**Answer**:
Normalize the schema and index fields used in filters.

**Steps to Design**:

1. Use normalized tables for better data organization.
2. Create indexes on columns frequently used in search filters.
3. Use dynamic queries or a search engine like Elasticsearch for complex filters.

**Code Example**:

```sql
CREATE INDEX idx_product_category_price ON products(category, price);
SELECT * FROM products
WHERE category = 'Electronics' AND price BETWEEN 100 AND 500;
```

**Explanation**:

- Indexing reduces filter query times, and search engines can handle unstructured or full-text searches.

---

### **Q33: You

put seperator and then add heading

---

# Database Scenario-Based Questions

---

### Q31: A report needs to display data from multiple tables with millions of rows. How would you optimize the query for performance?

**Answer**:
Use query optimization techniques like indexing, denormalization, and partitioning.

**Steps to Optimize**:

1. **Indexes**: Add composite or covering indexes to speed up join conditions.
2. **Partitioning**: Partition large tables to reduce query scanning time.
3. **Materialized Views**: Use materialized views for precomputed aggregated data.
4. **Query Caching**: Cache frequently accessed report data.

**Code Example**:

```sql
CREATE INDEX idx_report_date ON orders(order_date);
CREATE MATERIALIZED VIEW report_summary AS
SELECT customer_id, SUM(order_amount) AS total_spent
FROM orders
GROUP BY customer_id;
```

**Explanation**:

- Indexing speeds up table joins and filtering. Materialized views reduce redundant computations.

---

### Q32: An application must allow users to search with complex filters. How would you design the database schema and queries?

**Answer**:
Normalize the schema and index fields used in filters.

**Steps to Design**:

1. Use normalized tables for better data organization.
2. Create indexes on columns frequently used in search filters.
3. Use dynamic queries or a search engine like Elasticsearch for complex filters.

**Code Example**:

```sql
CREATE INDEX idx_product_category_price ON products(category, price);
SELECT * FROM products
WHERE category = 'Electronics' AND price BETWEEN 100 AND 500;
```

**Explanation**:

- Indexing reduces filter query times, and search engines can handle unstructured or full-text searches.

---

### Q33: You notice deadlocks in the database during high traffic. How would you detect and fix them?

**Answer**:
Deadlocks occur when two or more transactions block each other.

**Steps to Fix**:

1. **Detect**: Use database logs or tools (e.g., `` `SHOW ENGINE INNODB STATUS` `` in MySQL).
2. **Resolve**:
    - Break large transactions into smaller ones.
    - Use consistent ordering for accessing tables in queries.
    - Set a timeout for transactions.

**Code Example**:

```sql
SET innodb_lock_wait_timeout = 50; -- Set transaction timeout
```

**Explanation**:

- Analyzing deadlock logs helps identify conflicting transactions, and consistent ordering prevents circular waiting.

---

### Q34: Your database is running out of connections under load. How would you configure and tune connection pooling?

**Answer**:
Use connection pooling tools like HikariCP in Spring Boot to manage database connections efficiently.

**Steps to Configure**:

1. **Limit Connections**: Configure `maximumPoolSize` in HikariCP to prevent overloading.
2. **Timeout Settings**: Set connection and idle timeout values.
3. **Monitor Usage**: Use database monitoring tools to optimize pool size.

**Code Example**:

```properties
spring.datasource.hikari.maximum-pool-size=20
spring.datasource.hikari.connection-timeout=30000
spring.datasource.hikari.idle-timeout=60000
```

**Explanation**:

- Connection pooling reuses existing connections instead of creating new ones, saving time and resources.

---

### Q35: You need to migrate a production database to a new schema with minimal downtime. How would you plan this?

**Answer**:
Plan and execute a zero-downtime migration strategy.

**Steps to Migrate**:

1. **Versioning**: Use tools like Liquibase or Flyway for schema versioning.
2. **Blue-Green Deployment**: Deploy the new schema in parallel with the old one.
3. **Data Synchronization**: Migrate data incrementally using database triggers or scripts.

