Complete Database Systems Learning Path

> *A comprehensive guide from DBMS fundamentals to advanced concepts*

Table of Contents

1. DBMS Fundamentals

1.1 Core Database Concepts

Tables, Records, Fields, and Keys

A database is an organized collection of structured information, or data, stored electronically. The fundamental building blocks of a relational database include:

- **Table**: A collection of related data organized in rows and columns
- **Record/Row**: A single entry in a table that contains values for each field
- **Field/Column**: A category of information within a table
- **Key**: A special field used to identify, access, or establish relationships between records

Types of Keys:

- **Primary Key**: Uniquely identifies each record in a table
- **Foreign Key**: References a primary key in another table to establish a relationship
- **Composite Key**: Combination of two or more columns that uniquely identify a record
- **Candidate Key**: Field(s) that could serve as a primary key
- **Super Key**: Any set of fields that includes a candidate key
- **Surrogate Key**: An artificial key created solely for identification (e.g., auto-increment IDs)

Example Schema:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,  -- Primary key
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    email VARCHAR(100) UNIQUE,    -- Candidate key
    phone VARCHAR(20),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)  -- Foreign key
);
```

Relationships

Database relationships define how tables are related to each other:

- **One-to-One (1:1)**: Each record in Table A is related to exactly one record in Table B
- **One-to-Many (1:N)**: Each record in Table A can be related to many records in Table B
- **Many-to-Many (N:M)**: Records in Table A can relate to many records in Table B and vice versa (requires a junction table)

Example of a Many-to-Many Relationship:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL
);

CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    title VARCHAR(100) NOT NULL
);

-- Junction table for many-to-many relationship
CREATE TABLE enrollments (
    student_id INT,
    course_id INT,
    enrollment_date DATE NOT NULL,
    PRIMARY KEY (student_id, course_id),  -- Composite primary key
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

Interview Questions

Q: What is the difference between a primary key and a unique key?

A: Both primary keys and unique keys enforce uniqueness of column values. However, a primary key cannot contain NULL values, while a unique key can allow NULL values (typically only one NULL, as NULL != NULL in SQL). Additionally, a table can have only one primary key but multiple unique keys. Primary keys are also automatically indexed for performance.

Q: How would you design a database for a social media platform where users can follow each other?

A: This is a self-referential many-to-many relationship. I would create a users table and a separate junction table for followers:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50) UNIQUE,
    email VARCHAR(100) UNIQUE,
    -- other user attributes
);
```

```
CREATE TABLE followers (
    follower_id INT,
    followed_id INT,
    follow_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (follower_id, followed_id),
    FOREIGN KEY (follower_id) REFERENCES users(user_id),
    FOREIGN KEY (followed_id) REFERENCES users(user_id)
);
```

1.2 ACID Properties and Transactions

Transactions are logical units of work in a database. The ACID properties ensure that database transactions are processed reliably.

ACID Properties

- **Atomicity**: A transaction is treated as a single, indivisible unit that either completes entirely or fails entirely
- **Consistency**: A transaction brings the database from one valid state to another valid state
- **Isolation**: Concurrent transactions execute as if they were sequential
- **Durability**: Once a transaction is committed, it remains committed even in the case of system failure

Transaction Examples

Basic Transaction Syntax:

```
-- MySQL/PostgreSQL
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 456;
COMMIT;

-- SQL Server
BEGIN TRANSACTION;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 456;
COMMIT TRANSACTION;
```

Handling Transaction Errors:

```
BEGIN;
UPDATE accounts SET balance = balance - 100 WHERE account_id = 123;

-- Check if sufficient funds
IF (SELECT balance FROM accounts WHERE account_id = 123) < 0 THEN
    ROLLBACK;  -- Revert changes if balance would be negative
ELSE
    UPDATE accounts SET balance = balance + 100 WHERE account_id = 456;
    COMMIT;    -- Complete transaction if everything is ok
END IF;
```

Interview Questions

Q: Why is atomicity important in database transactions?

A: Atomicity ensures that if part of a transaction fails, the entire transaction fails and the database remains unchanged. This is critical for operations that require multiple related changes—like transferring money between accounts—where a partial update would cause data inconsistency. For example, if a bank transfer debits one account but fails to credit the other account, atomicity ensures the debit is rolled back so money doesn't disappear.

Q: How would you implement a transaction that must update inventory and create an order simultaneously?

A: I would wrap both operations in a single transaction to ensure that either both succeed or both fail:

```
BEGIN TRANSACTION;

-- First, check if there's enough inventory
DECLARE @available_quantity INT;
SELECT @available_quantity = quantity FROM inventory WHERE product_id = 101;

IF @available_quantity >= 5
BEGIN
    -- Update inventory
    UPDATE inventory
    SET quantity = quantity - 5
    WHERE product_id = 101;

    -- Create order
    INSERT INTO orders (customer_id, order_date, status)
    VALUES (1001, CURRENT_TIMESTAMP, 'PENDING');

    -- Get the newly created order ID
    DECLARE @new_order_id INT;
    SET @new_order_id = SCOPE_IDENTITY();

    -- Add order item
    INSERT INTO order_items (order_id, product_id, quantity, price)
    VALUES (@new_order_id, 101, 5, 19.99);

    COMMIT TRANSACTION;
END
ELSE
BEGIN
    -- Not enough inventory
    ROLLBACK TRANSACTION;
    -- Handle the error appropriately
END
```

1.3 Normalization and Denormalization

Database Normalization

Normalization is the process of organizing data to reduce redundancy and improve data integrity. It involves dividing larger tables into smaller ones and defining relationships between them.

First Normal Form (1NF):

- Each table must have a primary key
- Each column contains atomic (indivisible) values
- No repeating groups

Second Normal Form (2NF):

- Must be in 1NF
- All non-key attributes must depend on the entire primary key

Third Normal Form (3NF):

- Must be in 2NF
- No transitive dependencies (non-key attributes depend only on the primary key)

Boyce-Codd Normal Form (BCNF):

- Must be in 3NF
- For every dependency X → Y, X must be a superkey

Fourth Normal Form (4NF):

- Must be in BCNF
- No multi-valued dependencies

Fifth Normal Form (5NF):

- Must be in 4NF
- No join dependencies

Normalization Example

Unnormalized Table:

```
| order_id | customer_name | customer_email          | product_id | product_name |
product_category | quantity | price |
|----------|---------------|-------------------------|------------|--------------|-
-----------------|----------|-------|
| 1        | John Smith    | john.smith@example.com  | 101        | Keyboard     |
Electronics      | 1        | 49.99 |
| 1        | John Smith    | john.smith@example.com  | 102        | Mouse        |
Electronics      | 1        | 29.99 |
| 2        | Jane Doe      | jane.doe@example.com    | 102        | Mouse        |
Electronics      | 2        | 29.99 |
```

After Normalization (3NF):

```
-- Customers table
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL
);

-- Products table
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    category VARCHAR(50) NOT NULL,
    price DECIMAL(10, 2) NOT NULL
);

-- Orders table
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- Order Items table
CREATE TABLE order_items (
    order_id INT,
    product_id INT,
    quantity INT NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Denormalization

Denormalization is the process of adding redundant data to optimize read performance, often at the expense of write performance. It's typically used for:

- Read-heavy workloads
- Reporting and analytics
- Reducing complex joins
- Improving query response time

Denormalization Techniques:

1. **Materialized Views**: Precomputed result sets stored as tables
2. **Redundant Columns**: Copying frequently accessed columns across tables
3. **Pre-joined Tables**: Combining tables that are frequently joined
4. **Aggregate Tables**: Storing precomputed summary data

Denormalization Example:

```sql
-- Denormalized order table for reporting
CREATE TABLE order_report_view (
    order_id INT,
    order_date TIMESTAMP,
    customer_id INT,
    customer_name VARCHAR(100),
    customer_email VARCHAR(100),
    product_id INT,
    product_name VARCHAR(100),
    product_category VARCHAR(50),
    quantity INT,
    unit_price DECIMAL(10, 2),
    line_total DECIMAL(10, 2),
    PRIMARY KEY (order_id, product_id)
);
```

Interview Questions

Q: What are the trade-offs between normalization and denormalization?

A:

Normalization:

- Advantages: Reduces data redundancy, minimizes update anomalies, smaller tables, better integrity
- Disadvantages: Requires more joins, potentially slower read performance, more complex queries

Denormalization:

- Advantages: Faster query performance for read operations, fewer joins, simpler queries
- Disadvantages: Data redundancy, increased storage requirements, more complex updates, potential for inconsistency

The decision should be based on the specific workload: normalization is better for write-heavy applications with complex relationships, while denormalization works better for read-heavy applications and reporting.

Q: How would you decide when to denormalize a particular aspect of your database design?

A: I would consider denormalization when:

1. There's a significant read vs. write ratio imbalance (heavily read-oriented)
2. Performance analysis shows that joining tables is causing a bottleneck
3. Specific reports or queries are consistently slow despite indexing and optimization
4. The application requires near real-time responses for specific user-facing queries
5. The data doesn't change frequently, making maintenance of redundant data manageable

Before denormalizing, I would ensure:

- The benefit in read performance outweighs the added complexity in write operations
- There's a system in place to maintain consistency of the redundant data
- The space requirements are acceptable
- The business can tolerate potential inconsistencies during updates

1.4 Indexing Strategies

Indexes are data structures that improve the speed of data retrieval operations by providing quick lookup capabilities for the database engine.

Types of Indexes

1. **B-Tree Index**: The most common index type, balanced tree structure that maintains sorted data
2. **Hash Index**: Very fast for exact lookups, but not useful for range queries or sorting
3. **Bitmap Index**: Efficient for columns with low cardinality (few unique values)
4. **Full-Text Index**: Specialized for searching text content
5. **Spatial Index**: Optimized for geographic or geometric data
6. **Covering Index**: Includes all columns needed for a query
7. **Clustered Index**: Determines the physical order of data in a table
8. **Non-Clustered Index**: Creates a separate structure with pointers to the actual data

Creating Indexes

```
-- Basic index
CREATE INDEX idx_customer_last_name ON customers(last_name);

-- Unique index
CREATE UNIQUE INDEX idx_email ON customers(email);

-- Composite index (multiple columns)
CREATE INDEX idx_name ON customers(last_name, first_name);

-- Functional/Expression index
CREATE INDEX idx_lower_email ON customers(LOWER(email));

-- Partial/Filtered index (PostgreSQL)
CREATE INDEX idx_active_users ON users(last_login) WHERE status = 'active';
```

Index Usage Strategies

When to Index:

- Foreign key columns
- Columns frequently used in WHERE clauses
- Columns used in JOIN conditions
- Columns used in ORDER BY or GROUP BY
- Columns with high cardinality (many unique values)

When Not to Index:

- Small tables where full scans are fast
- Columns rarely used in queries
- Columns with low cardinality (few unique values)
- Frequently updated columns
- Columns with large data types

Query Execution Plans

Most relational databases provide tools to view the execution plan for a query, which helps understand how indexes are being used:

```
-- MySQL/MariaDB
EXPLAIN SELECT * FROM customers WHERE last_name = 'Smith';

-- PostgreSQL
EXPLAIN ANALYZE SELECT * FROM customers WHERE last_name = 'Smith';

-- SQL Server
SET SHOWPLAN_ALL ON;
GO
SELECT * FROM customers WHERE last_name = 'Smith';
GO
SET SHOWPLAN_ALL OFF;

