split tables across different databases

   ```
   DB1: Users, Profiles
   DB2: Orders, Payments
   DB3: Products, Inventory
   ```

2. **Horizontal Sharding**: Split rows across different servers

   ```
   Shard Key: user_id % 4
   Shard 0: users with IDs ending in 0, 4, 8
   Shard 1: users with IDs ending in 1, 5, 9
   Shard 2: users with IDs ending in 2, 6
   Shard 3: users with IDs ending in 3, 7
   ```

3. **Hash-based Sharding**: Use hash function to determine shard

   ```
   int shard = Math.abs(userId.hashCode()) % NUM_SHARDS;
   ```

## Caching Strategies

1. **Query Cache**: Cache frequently run queries (Redis or application-level)

2. **Object Cache**: Cache entities after loading from database

   ```
   @Cacheable("users")
   public User findById(Long id) {
       return userRepository.findById(id).orElse(null);
   }
   ```

3. **Cache Invalidation Patterns**:

   - Time-based expiration
   - Write-through (update cache + DB)
   - Write-behind (update cache, async DB update)

- Cache-aside (app manages cache/DB consistency)

## 🔬 Performance Testing & Monitoring

### Key MySQL Metrics to Monitor

1. **Query Performance**:
   - Queries per second (QPS)
   - Slow query count
   - Query latency percentiles (p95, p99)
2. **Resource Utilization**:
   - Connection count (vs. max_connections)
   - Buffer pool hit ratio
   - Disk I/O (reads/writes per second)
   - CPU usage
3. **InnoDB Metrics**:
   - Row operations (reads/updates/deletes per second)
   - Transaction throughput
   - Lock wait times
   - Deadlock count

```
-- Top 10 queries by execution time
SELECT
   SUBSTRING(digest_text, 1, 100) AS query_sample,
   count_star AS exec_count,
   round(avg_timer_wait/1000000000, 2) AS avg_exec_time_ms,
   round(sum_timer_wait/1000000000, 2) AS total_exec_time_ms
FROM performance_schema.events_statements_summary_by_digest
ORDER BY total_exec_time_ms DESC
LIMIT 10;
```

### Benchmarking Tools

1. **mysqlslap**: Built-in load simulation utility

```
mysqlslap --concurrency=50 --iterations=3 --query="SELECT * FROM
users WHERE status='active'" --create-schema=mydb
```

2. **sysbench**: Industry-standard database benchmark

```
sysbench --db-driver=mysql --mysql-user=root --mysql-password=pa
ssword \
  --mysql-db=mydb --range_size=100 \
  --table_size=10000 --tables=3 \
  --threads=6 --time=60 --events=0 \
  oltp_read_write prepare

sysbench --db-driver=mysql --mysql-user=root --mysql-password=pa
ssword \
  --mysql-db=mydb --range_size=100 \
  --table_size=10000 --tables=3 \
  --threads=6 --time=60 --events=0 \
  oltp_read_write run
```

## 📝 Practical Query Patterns

### Efficient Pagination

```
-- Bad (for deep pages)
SELECT * FROM products ORDER BY created_at DESC LIMIT 10000, 10;

-- Good (keyset pagination)
SELECT * FROM products
WHERE created_at < '2023-01-01 00:00:00'  -- Last timestamp from previo
us page
ORDER BY created_at DESC
LIMIT 10;
```

### Efficient Aggregation

```
-- Pre-aggregate data
CREATE TABLE daily_stats (
    date DATE PRIMARY KEY,
    orders_count INT,
    revenue DECIMAL(10,2),
    unique_customers INT
);

-- Update periodically rather than calculating on demand
INSERT INTO daily_stats
SELECT
    DATE(created_at) as date,
    COUNT(*) as orders_count,
    SUM(total) as revenue,
    COUNT(DISTINCT customer_id) as unique_customers
FROM orders
WHERE DATE(created_at) = CURDATE() - INTERVAL 1 DAY
GROUP BY DATE(created_at)
ON DUPLICATE KEY UPDATE
    orders_count = VALUES(orders_count),
    revenue = VALUES(revenue),
    unique_customers = VALUES(unique_customers);
```

### Efficient Joins

```
-- Push down WHERE conditions before JOIN
-- Bad
SELECT u.*, o.*
FROM users u
JOIN orders o ON u.id = o.user_id
WHERE u.status = 'active' AND o.created_at > '2023-01-01';

-- Good (filter early)
SELECT u.*, o.*
FROM users u
JOIN (
    SELECT * FROM orders
    WHERE created_at > '2023-01-01'
) o ON u.id = o.user_id
WHERE u.status = 'active';
```

## Upsert Pattern (Insert or Update)

```
-- Insert if not exists, update if exists
INSERT INTO users (id, email, name, updated_at)
VALUES (1, 'user@example.com', 'New User', NOW())
ON DUPLICATE KEY UPDATE
    name = VALUES(name),
    updated_at = VALUES(updated_at);
```