**Code Example**:

```yaml
# Flyway migration script
CREATE TABLE new_table (
    id SERIAL PRIMARY KEY,
    name VARCHAR(100)
);
```

**Explanation**:

- Versioning ensures schema changes are applied consistently, and blue-green deployment minimizes downtime.

---

### Q36: A table has a high number of updates and inserts, leading to degraded performance. How would you handle this?

**Answer**:
Optimize table structure and database settings to handle high write operations.

**Steps to Optimize**:

1. **Partitioning**: Split the table into smaller, manageable partitions.
2. **Index Management**: Remove unused indexes that slow down inserts and updates.
3. **Buffering**: Use in-memory buffers like Redis to handle frequent writes.

**Code Example**:

```sql
CREATE TABLE orders PARTITION BY RANGE (order_date) (
    PARTITION p2024 VALUES LESS THAN ('2024-01-01'),
    PARTITION p2025 VALUES LESS THAN ('2025-01-01')
);
```

**Explanation**:

- Partitioning reduces the scope of operations, improving query performance for high-transaction tables.

---

### Q37: Your application supports soft deletes but query performance is deteriorating. How would you resolve this?

**Answer**:
Use indexed flags and archival strategies.

**Steps to Optimize**:

1. **Index Soft Delete Flag**: Add an index to the `` `is_deleted` `` column.
2. **Archival**: Move deleted records to a separate archive table periodically.
3. **Filtered Indexes**: Create indexes that exclude soft-deleted records.

**Code Example**:

```sql
sql


CREATE INDEX idx_active_records ON users(is_deleted)
WHERE is_deleted = false;
```

**Explanation**:

- Indexing ensures filtering on `is_deleted` is efficient, and archival reduces table size.

---

## Q38: Multiple microservices need to access the same database but must not interfere with each other's data. How would you handle this?

**Answer**:
Isolate data for each microservice or use shared data carefully.

**Steps to Handle**:

1. **Schema Per Service**: Create separate schemas for each microservice.
2. **Access Control**: Use database roles and permissions.
3. **Event Sourcing**: Use an event-driven architecture for shared data access.

**Code Example**:

```sql
sql


CREATE SCHEMA service_a AUTHORIZATION user_a;
CREATE SCHEMA service_b AUTHORIZATION user_b;
```

**Explanation**:

- Isolating schemas avoids interference, and event sourcing ensures consistent updates across services.

---

## Q39: You need to maintain an audit trail of all data changes in a database. How would you implement this?

**Answer**:
Implement audit logging using triggers or ORM frameworks.

**Steps to Implement**:

1. **Database Triggers**: Use triggers to log changes to an audit table.
2. **Hibernate Envers**: Use Hibernate Envers for versioning and auditing.
3. **Custom Audit Tables**: Create tables to store audit logs.

**Code Example**:

```sql
CREATE TRIGGER audit_log AFTER UPDATE ON users
FOR EACH ROW
INSERT INTO audit_logs (user_id, old_value, new_value)
VALUES (OLD.id, OLD.name, NEW.name);
```

**Explanation**:

- Triggers capture changes automatically, and Envers provides ORM-based auditing.

---

## Q40: Your application is moving from a relational database to a NoSQL database. How would you handle this migration?

**Answer**:
Plan data model transformation and incremental migration.

**Steps to Migrate**:

1. **Data Model**: Map relational tables to NoSQL collections.
2. **Incremental Migration**: Migrate data in phases to ensure correctness.
3. **Testing**: Validate data consistency post-migration.

**Code Example**:

```json
{
  "user_id": 1,
  "name": "John Doe",
  "orders": [
    {"order_id": 101, "amount": 250},
    {"order_id": 102, "amount": 450}
  ]
}
```

**Explanation**:

- NoSQL databases use document or key-value structures, simplifying access patterns.

---

Let me know if you need further adjustments or clarifications!