-- Oracle
EXPLAIN PLAN FOR SELECT * FROM customers WHERE last_name = 'Smith';
SELECT * FROM TABLE(DBMS_XPLAN.DISPLAY);
```

Index Performance Considerations

1. **Index Selectivity**: The ratio of unique values to total rows (higher is better)
2. **Index Maintenance**: Indexes slow down INSERT, UPDATE, DELETE operations
3. **Index Size**: Indexes consume storage space and memory
4. **Index Fragmentation**: Over time, indexes can become fragmented, requiring maintenance
5. **Covering Indexes**: Include all columns needed by a query to avoid table lookups

Interview Questions

Q: How do you decide which columns to index in a database?

A: When deciding which columns to index, I consider:

1. Query patterns: Columns frequently used in WHERE clauses, JOIN conditions, and ORDER BY statements
2. Column cardinality: Columns with high cardinality (many unique values) benefit more from indexing
3. Data distribution: Columns with evenly distributed values work better with indexes
4. Write frequency: Less frequently updated columns are better candidates for indexing
5. Table size: Larger tables benefit more from proper indexing
6. Query execution plans: Using EXPLAIN to identify slow queries that could benefit from indexing

I also consider the trade-offs of adding indexes, as each index:

- Speeds up read operations but slows down write operations
- Consumes additional storage space
- Requires maintenance (especially with frequent updates)

Q: What is a covering index and when would you use one?

A: A covering index is an index that includes all the columns required by a query, allowing the database engine to fulfill the query using just the index without having to access the table data.

For example, if you frequently run:

```
SELECT first_name, last_name FROM customers WHERE status = 'active';
```

You could create a covering index:

```
CREATE INDEX idx_customer_covering ON customers(status, first_name, last_name);
```

Covering indexes provide significant performance benefits because:

1. They avoid the extra I/O operations required to access the actual table data
2. The index is typically much smaller than the full table, resulting in fewer disk reads
3. Index data is often better cached due to its smaller size

I would use covering indexes for frequently run queries that access a small subset of columns from large tables, especially in read-heavy workloads.

1.5 Query Processing and Optimization

Understanding how a database processes and optimizes queries is crucial for database performance tuning.

Query Processing Pipeline

1. **Parsing**: SQL statement is parsed into a syntax tree
2. **Validation**: Query is checked for semantic correctness
3. **Optimization**: Query optimizer creates an execution plan
4. **Execution**: The plan is executed to retrieve results
5. **Result Set Creation**: Results are assembled and returned

Query Optimization Techniques

Database Engine Optimizations:

1. **Statistics-Based Optimization**: Using statistics about data distribution
2. **Cost-Based Optimization**: Estimating cost of different execution plans
3. **Index Selection**: Choosing the most appropriate indexes
4. **Join Order Selection**: Determining the optimal order of table joins
5. **Join Algorithm Selection**: Hash join, nested loop join, merge join

Developer Optimizations:

1. **Query Rewriting**: Reformulating queries for better performance
2. **Proper Indexing**: Creating appropriate indexes for common queries
3. **Appropriate JOIN Types**: Using INNER, LEFT, RIGHT joins correctly
4. **Subquery Optimization**: Converting to JOINs when appropriate

5. **LIMIT/TOP Usage**: Restricting result set size
6. **Avoiding SELECT ***: Selecting only required columns

Query Analysis Tools

```
-- MySQL
EXPLAIN SELECT customers.*, orders.order_date
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.status = 'active';

-- PostgreSQL
EXPLAIN ANALYZE SELECT customers.*, orders.order_date
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.status = 'active';

-- SQL Server
SET STATISTICS IO ON;
SET STATISTICS TIME ON;
GO
SELECT customers.*, orders.order_date
FROM customers
JOIN orders ON customers.customer_id = orders.customer_id
WHERE customers.status = 'active';
GO
```

Common Query Performance Issues

1. **Missing Indexes**: Causing full table scans
2. **Non-SARGable Conditions**: Conditions that prevent index usage

   ```
   -- Non-SARGable: cannot use index on last_name
   WHERE UPPER(last_name) = 'SMITH'

   -- SARGable: can use index on last_name
   WHERE last_name = 'Smith'
   ```

3. **Cartesian Products**: Missing JOIN conditions
4. **Too Many Joins**: Complex queries joining many tables
5. **Large IN Lists**: Long lists of values in IN clauses
6. **Function Usage in WHERE**: Functions on indexed columns

Interview Questions

Q: What is the difference between WHERE and HAVING clauses?

A: Although both filter data, they serve different purposes and operate at different stages of query processing:

WHERE:

- Filters rows before any grouping occurs
- Applied directly to the table data
- Cannot reference aggregate functions
- Typically more efficient as it reduces the dataset early

HAVING:

- Filters groups after GROUP BY has been applied
- Applied to grouped results
- Can reference aggregate functions like COUNT, SUM, AVG
- Less efficient as filtering happens after grouping

Example:

```
-- Filter customers before grouping
SELECT city, COUNT(*) as customer_count
FROM customers
WHERE status = 'active'  -- Applied to individual rows
GROUP BY city
HAVING COUNT(*) > 10;    -- Applied to groups after aggregation
```

Q: How would you optimize a slow-performing query?

A: I would follow a systematic approach:

1. **Analyze the execution plan** using EXPLAIN or equivalent tools to identify bottlenecks:
    - Look for full table scans, inefficient joins, or missing indexes
2. **Check indexing**:
    - Add indexes for columns in WHERE, JOIN, ORDER BY, GROUP BY clauses
    - Consider composite or covering indexes for frequently run queries
3. **Rewrite the query**:
    - Simplify complex subqueries
    - Convert non-SARGable conditions to SARGable ones
    - Avoid SELECT * and request only needed columns
    - Use JOINs instead of correlated subqueries where appropriate
4. **Review data model**:
    - Consider denormalization for read-heavy scenarios
    - Check for proper normalization to avoid excessive joins
5. **Database-specific optimizations**:
    - Adjust database parameters (buffer sizes, memory allocation)

- Use materialized views or caching for expensive computations
- Consider partitioning for very large tables

6. **Application-level strategies**:
   - Implement connection pooling
   - Use prepared statements
   - Consider query pagination
7. **Hardware considerations**:
   - Evaluate if database server needs more CPU, memory, or faster storage

Example optimization:

```
-- Original slow query
SELECT c.*, o.order_date
FROM customers c, orders o
WHERE c.customer_id = o.customer_id
AND UPPER(c.status) = 'ACTIVE'
ORDER BY o.order_date DESC;

-- Optimized query
SELECT c.customer_id, c.name, c.email, o.order_date  -- Only needed columns
FROM customers c
INNER JOIN orders o ON c.customer_id = o.customer_id  -- Explicit join
WHERE c.status = 'active'  -- SARGable condition
ORDER BY o.order_date DESC
LIMIT 100;  -- Pagination if applicable
```

1.6 Concurrency Control Mechanisms

Concurrency control ensures that database transactions are executed in a way that preserves data consistency when multiple users or processes access the database simultaneously.

Concurrency Problems

1. **Lost Update**: When two transactions read and update the same data, and one overwrites the other's changes
2. **Dirty Read**: When a transaction reads data that has been modified by another transaction but not yet committed
3. **Non-Repeatable Read**: When a transaction reads the same data twice but gets different values
4. **Phantom Read**: When a transaction re-executes a query and finds new rows that match the query condition

Concurrency Control Methods

Locking Mechanisms:

1. **Shared Lock (S-Lock)**: Multiple transactions can read the same data simultaneously
2. **Exclusive Lock (X-Lock)**: Only one transaction can modify the data at a time

3. **Optimistic Locking**: Assumes conflicts are rare and checks for conflicts during commit
4. **Pessimistic Locking**: Assumes conflicts are likely and locks resources before accessing them
5. **Two-Phase Locking (2PL)**: Acquires all locks before releasing any locks

Lock Levels:

1. **Row-Level Locking**: Locks individual rows
2. **Page-Level Locking**: Locks database pages containing multiple rows
3. **Table-Level Locking**: Locks entire tables
4. **Database-Level Locking**: Locks the entire database

Other Concurrency Control Methods:

1. **Multiversion Concurrency Control (MVCC)**: Maintains multiple versions of data
2. **Timestamp Ordering**: Uses timestamps to determine transaction order
3. **Serialization Graph Check**: Ensures that transactions can be executed serially

Lock Commands Examples

```
-- MySQL explicit table locks
LOCK TABLES users READ;      -- Shared lock
LOCK TABLES users WRITE;     -- Exclusive lock
UNLOCK TABLES;               -- Release locks

-- PostgreSQL row-level locks
SELECT * FROM users WHERE user_id = 1 FOR UPDATE;  -- Exclusive lock
SELECT * FROM users WHERE user_id = 1 FOR SHARE;   -- Shared lock

-- SQL Server
BEGIN TRANSACTION;
SELECT * FROM users WITH (HOLDLOCK, ROWLOCK) WHERE user_id = 1;
-- Do work
COMMIT TRANSACTION;
```

Deadlocks

A deadlock occurs when two or more transactions are waiting indefinitely for one another to release locks.

Deadlock Example:

- Transaction A holds a lock on Resource 1 and requests a lock on Resource 2
- Transaction B holds a lock on Resource 2 and requests a lock on Resource 1
- Neither can proceed, resulting in a deadlock

Deadlock Resolution:

1. **Deadlock Detection**: Identify deadlocks by analyzing the wait-for graph
2. **Deadlock Prevention**: Design transactions to acquire resources in a consistent order
3. **Deadlock Avoidance**: Use timeouts to limit how long transactions wait for locks
4. **Deadlock Recovery**: Abort one transaction to allow others to proceed

Interview Questions

Q: What is MVCC and how does it help with database concurrency?

A: Multiversion Concurrency Control (MVCC) is a concurrency control method that allows multiple transactions to access the database simultaneously by creating different versions of database objects. Here's how it works and helps with concurrency:

1. When a transaction updates data, instead of overwriting the existing data, MVCC creates a new version of the data
2. Each transaction sees a snapshot of the database as it was at the beginning of the transaction
3. Read operations do not block write operations, and write operations do not block read operations

MVCC provides several benefits:

- Higher concurrency: Readers don't block writers and writers don't block readers
- Consistent reads: Each transaction sees a consistent snapshot of the database
- Reduced lock contention: Fewer locks are needed, improving performance
- No dirty reads: Transactions only see committed data (depending on isolation level)

Databases like PostgreSQL, Oracle, and MySQL's InnoDB engine implement MVCC, albeit with different approaches.

Q: How would you prevent deadlocks in a high-concurrency database application?

A: To prevent deadlocks in high-concurrency database applications, I would implement these strategies:

1. **Consistent resource acquisition order**:
    - Always acquire locks in the same order across all transactions
    - For example, if you need to update both orders and customers tables, always lock orders first, then customers
2. **Minimize transaction scope and duration**:
    - Keep transactions as short as possible
    - Only acquire locks when needed and release them promptly
    - Move read-only operations outside transaction blocks when possible
3. **Use appropriate isolation levels**:
    - Use the lowest isolation level that meets the application's consistency requirements
    - Consider READ COMMITTED for most operations instead of SERIALIZABLE
4. **Implement timeouts**:
    - Set lock timeout parameters to prevent indefinite waiting