## Bulk Operations

```
-- Bulk insert (much faster than individual inserts)
INSERT INTO log_entries (user_id, action, created_at)
VALUES
    (1, 'login', NOW()),
    (2, 'purchase', NOW()),
    (3, 'update', NOW()),
    (4, 'login', NOW()),
    (5, 'logout', NOW());

-- Bulk update
UPDATE users
SET status = 'inactive'
WHERE id IN (1, 5, 10, 15, 20);

-- Batch processing pattern (for large datasets)
SET @batch_size = 10000;
SET @offset = 0;

process_loop: REPEAT
    UPDATE users
    SET last_login = NOW()
    WHERE status = 'active'
    LIMIT @batch_size;
    SET @offset = @offset + ROW_COUNT();
    SELECT SLEEP(0.1); -- Prevent server overload

UNTIL ROW_COUNT() = 0 END REPEAT;
```

## 🔒 Data Security & Compliance

### Data Encryption

```
-- Transparent Data Encryption (TDE)
-- Enable in my.cnf:
-- [mysqld]
-- early-plugin-load=keyring_file.so
-- keyring_file_data=/var/lib/mysql-keyring/keyring

-- Create encrypted table
CREATE TABLE users (
    id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(255) NOT NULL,
    password VARCHAR(255) NOT NULL
) ENCRYPTION='Y';

-- Column-level encryption
CREATE TABLE payments (
    id INT PRIMARY KEY,
    user_id INT,
    credit_card VARCHAR(255) NOT NULL,

    -- Store encrypted data
    credit_card_encrypted VARBINARY(255) GENERATED ALWAYS AS
        (AES_ENCRYPT(credit_card, UNHEX(SHA2('encryption_key', 512))))
);

-- Query encrypted data
SELECT
    id,
    AES_DECRYPT(credit_card_encrypted, UNHEX(SHA2('encryption_key', 5
12))) AS decrypted_cc
FROM payments;
```

## Audit Logging

```
-- Create audit table
CREATE TABLE audit_log (
    id INT AUTO_INCREMENT PRIMARY KEY,
    table_name VARCHAR(64) NOT NULL,
    record_id INT NOT NULL,
    action ENUM('INSERT', 'UPDATE', 'DELETE') NOT NULL,
    old_values JSON,
    new_values JSON,
    user_id INT,
    timestamp TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Audit trigger example
DELIMITER //
CREATE TRIGGER users_after_update
AFTER UPDATE ON users
FOR EACH ROW
BEGIN
    INSERT INTO audit_log (
        table_name, record_id, action,
        old_values, new_values, user_id
    )
    VALUES (
        'users', NEW.id, 'UPDATE',
        JSON_OBJECT(
            'email', OLD.email,
            'status', OLD.status
        ),
        JSON_OBJECT(
            'email', NEW.email,
            'status', NEW.status
        ),
        @current_user_id
```

```
  );
END //
DELIMITER ;
```

## 📋 Interview Preparation Tips

### Critical Concepts to Master

1. **Indexing strategies** - Explain when to use which type, covering indexes, index selectivity
2. **Query optimization techniques** - Demonstrate how to analyze and fix slow queries
3. **Execution plan interpretation** - Show ability to read EXPLAIN output and identify issues
4. **Database normalization** - Explain normal forms and when to denormalize
5. **Transaction isolation levels** - Describe consistency tradeoffs and real-world applications
6. **Scaling strategies** - Discuss horizontal vs. vertical scaling, sharding approaches
7. **Common performance bottlenecks** - Connection management, N+1 queries, inefficient queries

### Sample Interview Questions

1. "How would you optimize a query that's performing poorly?"
2. "Describe the difference between clustered and non-clustered indexes."
3. "How would you design a database to handle 1 million transactions per day?"
4. "What's the impact of adding an index to a write-heavy table?"
5. "How would you implement a soft delete with minimal performance impact?"
6. "Explain the differences between INNER JOIN, LEFT JOIN, and RIGHT JOIN with examples."
7. "How would you troubleshoot deadlocks in MySQL?"

8. "Design a sharding strategy for a social media application."
9. "How would you monitor MySQL performance in production?"
10. "Explain window functions and give examples of when they're useful."

### Practical Demonstration Tips

1. **Show your process** - Verbalize your thinking about query optimization step by step
2. **Use real examples** - Discuss actual performance issues you've solved
3. **Have metrics ready** - "I improved query performance from 2s to 50ms by..."
4. **Reference tools** - Mention tools like Percona Toolkit, MySQL Workbench, slow query analysis
5. **Mention both theoretical knowledge and practical experience**

---

## 📚 Further Resources

- **High Performance MySQL** (O'Reilly) - Industry-standard reference
- **MySQL 8.0 Reference Manual** - Official documentation
- **Percona Database Performance Blog** - Advanced MySQL performance topics
- **Use the Index, Luke!** - Excellent resource on indexing strategies

This cheat sheet covers MySQL and query optimization concepts required for senior backend roles at the 60 LPA level. It focuses on the practical aspects you'll need to discuss in interviews and implement in high-scale production environments.