```
-- SQL Server example
SET LOCK_TIMEOUT 5000;  -- 5 seconds
```

5. **Avoid user input within transactions**:

- Collect all necessary user input before beginning a transaction

6. **Use optimistic concurrency control** when appropriate:
   - Instead of locking, check if data has changed before updating

```
UPDATE users
SET last_login = CURRENT_TIMESTAMP
WHERE user_id = 123 AND last_updated = @original_timestamp;
```

7. **Consider application-level solutions**:
   - Implement a queuing system for high-contention operations
   - Use database connection pooling to limit concurrent connections

8. **Monitor and analyze deadlocks**:
   - Enable deadlock logging
   - Regularly review deadlock graphs to identify patterns

1.7 Database Security Principles

Database security involves protecting data from unauthorized access, corruption, or loss.

Authentication and Authorization

Authentication verifies who the user is, while **authorization** determines what they can do.

User Management:

```
-- Create user (MySQL)
CREATE USER 'app_user'@'localhost' IDENTIFIED BY 'secure_password';

-- Grant privileges (MySQL)
GRANT SELECT, INSERT, UPDATE ON my_database.* TO 'app_user'@'localhost';

-- Revoke privileges
REVOKE DELETE ON my_database.* FROM 'app_user'@'localhost';

-- Create role (PostgreSQL)
CREATE ROLE readonly;
GRANT CONNECT ON DATABASE my_database TO readonly;
GRANT USAGE ON SCHEMA public TO readonly;
GRANT SELECT ON ALL TABLES IN SCHEMA public TO readonly;

-- Assign role to user
GRANT readonly TO app_user;
```

Access Control Models

1. **Discretionary Access Control (DAC)**: Access rights managed by object owner
2. **Mandatory Access Control (MAC)**: System-enforced access based on sensitivity labels
3. **Role-Based Access Control (RBAC)**: Access rights assigned to roles, and users assigned to roles
4. **Attribute-Based Access Control (ABAC)**: Dynamic access based on user attributes, resource attributes, and environmental conditions

Data Encryption

1. **Transparent Data Encryption (TDE)**: Encrypts data files at rest
2. **Column-Level Encryption**: Encrypts specific columns containing sensitive data
3. **Client-Side Encryption**: Data encrypted before sending to the database
4. **Transport Layer Security (TLS)**: Secures data in transit

Encryption Examples:

```
-- Enable TLS for connections (MySQL)
ALTER USER 'app_user'@'localhost' REQUIRE SSL;

-- Column-level encryption (PostgreSQL with pgcrypto)
CREATE EXTENSION pgcrypto;

-- Insert encrypted data
INSERT INTO users (username, credit_card)
VALUES ('john_doe', pgp_sym_encrypt('1234-5678-9012-3456', 'encryption_key'));

-- Query encrypted data
SELECT username, pgp_sym_decrypt(credit_card::bytea, 'encryption_key')
FROM users;
```

SQL Injection Prevention

SQL injection is one of the most common database security vulnerabilities.

Vulnerable Code:

```
// DO NOT DO THIS
String query = "SELECT * FROM users WHERE username = '" + username + "'";
Statement stmt = connection.createStatement();
ResultSet rs = stmt.executeQuery(query);
```

Secure Code (Using Prepared Statements):

```
// DO THIS INSTEAD
String query = "SELECT * FROM users WHERE username = ?";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, username);
ResultSet rs = stmt.executeQuery();
```

Other SQL Injection Defenses:

1. **Input Validation**: Validate all user inputs against expected patterns
2. **Stored Procedures**: Use parameterized stored procedures
3. **Least Privilege**: Restrict database account permissions
4. **ORM Frameworks**: Use secure ORM frameworks that handle parameterization

Auditing and Monitoring

Maintaining an audit trail of database activity is crucial for security.

Auditing Setup:

```
-- Oracle auditing
AUDIT SELECT, UPDATE, DELETE ON employees;

-- SQL Server auditing
CREATE SERVER AUDIT DataAudit
TO FILE (FILEPATH = 'C:\Audits\');

CREATE DATABASE AUDIT SPECIFICATION UserTableAudit
FOR SERVER AUDIT DataAudit
ADD (SELECT, INSERT, UPDATE, DELETE ON users BY dbo);
```

Other Security Best Practices:

1. **Regular Backups**: Maintain regular backups with tested restoration procedures
2. **Security Patching**: Keep database software updated with security patches
3. **Network Security**: Use firewalls and network segmentation
4. **Data Masking**: Mask sensitive data in non-production environments
5. **Security Assessments**: Conduct regular security assessments and penetration testing

Interview Questions

Q: How would you secure a database containing sensitive customer information?

A: Securing a database with sensitive customer information requires a multi-layered approach:

1. **Access Control and Authentication**:
   - Implement strong password policies
   - Use multi-factor authentication for database access
   - Apply the principle of least privilege (users only get permissions they need)
   - Implement role-based access control
   - Regularly audit and review user permissions
2. **Data Encryption**:
   - Enable transparent data encryption (TDE) for data at rest
   - Use column-level encryption for PII (Personally Identifiable Information)
   - Secure data in transit using TLS/SSL

    - Consider client-side encryption for highly sensitive data
    - Properly manage encryption keys

3. **Protection Against SQL Injection**:

    - Use parameterized queries or prepared statements
    - Implement input validation
    - Consider using ORM frameworks with built-in protection
    - Use stored procedures where appropriate

4. **Auditing and Monitoring**:

    - Enable database activity monitoring
    - Track and alert on suspicious access patterns
    - Maintain comprehensive audit logs
    - Implement a SIEM (Security Information and Event Management) solution
    - Set up alerts for unusual query patterns or data access

5. **Infrastructure Security**:

    - Place databases behind firewalls
    - Implement network segmentation
    - Disable unused services and ports
    - Keep database software patched and updated
    - Use database proxies or connection pooling

6. **Operational Security**:

    - Regular security assessments and penetration testing
    - Data masking in non-production environments
    - Secure backup and recovery procedures
    - Implement a robust disaster recovery plan
    - Employee security awareness training

Q: What is the difference between encryption and hashing, and when would you use each in a database?

A: Encryption and hashing are both cryptographic techniques but serve different purposes:

Encryption:

- Bidirectional process: data can be encrypted and decrypted with the right key
- Preserves data size or increases it slightly
- Used when original data needs to be retrieved

Hashing:

- One-way function: original data cannot be retrieved from the hash
- Output has fixed length regardless of input size
- Used when original data doesn't need to be retrieved

When to use each in a database:

Use Encryption For:

- Credit card numbers, which need to be processed for transactions
- Personal identifiable information that needs to be displayed occasionally
- Medical records that need to be accessed in their original form
- Communication content that needs to remain confidential but accessible

Implementation example:

```
-- Encrypting credit card info (PostgreSQL)
INSERT INTO payments (user_id, credit_card)
VALUES (1001, pgp_sym_encrypt('4111-1111-1111-1111', 'encryption_key'));

-- Retrieving the encrypted data
SELECT user_id, pgp_sym_decrypt(credit_card::bytea, 'encryption_key')
FROM payments;
```

Use Hashing For:

- Passwords, which should never be stored in plaintext
- Data integrity verification
- Checking if two values match without revealing the values
- Creating unique identifiers for data

Implementation example:

```
-- Storing a password hash (PostgreSQL)
INSERT INTO users (username, password_hash)
VALUES ('john_doe', crypt('mysecretpassword', gen_salt('bf')));

-- Verifying a password
SELECT * FROM users
WHERE username = 'john_doe'
  AND password_hash = crypt('enteredpassword', password_hash);
```

Key considerations:

- Always salt password hashes to prevent rainbow table attacks
- Use industry-standard algorithms (bcrypt, Argon2 for passwords; AES, RSA for encryption)
- Manage encryption keys securely, separate from the encrypted data
- For regulatory compliance (like GDPR), consider if encryption allows for data erasure requirements

2. Comparative Analysis of Database Types

2.1 Relational vs. Non-Relational Databases

Relational Databases (RDBMS)

Characteristics:

- Based on relational model with tables, rows, and columns

- Strong schema enforcement
- ACID compliance
- SQL query language
- Established technology with mature tools and support

Common Examples:

- MySQL/MariaDB
- PostgreSQL
- Oracle Database
- Microsoft SQL Server
- IBM Db2

Strengths:

- Data integrity and consistency
- Complex queries and reporting
- Transactions and ACID compliance
- Well-established standards and tools
- Normalization to reduce redundancy
- Strong security features

Weaknesses:

- Rigid schema makes changes difficult
- Scaling challenges (primarily vertical)
- Performance can degrade with large volumes of data
- Less suitable for unstructured or semi-structured data
- JOINs can become expensive at scale

Non-Relational Databases (NoSQL)

Characteristics:

- Various data models beyond tables
- Schema flexibility or schema-less design
- Distributed architecture
- Designed for scalability
- Various query mechanisms (not just SQL)

Common Examples:

- MongoDB (document)
- Redis (key-value)
- Cassandra (wide-column)
- Neo4j (graph)
- Elasticsearch (search)

Strengths:

- Horizontal scalability
- Schema flexibility for evolving data

- High throughput for specific use cases
- Better handling of unstructured/semi-structured data
- Distributed architecture for fault tolerance
- Specific optimizations for particular data access patterns

Weaknesses:

- Often sacrifice ACID for performance
- Limited support for complex queries (depends on type)
- Less mature tooling in some cases
- Varied query languages (less standardization)
- Less emphasis on referential integrity
- Potential for data duplication

Side-by-Side Comparison

Aspect	Relational (RDBMS)	Non-Relational (NoSQL)
Data Model	Tables with rows and columns	Various: document, key-value, column, graph
Schema	Rigid, predefined	Flexible, dynamic, or schemaless
Scaling	Primarily vertical, with complexity for horizontal	Primarily horizontal, designed for distribution
Query Language	SQL (standardized)	Database-specific languages or APIs
Transactions	Strong ACID support	Varies: some offer ACID, others eventual consistency
Relationships	Foreign keys, JOINs	Varies: embedded documents, references, or graph relationships
Use Cases	Financial systems, ERP, CRM, traditional applications	Big data, real-time web apps, content management, IoT
Data Integrity	Strong enforcement via constraints	Often application-enforced
Maturity	High, decades of development	Varies by system, generally newer

Interview Questions

Q: When would you choose a NoSQL database over a relational database?

A: I would consider using a NoSQL database over a relational database in the following scenarios:

1. **When dealing with unstructured or semi-structured data**:
    - Document databases like MongoDB excel at storing JSON-like data with varying structure
    - When the schema is expected to evolve frequently and unpredictably
2. **For high write throughput requirements**:

    - Key-value stores like Redis or DynamoDB for session stores, caching, or real-time analytics
    - When write performance is more critical than complex querying capability

3. **When horizontal scalability is a primary concern**:

    - Distributed NoSQL databases like Cassandra or MongoDB for applications expecting massive growth
    - When data volume might exceed what a single server can reasonably handle

4. **For specific data access patterns**:

    - Graph databases like Neo4j for highly connected data with complex relationships
    - Time-series databases for sensor data, metrics, or event tracking
    - Search engines like Elasticsearch for full-text search capabilities

5. **For globally distributed applications**:

    - Databases with multi-region capabilities when low latency across geographic regions is required
    - When the application needs to function even during network partitions

6. **When ACID transactions aren't critical**:

    - Applications that can tolerate eventual consistency
    - Systems where high availability and partition tolerance are more important than consistency

Real-world example: A content management system for a media company might use MongoDB because:

- Content structure varies by type (articles, videos, podcasts)
- Schema evolves as new content types are added
- Read performance needs to scale for traffic spikes
- The application requires high write throughput for user comments
- Complex relationships can be modeled with embedded documents or references

However, I would still use a relational database for:

- Financial systems requiring ACID transactions
- Applications with complex reporting needs
- Systems with highly structured data and stable schemas
- When data integrity and consistency are non-negotiable requirements

Q: How do relational and non-relational databases differ in terms of scaling?

A: Relational and non-relational databases employ fundamentally different scaling approaches:

Relational Database Scaling:

1. **Vertical Scaling (Scale Up)**:

    - Adding more resources (CPU, RAM, faster storage) to a single server
    - Easier to implement but has physical hardware limitations
    - Eventually hits a ceiling regardless of hardware investment
    - Doesn't improve fault tolerance

2. **Limited Horizontal Scaling Options**:

    - **Replication**: Master-slave setups where slaves handle read operations

- **Sharding**: Splitting data across multiple instances by some partition key
- **Read-Write Splitting**: Directing reads to replicas, writes to primary

3. **Scaling Challenges**:

- JOINs across sharded data become complex and expensive
- Maintaining ACID properties across distributed systems is difficult
- Scaling often requires application changes
- Cross-shard transactions are particularly challenging

4. **Tools and Approaches**:

- Database-specific clustering solutions (Oracle RAC, MySQL Cluster)
- Proxy layers (ProxySQL, MySQL Router)
- Database-specific sharding implementations

Non-Relational Database Scaling:

1. **Horizontal Scaling (Scale Out)**:

- Built from the ground up for distribution across many nodes
- Can theoretically scale linearly by adding more commodity servers
- Often incorporates auto-sharding capabilities
- Designed with network partitioning in mind (CAP theorem trade-offs)

2. **Data Distribution Methods**:

- **Automatic Sharding**: Data automatically distributed based on shard keys
- **Replication**: Built-in replication for redundancy and performance
- **Masterless Design**: Some systems use peer-to-peer architecture with no single point of failure

3. **Scaling Advantages**:

- No need for complex JOINs that don't scale well
- Simplified data model built for distribution
- Data locality can be optimized for access patterns
- Better fault tolerance through redundancy

4. **Database-Specific Implementations**:

- **MongoDB**: Replica sets and automatic sharding
- **Cassandra**: Ring architecture with no single point of failure
- **DynamoDB**: Managed service with automatic scaling
- **Redis Cluster**: Hash slot-based sharding

Real-World Example:

Consider a social media application that needs to store user posts and scale to millions of users:

- **Relational Approach**:

- Initial scale with vertical scaling
- Eventually implement read replicas for read scaling
- Complex sharding scheme by user_id requires application changes

    - Cross-shard JOINs become problematic for features like "posts from friends"

- **NoSQL Approach**:
    - Start with a document database like MongoDB
    - Store posts in a collection sharded by user_id
    - Automatically scale by adding more shards as data grows
    - Denormalize data to avoid complex JOINs
    - Built-in replication for high availability

Hybrid Approaches:

Modern systems often take hybrid approaches:

- Using relational databases for transactional data
- NoSQL for specific high-scale components
- Caching layers with Redis or Memcached
- Event sourcing with message queues to decouple systems

The key is aligning the scaling strategy with the application's specific requirements for consistency, availability, and partition tolerance.

2.2 Choosing the Right Database

Selecting the right database technology depends on evaluating many factors related to your application's requirements.

Key Decision Factors

1. **Data Structure and Complexity**
    - Structured data with relationships → Relational databases
    - Unstructured or semi-structured data → Document databases
    - Simple key-value data → Key-value stores
    - Highly connected data → Graph databases
    - Time-based metrics → Time-series databases
2. **Scalability Requirements**
    - Vertical scaling needs → Typically relational databases
    - Horizontal scaling needs → Distributed NoSQL systems
    - Read-heavy workloads → Consider caching or read replicas
    - Write-heavy workloads → Sharded systems or log-structured databases
3. **Consistency Requirements**
    - Strong consistency needs → Relational or CP NoSQL systems
    - Eventual consistency acceptable → AP NoSQL systems
    - ACID transactions required → Relational or ACID-compliant NoSQL
4. **Query Patterns**
    - Complex joins and aggregations → Relational databases

- Simple lookups by key → Key-value stores
- Range queries → B-tree based databases
- Full-text search → Search engines (Elasticsearch)
- Graph traversals → Graph databases

5. **Performance Requirements**

- Low latency needs → In-memory databases
- High throughput → Distributed systems
- Batch processing → Data warehouses
- Real-time analytics → Column stores or specialized OLAP systems

6. **Operational Factors**

- Team expertise → Familiar technologies may reduce risks
- Budget constraints → Open source vs. commercial solutions
- Administrative overhead → Managed services vs. self-hosted
- Ecosystem and tooling → Maturity of surrounding tools

Decision Framework

When selecting a database, follow this process:

1. **Identify Requirements**:

- Document data model characteristics
- Quantify expected data volume and growth
- Specify query patterns and access frequency
- Define consistency and availability needs
- Establish performance SLAs

2. **Evaluate Options**:

- Create a shortlist based on primary requirements
- Analyze strengths and weaknesses for your use case
- Consider hybrid approaches when appropriate

3. **Test and Validate**:

- Benchmark with representative workloads
- Test failure scenarios and recovery procedures
- Evaluate operational complexity

4. **Plan for Future**:

- Consider future scaling needs
- Evaluate migration difficulty if needs change
- Assess vendor lock-in risks

Database Selection Matrix

Database Type	Best For	Consider When	Examples
Relational	Structured data with relationships; ACID transactions; Complex queries	Data structure is stable; Consistency is critical; Complex reporting needed	PostgreSQL, MySQL, Oracle, SQL Server
Document	Semi-structured data; Rapid development; Evolving schemas	Schema flexibility needed; Document-oriented data; Moderate relationship complexity	MongoDB, Couchbase, Azure Cosmos DB
Key-Value	Simple data structures; High throughput; Caching	Simple data access patterns; Extreme performance needed; Caching layer	Redis, DynamoDB, Riak
Wide-Column	Big data; Time-series; High write throughput	Large data with known access patterns; High write volumes; Analytics	Cassandra, HBase, Google Bigtable
Graph	Highly connected data; Relationship-focused queries	Data relationships are the primary value; Complex traversals needed	Neo4j, Amazon Neptune, JanusGraph
Time-Series	Metrics; Monitoring; IoT data	Time is the primary axis; High ingest rates; Retention policies	InfluxDB, TimescaleDB, Prometheus
Search	Full-text search; Log analysis; Faceted search	Text-heavy applications; Need for complex text queries; Analytics on text	Elasticsearch, Solr, Algolia

Interview Questions

Q: For a social media application with features similar to Twitter, what database choices would you make and why?

A: For a Twitter-like social media application, I would recommend a polyglot persistence architecture with different database systems handling specific aspects of the application:

1. **User Profiles and Authentication**:
    - **Database Choice**: Relational database (PostgreSQL)
    - **Rationale**:
        - Structured data with clear relationships
        - ACID transactions important for account operations
        - Strong consistency required for authentication
        - Relatively low volume with predictable growth
2. **Posts/Tweets Storage**:
    - **Database Choice**: Document database (MongoDB)
    - **Rationale**:
        - Semi-structured content with varying attributes (text, media, links)
        - High write throughput for post creation
        - Sharding capability for horizontal scaling
        - Flexible schema for evolving feature set (new post types, attributes)

        - Ability to embed engagement data (likes, shares)

3. **Social Graph (Following/Followers)**:
    - **Database Choice**: Graph database (Neo4j)
    - **Rationale**:
        - Optimized for relationship traversals
        - Efficient for queries like "friends of friends" or "common followers"
        - Natural fit for recommendation algorithms
        - Performance advantage for deep relationship queries

4. **Timeline Service**:
    - **Database Choice**: In-memory key-value store (Redis)
    - **Rationale**:
        - Extremely fast reads for timeline generation
        - Built-in sorted sets perfect for time-ordered data
        - Pub/sub capabilities for real-time updates
        - Can serve as a cache layer over the post storage

5. **Search Functionality**:
    - **Database Choice**: Search engine (Elasticsearch)
    - **Rationale**:
        - Optimized for full-text search across posts
        - Support for faceted search and filtering
        - Good for hashtag and mention indexing
        - Analytics capabilities for trending topics

6. **Analytics and Metrics**:
    - **Database Choice**: Columnar database (ClickHouse)
    - **Rationale**:
        - Optimized for analytical queries
        - High compression for historical data
        - Fast aggregations for metrics dashboards
        - Ability to handle high-volume event data

Data Flow Example:

- When a user posts a tweet:
    1. Store in MongoDB for persistence
    2. Update Redis for timeline delivery
    3. Index in Elasticsearch for searchability
    4. Record engagement events in ClickHouse
    5. Update graph relationships in Neo4j if mentions occur

Implementation Considerations:

- Use message queues (Kafka/RabbitMQ) for asynchronous processing
- Implement eventual consistency patterns where appropriate
- Consider replication strategies for global availability

- Cache hot data paths aggressively

This polyglot approach leverages the strengths of each database type rather than forcing a single database to handle all workloads suboptimally.

Q: How would you design a database system for an e-commerce platform that needs to handle seasonal traffic spikes?

A: For an e-commerce platform with seasonal traffic spikes, I would design a database architecture that prioritizes scalability, availability, and performance while maintaining data consistency where critical:

1. **Product Catalog**:
    - **Database**: Document database (MongoDB)
    - **Design Considerations**:
        - Products have varying attributes across categories
        - Read-heavy workload (many more product views than updates)
        - Implement read replicas for scaling during high traffic
        - Denormalize category and brand information for faster queries
        - Shard by product category for horizontal scaling
2. **User Profiles and Authentication**:
    - **Database**: Relational database (PostgreSQL)
    - **Design Considerations**:
        - ACID transactions for account operations
        - User data changes infrequently relative to catalog browsing
        - Sensitive data requiring strict access controls
        - Vertical scaling usually sufficient for this component
3. **Shopping Cart and Checkout**:
    - **Primary Database**: In-memory database with persistence (Redis)
    - **Secondary Database**: Relational database (PostgreSQL) for completed orders
    - **Design Considerations**:
        - Cart operations need to be extremely fast
        - Temporary data perfect for Redis expiration features
        - Ability to handle cart abandonment with TTL
        - Move to relational storage only upon order completion
        - Redis clustering for horizontal scaling during traffic spikes
4. **Inventory Management**:
    - **Database**: Relational database with strong consistency (PostgreSQL)
    - **Design Considerations**:
        - Critical for maintaining inventory accuracy
        - Implement row-level locking for inventory updates
        - Use connection pooling to handle concurrent sessions
        - Consider read replicas for inventory display vs. reservation
        - Implement inventory caching with short TTL for high-traffic products
5. **Order Processing**:

- **Database**: Relational database (PostgreSQL) with queue system (Kafka)
- **Design Considerations**:
  - ACID transactions for financial records
  - Implement event sourcing for order state changes
  - Use Kafka for processing backpressure during spikes
  - Consider time-partitioned tables for historical orders
  - Vertical scaling for transactional aspects, horizontal for processing

6. **Product Search and Recommendations**:

- **Database**: Search engine (Elasticsearch)
- **Design Considerations**:
  - Optimized for full-text search and faceted filtering
  - Can handle complex queries for recommendations
  - Scalable for high concurrent search volumes
  - Cluster for horizontal scaling during traffic spikes

7. **Analytics and Reporting**:

- **Database**: Data warehouse (Snowflake/Redshift) with ETL pipeline
- **Design Considerations**:
  - Separate from operational databases
  - Scheduled batch processing for reports
  - Star schema for efficient analytics queries
  - Consider throttling during peak traffic periods

Scaling Strategies for Traffic Spikes:

1. **Read Scalability**:

- Implement CDN for static product images and descriptions
- Use read replicas for read-heavy components
- Aggressive caching with Redis/Memcached for hot products
- Consider eventual consistency where appropriate

2. **Write Scalability**:

- Queue non-critical writes during peak periods
- Implement command-query responsibility segregation (CQRS)
- Use database proxy layers (ProxySQL, PgBouncer) for connection management
- Shard write-heavy components like product reviews

3. **Elasticity Approach**:

- Use managed database services with auto-scaling capabilities
- Implement predictive scaling based on historical patterns
- Design for database instance provisioning ahead of known traffic events
- Consider serverless database options for extreme elasticity

4. **Resilience Considerations**:

- Implement circuit breakers for database connections

    - Design graceful degradation of non-critical features
    - Geographically distributed database instances
    - Multi-region setup for disaster recovery

By combining different database types optimized for specific workloads, implementing appropriate scaling strategies, and preparing for seasonal spikes, this architecture can maintain performance under varying load conditions while ensuring data consistency for critical operations.

2.3 Scaling Strategies

Scaling a database involves increasing its capacity to handle growing workloads, whether in terms of data volume, query throughput, or concurrent connections.

Vertical Scaling (Scale Up)

Vertical scaling involves adding more resources to the existing database server.

Techniques:

- Adding more CPU cores
- Increasing RAM
- Using faster storage (SSDs, NVMe)
- Upgrading network interfaces

Advantages:

- Simpler implementation (no application changes needed)
- Maintains single-instance simplicity
- Good for moderate growth
- Fewer operational complexities

Disadvantages:

- Physical hardware limits
- Expensive as you approach high-end hardware
- Single point of failure
- Downtime during upgrades

Example Implementation:

- Upgrading a MySQL server from 8 cores/32GB RAM to 16 cores/128GB RAM
- Moving from SATA SSDs to NVMe storage
- Adding more buffer pool memory for InnoDB

Horizontal Scaling (Scale Out)

Horizontal scaling involves distributing the database across multiple servers.

Techniques:

1. **Replication**:
    - Master-slave (primary-replica) replication

- Multi-master replication
- Read replicas for read scaling

2. **Sharding/Partitioning**:
   - Range-based sharding
   - Hash-based sharding
   - Directory-based sharding
   - Geographic sharding

3. **Distributed Database Systems**:
   - Natively distributed databases (Cassandra, CockroachDB)
   - Database clusters (Galera, Vitess)
   - NoSQL distributed systems

Advantages:

- Theoretically unlimited scaling potential
- Better fault tolerance and availability
- Can use commodity hardware
- Geographical distribution possibilities

Disadvantages:

- Increased complexity
- Potential data consistency challenges
- Application changes often required
- More complex management and monitoring

Replication Strategies

Master-Slave Replication:

```
[Master DB] --writes--> [Slave DB 1]
          \--writes--> [Slave DB 2]
          \--writes--> [Slave DB 3]
```

Example Configuration (MySQL):

```
# Master Configuration
server-id = 1
log_bin = /var/log/mysql/mysql-bin.log
binlog_format = ROW

# Slave Configuration
server-id = 2
relay_log = /var/log/mysql/mysql-relay-bin.log
read_only = ON
```

Multi-Master Replication:

```
[Master DB 1] <---> [Master DB 2]
```

Example Configuration (MySQL Group Replication):

```
# Group Replication settings
plugin_load = 'group_replication.so'
group_replication_group_name = "aaaaaaaa-bbbb-cccc-dddd-eeeeeeeeeeee"
group_replication_start_on_boot = ON
group_replication_local_address = "server1:33061"
group_replication_group_seeds = "server1:33061,server2:33061"
group_replication_bootstrap_group = ON
```

Sharding Strategies

Range-Based Sharding:

- Divide data based on ranges of a key
- Example: Customers A-M on Shard 1, N-Z on Shard 2

Hash-Based Sharding:

- Use a hash function to determine which shard contains the data
- Example: `shard_id = hash(customer_id) % num_shards`

Consistent Hashing:

- Special hashing approach that minimizes redistribution when adding/removing shards
- Used in systems like Cassandra and DynamoDB

Example Sharding Implementation:

```
# Simple hash-based sharding in Python
def get_shard_id(key, num_shards):
    return hash(key) % num_shards

# Connect to the appropriate shard
def get_connection(key):
    shard_id = get_shard_id(key, NUM_SHARDS)
    return shard_connections[shard_id]

# Example usage
user_id = "user123"
conn = get_connection(user_id)
result = conn.execute("SELECT * FROM users WHERE id = ?", (user_id,))
```

Database-Specific Scaling Solutions

MySQL Scaling Options:

- MySQL Replication for read scaling
- MySQL Cluster for distributed storage
- ProxySQL for query routing
- Vitess for horizontal sharding

PostgreSQL Scaling Options:

- PostgreSQL streaming replication
- Citus for distributed PostgreSQL
- PgBouncer for connection pooling
- TimescaleDB for time-series scaling

MongoDB Scaling Options:

- Replica sets for high availability
- Automatic sharding
- Zone sharding for geographic distribution
- Read preferences for query routing

Cassandra Scaling Options:

- Masterless architecture (peer-to-peer)
- Built-in data distribution with consistent hashing
- Tunable consistency levels
- Multi-datacenter replication

Caching Strategies

Caching can significantly improve database scalability by reducing the load on the database.

Cache Levels:

1. **Application-level cache**: In-memory caching within the application
2. **Distributed cache**: Redis, Memcached
3. **Database query cache**: Built-in database caching
4. **Result cache**: Caching of computed results
5. **CDN**: For static content

Caching Patterns:

- Cache-aside (lazy loading)
- Write-through cache
- Write-behind cache
- Refresh-ahead cache

Example Redis Caching Implementation:

```
import redis
import json
```

```
# Connect to Redis
r = redis.Redis(host='localhost', port=6379, db=0)

def get_user(user_id):
    # Try to get from cache first
    cached_user = r.get(f"user:{user_id}")

    if cached_user:
        return json.loads(cached_user)

    # If not in cache, get from database
    user = database.query(f"SELECT * FROM users WHERE id = {user_id}")

    # Store in cache for future (with 1-hour expiration)
    r.setex(f"user:{user_id}", 3600, json.dumps(user))

    return user
```

Interview Questions

Q: How would you scale a database system that has reached its vertical scaling limits?

A: When a database system has reached its vertical scaling limits, I would implement a comprehensive horizontal scaling strategy following these steps:

1. **Analyze the Current Bottlenecks**:
    - Determine whether the limitation is read-heavy, write-heavy, or storage capacity
    - Profile queries to identify specific performance issues
    - Understand access patterns and data distribution
2. **Implement Read Scaling First**:
    - Set up read replicas to distribute read queries
    - Configure the application to route read vs. write traffic appropriately
    - Use asynchronous replication to minimize impact on the primary server
    - Example implementation for MySQL:

```
-- On primary server
CREATE USER 'replication_user'@'%' IDENTIFIED BY 'password';
GRANT REPLICATION SLAVE ON *.* TO 'replication_user'@'%';

-- On replica server
CHANGE MASTER TO
  MASTER_HOST='primary_server_ip',
  MASTER_USER='replication_user',
  MASTER_PASSWORD='password',
  MASTER_LOG_FILE='mysql-bin.000001',
  MASTER_LOG_POS=123;
START SLAVE;
```

3. **Implement Caching Strategy**:
   - Add Redis or Memcached layer for frequently accessed data
   - Implement cache invalidation strategies
   - Consider application-level query result caching

```
# Application-level caching with Redis
def get_product(product_id):
    cache_key = f"product:{product_id}"
    cached_data = redis_client.get(cache_key)

    if cached_data:
        return json.loads(cached_data)

    # Cache miss - get from database
    product = db.query(f"SELECT * FROM products WHERE id = {product_id}")
    # Cache for 10 minutes
    redis_client.setex(cache_key, 600, json.dumps(product))
    return product
```

4. **Implement Database Sharding**:
   - Choose appropriate sharding key based on access patterns
   - Implement sharding logic in the application layer or with a middleware solution
   - Examples:
     - **Application-level sharding**:

```
def get_shard(customer_id):
    return customer_id % SHARD_COUNT

def execute_query(customer_id, query, params):
    shard_id = get_shard(customer_id)
    connection = shard_connections[shard_id]
    return connection.execute(query, params)
```

     - **Using a specialized sharding solution like Vitess for MySQL**

5. **Implement Connection Pooling**:
   - Add connection pooling to manage database connections efficiently
   - Configure optimal pool size based on workload characteristics
   - Example using PgBouncer for PostgreSQL:

```
[databases]
mydb = host=127.0.0.1 port=5432 dbname=mydb

[pgbouncer]
listen_port = 6432
listen_addr = *
auth_type = md5
auth_file = userlist.txt
pool_mode = transaction
max_client_conn = 1000
default_pool_size = 100
```

6. **Consider Database-Specific Distributed Solutions**:
   - For MySQL: Vitess, Galera Cluster, or MySQL Cluster
   - For PostgreSQL: Citus, PostgreSQL with logical replication
   - For MongoDB: Built-in sharding and replica sets
   - For completely distributed approach: CockroachDB, Amazon Aurora, or Google Spanner

7. **Implement a Data Access Layer**:
   - Abstract database access to hide sharding complexity
   - Implement query routing logic
   - Add retry mechanisms and circuit breakers for resilience

```
public class DataAccessLayer {
    private ShardManager shardManager;
    private ConnectionPoolManager poolManager;

    public <T> T executeQuery(String userId, String query, Object... params) {
        int shardId = shardManager.getShardForUser(userId);
        Connection conn = poolManager.getConnection(shardId);
        try {
            return executeWithRetry(conn, query, params);
        } finally {
            conn.release();
        }
    }

    private <T> T executeWithRetry(Connection conn, String query, Object...
params) {
        // Implementation with retry logic
    }
}
```

8. **Geographic Distribution**:
   - Implement multi-region deployment for global applications
   - Consider active-active or active-passive setups depending on consistency requirements
   - Use data locality to reduce latency for users

9. **Monitoring and Maintenance**:
    - Implement comprehensive monitoring across all database instances
    - Automate shard balancing and data redistribution
    - Plan for adding/removing shards as the system grows
10. **Data Consistency Strategy**:
    - Define clear consistency requirements for different operations
    - Implement distributed transactions where necessary
    - Consider eventual consistency models for non-critical operations

The specific implementation would depend on the database technology and application requirements, but this approach provides a comprehensive roadmap for transitioning from vertical to horizontal scaling.

Q: What are the challenges of implementing database sharding and how would you address them?

A: Implementing database sharding comes with several significant challenges. Here's how I would address each one:

1. **Challenge: Data Distribution and Hotspots**

   When data is unevenly distributed across shards, some shards can become hotspots, causing performance bottlenecks.

   Solution:
   - Implement consistent hashing with virtual nodes to distribute data more evenly
   - Choose sharding keys that provide natural distribution (e.g., user_id rather than country_code)
   - Monitor shard performance and implement dynamic rebalancing when hotspots are detected
   - Consider compound sharding keys for more granular distribution

   Example:

```
// Consistent hashing with virtual nodes
public class ConsistentHashShardManager {
    private final int VIRTUAL_NODE_COUNT = 256;
    private final TreeMap<Long, Integer> ring = new TreeMap<>();

    public ConsistentHashShardManager(int shardCount) {
        // Add shards to the ring with virtual nodes
        for (int i = 0; i < shardCount; i++) {
            for (int v = 0; v < VIRTUAL_NODE_COUNT; v++) {
                ring.put(hash("shard-" + i + "-vnode-" + v), i);
            }
        }
    }

    public int getShardForKey(String key) {
        if (ring.isEmpty()) {
            return 0;
        }

        long hash = hash(key);
```

```
        if (!ring.containsKey(hash)) {
            Map.Entry<Long, Integer> entry = ring.ceilingEntry(hash);
            if (entry == null) {
                entry = ring.firstEntry();
            }
            return entry.getValue();
        }
        return ring.get(hash);
    }

    private long hash(String key) {
        // Consistent hash implementation
    }
}
```

2. **Challenge: Cross-Shard Queries and Joins**

   When data is distributed across multiple shards, queries that span shards become complex and inefficient.

   Solution:

   - Design sharding schema to minimize cross-shard queries (data colocation)
   - Implement scatter-gather query pattern for unavoidable cross-shard operations
   - Consider a federated query layer that can optimize multi-shard queries
   - Use denormalization to avoid common joins
   - Create global indexes for frequently queried data

   Example:

```
def get_user_with_orders(user_id):
    # Locate the shard containing the user
    user_shard = get_shard_for_user(user_id)
    user_data = user_shard.execute("SELECT * FROM users WHERE id = ?",
[user_id])

    # Get orders from the same shard (co-located data)
    if user_data:
        orders = user_shard.execute("SELECT * FROM orders WHERE user_id = ?",
[user_id])
        user_data['orders'] = orders

    return user_data
```

3. **Challenge: Schema Changes and Migrations**

   Applying schema changes across multiple shards is complex and risky.

   Solution:

   - Implement a schema versioning system
   - Use online schema change tools (e.g., gh-ost, pt-online-schema-change)
   - Perform rolling migrations shard by shard

- Implement backward compatibility for schema changes
- Automate the migration process with proper verification steps

Example Migration Process:

```
1. Create a new schema version in the repository
2. Deploy updated application code that can work with both old and new schemas
3. For each shard:
   a. Create new table structure
   b. Incrementally copy and sync data
   c. Verify data integrity
   d. Switch traffic to new table
   e. Remove old table once stable
4. Repeat for each shard sequentially or in batches
```

4. **Challenge: Maintaining Data Consistency**

   Ensuring ACID properties across shards is difficult, especially for transactions that span multiple shards.

   Solution:

   - Use distributed transaction protocols (2PC, saga pattern)
   - Implement compensating transactions for rollbacks
   - Consider relaxing consistency requirements where appropriate
   - Use the Outbox Pattern for eventual consistency
   - Implement distributed locks for critical operations

   Example Saga Implementation:

```java
public class OrderSagaCoordinator {
    public void createOrder(Order order) {
        try {
            // Step 1: Create order in Orders shard
            ordersService.createOrder(order);

            // Step 2: Update inventory in Products shard
            try {
                inventoryService.reserveProducts(order.getItems());
            } catch (Exception e) {
                // Compensating transaction
                ordersService.cancelOrder(order.getId());
                throw e;
            }

            // Step 3: Process payment
            try {
                paymentService.processPayment(order.getPaymentDetails());
            } catch (Exception e) {
                // Compensating transactions
                inventoryService.releaseProductReservation(order.getItems());
                ordersService.cancelOrder(order.getId());
                throw e;
```

```
            }

            // Complete the order
            ordersService.completeOrder(order.getId());

        } catch (Exception e) {
            // Handle failure
            logger.error("Order creation failed", e);
            throw e;
        }
    }
}
```

5. **Challenge: Shard Key Selection**

   Choosing the wrong shard key can lead to significant performance and scaling issues.

   Solution:

   - Carefully analyze query patterns before selecting a shard key
   - Choose high-cardinality, evenly-distributed attributes
   - Avoid keys that cause frequent resharding
   - Consider compound shard keys for complex access patterns
   - Beware of monotonically increasing keys (like timestamps)

   Selection Process:

```
1. Identify the most common queries and their WHERE clauses
2. Analyze data distribution across potential key candidates
3. Evaluate growth patterns and potential hotspots
4. Balance query efficiency with write distribution
5. Test with realistic workloads before committing
```

6. **Challenge: Resharding**

   Adding or removing shards requires redistributing data, which can be complex and risky.

   Solution:

   - Use consistent hashing to minimize data movement during resharding
   - Implement incremental resharding to limit impact
   - Build tooling for automated resharding operations
   - Perform resharding during low-traffic periods
   - Have a rollback strategy ready

   Resharding Process:

```
1. Add new shard instances to the cluster
2. Update hash ring to include new shards
3. Gradually migrate data ranges to new distribution
4. Verify data integrity after each migration step
```

```
5. Update application configuration to use new shard map
6. Monitor and address any issues
```

7. **Challenge: Operational Complexity**

   Sharded databases are more complex to manage, monitor, and troubleshoot.

   Solution:

   - Implement comprehensive monitoring across all shards
   - Create centralized logging and alerting
   - Automate common operations with scripts and tools
   - Develop clear runbooks for operational procedures
   - Use management tools specific to your database technology
   - Consider database-as-a-service options that handle sharding

   Monitoring Setup:

   ```
   1. Per-shard metrics: queries/second, latency, error rates
   2. Cross-shard operation tracking
   3. Shard balance monitoring
   4. Replication lag monitoring
   5. Centralized log aggregation
   6. Automated alerts for anomalies
   ```

8. **Challenge: Global Secondary Indexes**

   Maintaining efficient indexes across sharded data is complicated.

   Solution:

   - Implement local indexes on each shard
   - For global indexes, consider:
     - Separate index shards with their own distribution strategy
     - Index synchronization mechanisms
     - Eventually consistent global indexes
   - Use specialized databases with built-in support for global indexes

   Example Architecture:

   ```
   Data Shards (By user_id):
     Shard 1: users 1-1000
     Shard 2: users a1001-2000

   Global Email Index:
     Index Shard A: email addresses a-m
     Index Shard B: email addresses n-z

   // When querying by email:
   1. Query appropriate index shard
   ```

```
2. Get user_id from index
3. Query appropriate data shard
```

By addressing these challenges methodically, sharding can be implemented successfully, enabling horizontal scaling while minimizing operational risks and application impact.

2.4 Performance Characteristics

Different database types have distinct performance characteristics that make them suitable for specific workloads.

Performance Metrics

1. **Throughput**: Number of operations per second
2. **Latency**: Time to complete an operation
3. **Concurrency**: Ability to handle simultaneous operations
4. **Scalability**: Performance change as load increases
5. **Resource Utilization**: CPU, memory, I/O, and network usage

Relational Database Performance

Strengths:

- Efficient for complex queries and joins
- Good for transactional workloads (OLTP)
- Well-optimized query planners
- Mature tooling for performance tuning

Weaknesses:

- Scaling challenges for very large datasets
- Join performance degrades with scale
- Lock contention under high concurrency
- Schema changes can be resource-intensive

Performance Optimization Techniques:

- Proper indexing strategies
- Query optimization and rewriting
- Partitioning large tables
- Buffer pool tuning
- Connection pooling
- Read-write splitting

Example MySQL Performance Configuration:

```
[mysqld]
# InnoDB Buffer Pool (allocate 70-80% of available memory)
innodb_buffer_pool_size = 16G
innodb_buffer_pool_instances = 8

# Query Cache (disable for high-throughput systems)
```

```
query_cache_type = 0
query_cache_size = 0

# Concurrency Settings
max_connections = 500
innodb_thread_concurrency = 16
thread_cache_size = 32

# InnoDB Log Settings
innodb_log_file_size = 2G
innodb_log_buffer_size = 32M
innodb_flush_log_at_trx_commit = 2

# Table Settings
table_open_cache = 4000
```

NoSQL Database Performance

Document Stores (MongoDB):

- Fast for single-document operations
- Slower for aggregations compared to relational databases
- Excellent for read-heavy workloads
- Good horizontal scaling through sharding

Key-Value Stores (Redis):

- Extremely fast single-key operations
- In-memory performance (sub-millisecond latency)
- Limited query capabilities
- High throughput for simple operations

Column-Family Stores (Cassandra):

- Excellent write throughput
- Fast range scans within a partition
- Consistent performance at scale
- Less efficient for complex queries

Performance Optimization Techniques:

- Data model optimization for access patterns
- Partition key selection for data distribution
- Denormalization for query performance
- Bulk operations for efficiency
- Indexing strategies appropriate to each database

Example Cassandra Performance Configuration:

```
# cassandra.yaml
concurrent_reads: 32
concurrent_writes: 128
```

```
concurrent_compactors: 2
compaction_throughput_mb_per_sec: 64
read_request_timeout_in_ms: 5000
write_request_timeout_in_ms: 2000
memtable_heap_space_in_mb: 2048
memtable_offheap_space_in_mb: 2048
```

In-Memory Database Performance

Characteristics:

- Extremely low latency (microseconds)
- High throughput for read and write operations
- Limited by available memory
- Potential durability tradeoffs

Use Cases:

- Caching layers
- Real-time analytics
- Session stores
- Leaderboards and counters
- Message brokers

Example Redis Benchmark Results:

```
SET: 120,000 requests per second
GET: 140,000 requests per second
INCR: 110,000 requests per second
LPUSH: 100,000 requests per second
LPOP: 105,000 requests per second
```

Performance Comparison Table

Database Type	Read Performance	Write Performance	Query Flexibility	Scaling Characteristic	Memory Utilization
Relational	Good for complex queries	Moderate, affected by indexes	Excellent (SQL)	Vertical, then challenging	High for active dataset
Document	Very good for document retrieval	Good, especially bulk inserts	Good within documents	Horizontal via sharding	Moderate to high
Key-Value	Excellent for simple lookups	Excellent for single records	Limited to key lookups	Linear horizontal scaling	Low to moderate

Database Type	Read Performance	Write Performance	Query Flexibility	Scaling Characteristic	Memory Utilization
Column-Family	Good for column scans	Excellent, optimized for writes	Limited, focused on column families	Nearly linear horizontal	Moderate
Graph	Excellent for connected data	Moderate for relationships	Excellent for traversals	Challenging beyond single instance	High
In-Memory	Extremely fast	Extremely fast	Varies by implementation	Limited by memory	Very high

Interview Questions

Q: How would you optimize the performance of a relational database for a read-heavy application?

A: To optimize the performance of a relational database for a read-heavy application, I would implement a multi-faceted approach:

1. **Implement a Comprehensive Indexing Strategy**:
   - Create indexes on frequently queried columns
   - Consider covering indexes for common queries
   - Use composite indexes for multi-column filtering
   - Monitor index usage and remove unused indexes

```
-- Example: Creating a covering index for a common query
CREATE INDEX idx_products_category_price ON products(category_id, price)
INCLUDE (name, description);

-- Example: Monitoring index usage in PostgreSQL
SELECT relname, idx_scan, idx_tup_read, idx_tup_fetch FROM
pg_stat_user_indexes;
```

2. **Implement Read Replicas**:
   - Set up multiple read replicas to distribute query load
   - Configure an appropriate replication strategy (synchronous vs. asynchronous)
   - Use a connection pool or proxy to route queries

```
Primary DB ──────┐
     │           │
     ▼           ▼
Read Replica 1  Read Replica 2
```

```
# Example: Connection routing logic
def get_db_connection(query_type):
    if query_type == "READ":
        # Round-robin selection from read replicas
        replica_index = next_replica_index()
        return read_replica_pool[replica_index]
    else:
        return primary_connection
```

3. **Implement Caching Layers**:
    - Add Redis/Memcached for caching frequent queries
    - Implement a multi-level caching strategy
    - Set appropriate cache invalidation policies

```
def get_product(product_id):
    # Try cache first
    cache_key = f"product:{product_id}"
    cached = cache.get(cache_key)
    if cached:
        return cached

    # Cache miss - get from database
    product = db.query("SELECT * FROM products WHERE id = %s", [product_id])

    # Store in cache with appropriate TTL
    cache.set(cache_key, product, ttl=3600)  # 1 hour cache
    return product
```

4. **Optimize Query Patterns**:
    - Analyze and rewrite slow queries
    - Use EXPLAIN to understand query execution plans
    - Minimize the use of SELECT *
    - Use appropriate JOINs (prefer inner joins where possible)
    - Implement pagination for large result sets

```
-- Before optimization
SELECT * FROM orders JOIN customers ON orders.customer_id = customers.id
WHERE orders.status = 'processing';

-- After optimization
SELECT o.id, o.order_date, o.total, c.name, c.email
FROM orders o
INNER JOIN customers c ON o.customer_id = c.id
WHERE o.status = 'processing'
LIMIT 100 OFFSET 0;
```

5. **Database Configuration Tuning**:
   - Increase buffer pool/cache size for read operations
   - Optimize read-ahead settings
   - Tune query cache (where appropriate)
   - Configure appropriate isolation level

```
# MySQL example configuration
innodb_buffer_pool_size = 20G         # Larger buffer for caching data
innodb_buffer_pool_instances = 10     # Multiple instances for concurrency
read_rnd_buffer_size = 8M             # Larger read buffer
join_buffer_size = 4M                 # Larger join buffer
innodb_read_io_threads = 12           # More read I/O threads
```

6. **Consider Materialized Views**:
   - Create materialized views for complex, frequent queries
   - Set up a refresh schedule based on data change frequency

```
-- PostgreSQL materialized view example
CREATE MATERIALIZED VIEW product_category_summary AS
SELECT c.name AS category,
       COUNT(p.id) AS product_count,
       AVG(p.price) AS avg_price
FROM products p
JOIN categories c ON p.category_id = c.id
GROUP BY c.name;

-- Refresh schedule
CREATE FUNCTION refresh_mat_views() RETURNS void AS $$
BEGIN
  REFRESH MATERIALIZED VIEW product_category_summary;
END;
$$ LANGUAGE plpgsql;

SELECT cron.schedule('0 */3 * * *', 'SELECT refresh_mat_views()');
```

7. **Data Partitioning/Sharding**:
   - Partition large tables by appropriate columns (date, category, etc.)
   - Consider vertical partitioning for wide tables
   - Implement horizontal sharding for very large datasets

```
-- Partitioning example (PostgreSQL)
CREATE TABLE orders (
    id SERIAL,
    customer_id INT,
    order_date DATE,
    total DECIMAL(10,2)
```

```
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_2023 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');

CREATE TABLE orders_2024 PARTITION OF orders
    FOR VALUES FROM ('2024-01-01') TO ('2025-01-01');
```

8. **Denormalization for Read Performance**:
    - Selectively denormalize data to reduce JOINs
    - Create summary tables for reporting queries
    - Implement data duplication where appropriate

```
-- Example: Denormalized product table with category data
CREATE TABLE denorm_products (
    id INT PRIMARY KEY,
    name VARCHAR(255),
    price DECIMAL(10,2),
    category_id INT,
    category_name VARCHAR(255),  -- Denormalized from categories table
    category_path VARCHAR(255)   -- Denormalized from categories table
);
```

9. **Consider Database Hardware**:
    - Use SSDs for database storage
    - Ensure sufficient RAM for buffer pool
    - Separate data files and transaction logs to different disks
    - Consider RAID configurations optimized for reads

10. **Implement Connection Pooling**:
    - Use connection pooling to reduce connection overhead
    - Configure optimal pool size
    - Consider separate pools for different query types

```
// HikariCP configuration example
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql://hostname:5432/database");
config.setUsername("username");
config.setPassword("password");
config.setMaximumPoolSize(100);
config.setMinimumIdle(20);
config.setConnectionTimeout(30000);

HikariDataSource dataSource = new HikariDataSource(config);
```

11. **Monitoring and Ongoing Optimization**:

- Implement query performance monitoring
- Set up alerts for slow queries
- Regularly review and optimize based on changing patterns
- Consider using tools like PMM (Percona Monitoring and Management)

By implementing this comprehensive strategy, the relational database can be optimized to handle read-heavy workloads efficiently while maintaining data consistency and reliability.

Q: Compare the performance characteristics of Redis, MongoDB, and PostgreSQL for different use cases.

A: Here's a detailed comparison of the performance characteristics of Redis, MongoDB, and PostgreSQL across different use cases:

Redis (In-Memory Key-Value Store)

Performance Profile:

- Extremely low latency (typically < 1ms)
- Very high throughput (100,000+ operations/second on modest hardware)
- In-memory operations with optional persistence
- Single-threaded core architecture (though Redis 6+ offers multithreaded I/O)
- Limited by available memory

Optimal Use Cases:

1. **Caching Layer:**
   - **Performance Metrics:** Sub-millisecond response times, 100K+ reads/sec
   - **Why It Excels:** In-memory storage with O(1) key lookups
   - **Example Scenario:** Caching database queries or API responses

```
# Redis caching implementation
def get_user_profile(user_id):
    cache_key = f"user:{user_id}"
    cached_data = redis.get(cache_key)

    if cached_data:
        return json.loads(cached_data)

    # Cache miss
    user_data = database.query("SELECT * FROM users WHERE id = %s", [user_id])
    redis.setex(cache_key, 3600, json.dumps(user_data))  # 1 hour cache
    return user_data
```

2. **Session Store:**
   - **Performance Metrics:** < 0.5ms response times, high concurrent access
   - **Why It Excels:** Fast operations, built-in TTL support, atomic operations
   - **Example Scenario:** Web application session management

```
SET session:1234 {user_data} EX 3600  # Set with expiration
GET session:1234                      # Retrieve session
```

3. **Real-Time Leaderboards/Counters:**
    - **Performance Metrics:** 80K+ increments/sec, sorted set operations ~50K/sec
    - **Why It Excels:** Specialized data structures (Sorted Sets) with efficient range queries
    - **Example Scenario:** Gaming leaderboards, real-time analytics

```
ZINCRBY leaderboard 10 user:123  # Add 10 points to user 123
ZREVRANGE leaderboard 0 9 WITHSCORES  # Get top 10 users
```

4. **Rate Limiting:**
    - **Performance Metrics:** 100K+ operations/sec, microsecond precision
    - **Why It Excels:** Fast incrementing counters, automatic expiry
    - **Example Scenario:** API rate limiting

```
INCR ratelimit:user:123
EXPIRE ratelimit:user:123 60  # Reset after 60 seconds
```

5. **Pub/Sub Messaging:**
    - **Performance Metrics:** 1M+ messages/sec, low latency (1-2ms)
    - **Why It Excels:** Lightweight implementation, non-persistent messaging
    - **Example Scenario:** Real-time notifications, chat applications

```
SUBSCRIBE channel1
PUBLISH channel1 "Hello World"
```

Performance Limitations:

- Memory capacity constraints
- Limited complex query capabilities
- Single-threaded core (though pipelining helps)
- Eventual consistency in clustered mode
- Limited durability guarantees with high performance settings

MongoDB (Document Database)

Performance Profile:

- Good read/write throughput (10,000+ operations/second)
- Moderate latency (typically 1-10ms)
- Excellent horizontal scalability through sharding

- Rich query language with moderate processing overhead
- Memory-mapped storage engine (WiredTiger)

Optimal Use Cases:

1. **Content Management Systems:**
   - **Performance Metrics:** 5-10K reads/sec, 1-3K writes/sec
   - **Why It Excels:** Flexible schema for varied content types, good indexing
   - **Example Scenario:** Storing blog posts, articles with different structures

```
// Blog post document
{
  _id: ObjectId("..."),
  title: "MongoDB Performance",
  content: "...",
  author: { name: "Jane Smith", bio: "..." },
  tags: ["database", "performance", "nosql"],
  comments: [
    { user: "user123", text: "Great article!", date: ISODate("...") }
  ]
}
```

2. **Product Catalogs:**
   - **Performance Metrics:** 10K+ reads/sec, efficient indexing for filtered queries
   - **Why It Excels:** Supports complex nested documents, good for hierarchical data
   - **Example Scenario:** E-commerce product listings with varying attributes

```
// Query for products with specific attributes
db.products
  .find({
    category: 'electronics',
    price: { $lt: 1000 },
    'specs.ram': '16GB',
  })
  .sort({ price: 1 })
  .limit(20);
```

3. **User Profiles and Preferences:**
   - **Performance Metrics:** 20K+ read operations/sec, 5K+ writes/sec
   - **Why It Excels:** Schema flexibility for evolving user attributes, index support
   - **Example Scenario:** Social media profiles, app configurations

```
// User profile update
db.users.updateOne(
  { _id: ObjectId('...') },
  {
```

```
    $set: { 'preferences.darkMode': true },
    $push: { devices: { type: 'mobile', lastLogin: new Date() } },
  }
);
```

4. **Real-Time Analytics:**
   - **Performance Metrics:** 50K+ inserts/sec with bulk operations, aggregation framework
   - **Why It Excels:** Good at handling high-volume streaming data, flexible schema
   - **Example Scenario:** IoT data collection, event tracking

```
// Aggregation for real-time metrics
db.events.aggregate([
  { $match: { timestamp: { $gte: new Date(Date.now() - 3600000) } } },
  { $group: { _id: '$eventType', count: { $sum: 1 } } },
  { $sort: { count: -1 } },
]);
```

5. **Mobile Applications:**
   - **Performance Metrics:** Efficient document storage, good for occasional sync
   - **Why It Excels:** JSON-native format matches app data structures, flexible schema
   - **Example Scenario:** Mobile app with offline capabilities and sync

```
// Sync only changed documents
db.userContent.find({
  userId: 'user123',
  lastModified: { $gt: lastSyncTimestamp },
});
```

Performance Limitations:

- Higher latency than in-memory databases
- Query performance degrades with complex joins (lookups)
- Aggregation pipeline can be memory-intensive
- Index size can become significant with large collections
- Write lock contention in high-concurrency environments

PostgreSQL (Relational Database)

Performance Profile:

- Moderate to high throughput (5,000+ transactions/second on optimized systems)
- Moderate latency (typically 5-20ms for simple queries)
- Advanced query optimizer with sophisticated execution planning
- Excellent for complex relational data and transactions
- ACID compliance with robust consistency guarantees

Optimal Use Cases:

1. **Transactional Systems (OLTP):**
   - **Performance Metrics:** 1-5K transactions/sec with proper tuning
   - **Why It Excels:** Full ACID compliance, row-level locking, MVCC
   - **Example Scenario:** Banking systems, order processing

```
BEGIN;
-- Update account balances (atomic)
UPDATE accounts SET balance = balance - 100.00 WHERE id = 123;
UPDATE accounts SET balance = balance + 100.00 WHERE id = 456;
COMMIT;
```

2. **Complex Reporting and Analytics:**
   - **Performance Metrics:** Complex JOIN operations 10-100x faster than NoSQL alternatives
   - **Why It Excels:** Sophisticated query planner, efficient JOIN algorithms
   - **Example Scenario:** Business intelligence dashboards

```
SELECT
  c.category_name,
  EXTRACT(MONTH FROM o.order_date) as month,
  SUM(oi.quantity * oi.unit_price) as revenue
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
JOIN categories c ON p.category_id = c.category_id
WHERE o.order_date >= '2023-01-01' AND o.order_date < '2024-01-01'
GROUP BY c.category_name, EXTRACT(MONTH FROM o.order_date)
ORDER BY c.category_name, month;
```

3. **Geographic Information Systems:**
   - **Performance Metrics:** Spatial queries 10-100x faster than non-specialized databases
   - **Why It Excels:** PostGIS extension with R-tree spatial indexing
   - **Example Scenario:** Location-based services, mapping applications

```
-- Find all restaurants within 1km of a point
SELECT name, address
FROM restaurants
WHERE ST_DWithin(
  location::geography,
  ST_MakePoint(-73.9857, 40.7484)::geography,
  1000  -- 1km in meters
);
```

4. **Full-Text Search Systems:**

- **Performance Metrics:** Comparable to specialized search engines for moderate workloads
- **Why It Excels:** Advanced text search capabilities (tsvector, GIN indexes)
- **Example Scenario:** Document search, content discovery

```
-- Create a text search index
CREATE INDEX idx_articles_fts ON articles
USING gin(to_tsvector('english', title || ' ' || content));

-- Search query
SELECT id, title
FROM articles
WHERE to_tsvector('english', title || ' ' || content) @@
      to_tsquery('english', 'database & performance');
```

5. **Multi-tenant Applications:**
   - **Performance Metrics:** Efficient row-level security, schema isolation
   - **Why It Excels:** Schema management, row-level security, table partitioning
   - **Example Scenario:** SaaS applications with data isolation requirements

```
-- Row-level security for multi-tenancy
CREATE TABLE customer_data (
  id SERIAL PRIMARY KEY,
  tenant_id INTEGER NOT NULL,
  customer_name TEXT,
  data JSONB
);

CREATE POLICY tenant_isolation ON customer_data
  USING (tenant_id = current_setting('app.tenant_id')::INTEGER);

ALTER TABLE customer_data ENABLE ROW LEVEL SECURITY;
```

Performance Limitations:

- Vertical scaling has physical limits
- Complex query planning can add overhead
- Vacuum process can impact performance
- Write amplification with heavy indexing
- Connection management overhead

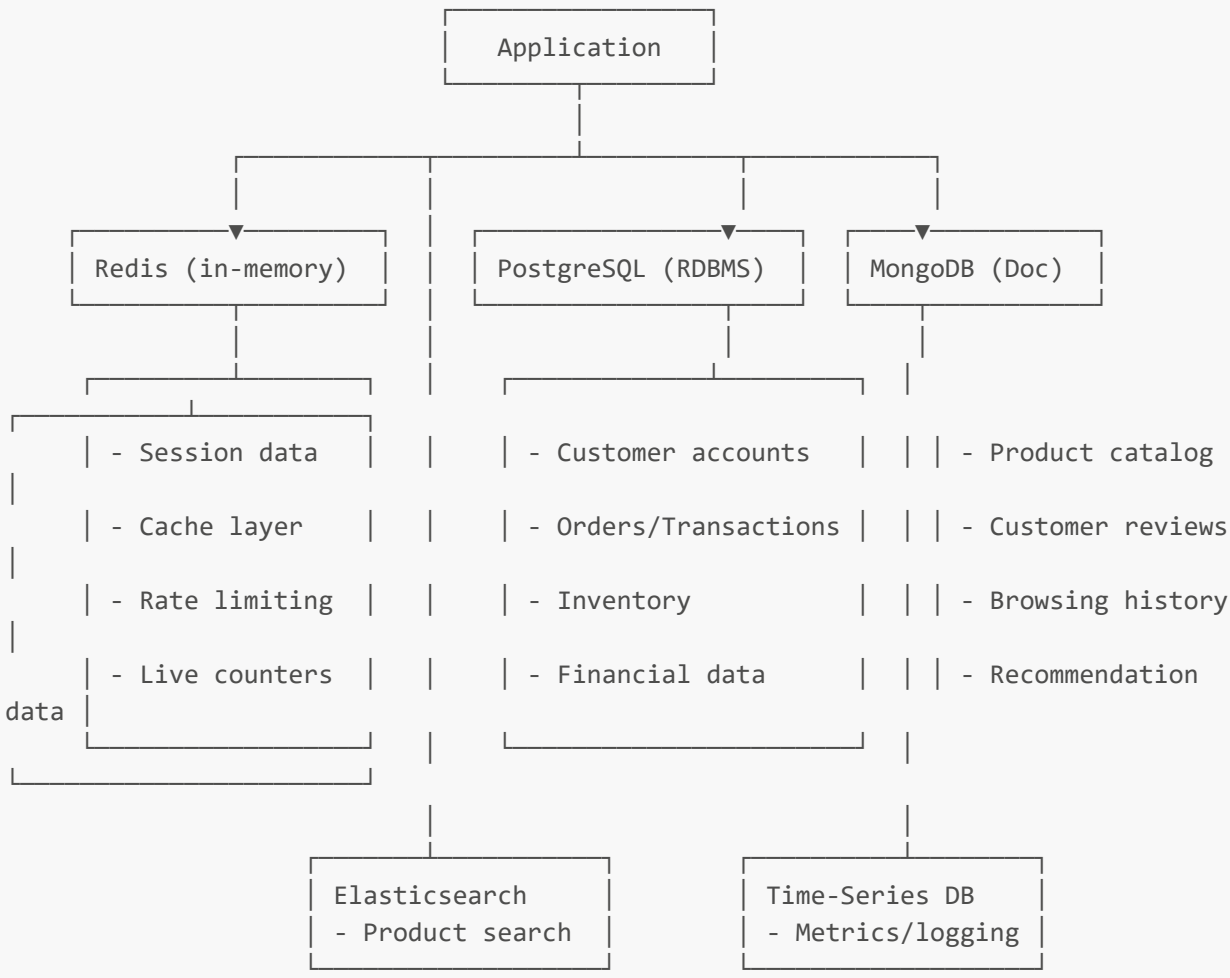
Performance Comparison: Redis vs. MongoDB vs. PostgreSQL

Scenario	Redis	MongoDB	PostgreSQL
Simple Key-Value Lookups	~200K ops/sec	~50K ops/sec	~10K ops/sec

Scenario	Redis	MongoDB	PostgreSQL
Complex Queries	Limited capability	~5K ops/sec	~2K ops/sec with better consistency
Write Throughput	~100K ops/sec	~10K ops/sec	~5K ops/sec with ACID
Data Size Handling	Limited by memory	Terabytes with sharding	Terabytes with proper config
Join Operations	N/A	Limited ($lookup)	Excellent performance
Memory Efficiency	High (specialized structures)	Moderate	Moderate to high with proper tuning
Transaction Support	Limited (Redis 7.0+)	Limited (multi-document)	Full ACID compliance

Real-World Hybrid Architecture Example

For an e-commerce application with high traffic:



Implementation Notes:

- Redis handles high-velocity data and caching

- PostgreSQL manages transactional data requiring ACID properties
- MongoDB stores flexible, document-oriented product information
- Each database is optimized for specific access patterns

Interview Questions

Q: For a social media application experiencing performance issues with feed generation, how would you decide between Redis, MongoDB, and PostgreSQL for storing and retrieving user feeds?

A: When addressing performance issues with social media feed generation, I would evaluate Redis, MongoDB, and PostgreSQL based on the specific feed requirements:

Analysis of Requirements for Social Media Feeds:

1. **Read-to-Write Ratio**: Extremely read-heavy (users view feeds much more than they post)
2. **Access Pattern**: Time-ordered content with pagination
3. **Data Relationship Complexity**: Posts may include user data, engagement metrics, media
4. **Consistency Requirements**: Eventual consistency acceptable for feeds
5. **Query Patterns**: Filtering by user connections, trending content, personalization
6. **Scale Considerations**: Must handle millions of users and posts

Redis Approach:

```
Pros:
- Extremely fast retrieval (sub-millisecond)
- Built-in sorted sets perfect for time-ordered data
- Atomic operations for engagement counters
- Excellent for caching hot/trending content

Cons:
- Memory constraints limit historical feed depth
- Limited querying capabilities for complex filtering
- Requires application logic for joins/relationships

Implementation Strategy:
```

```
# Store timeline as a sorted set with timestamp as score
def add_post_to_feeds(post_id, user_id, timestamp):
    # Get user's followers
    follower_ids = get_followers(user_id)

    pipeline = redis.pipeline()
    # Add to each follower's feed
    for follower_id in follower_ids:
        feed_key = f"feed:{follower_id}"
        pipeline.zadd(feed_key, {post_id: timestamp})
        # Trim feed to last 1000 posts to manage memory
        pipeline.zremrangebyrank(feed_key, 0, -1001)
    pipeline.execute()

# Retrieve user feed with pagination
```

```
def get_user_feed(user_id, page=0, size=20):
    start = page * size
    end = start + size - 1

    # Get post IDs from sorted set (newest first)
    post_ids = redis.zrevrange(f"feed:{user_id}", start, end)

    # Fetch actual post content (could come from another database)
    posts = []
    for post_id in post_ids:
        post_data = get_post_data(post_id)  # This might fetch from MongoDB
        posts.append(post_data)

    return posts
```

MongoDB Approach:

```
Pros:
- Flexible schema for evolving post types
- Good read performance with proper indexing
- Built-in aggregation pipeline for feed generation
- Can store complete post content

Cons:
- Not as fast as Redis for simple time-based ordering
- Requires careful index design for performance
- Higher latency than in-memory solutions

Implementation Strategy:
```

```
// Posts collection structure
{
  _id: ObjectId("..."),
  user_id: ObjectId("..."),
  content: "...",
  media_urls: ["url1", "url2"],
  created_at: ISODate("..."),
  engagement: {
    likes: 42,
    comments: 7,
    shares: 3
  },
  // Denormalized user data for feed display
  user_data: {
    username: "user123",
    profile_pic: "url"
  }
}

// Feed generation query
db.posts.aggregate([
```

```
  // Match posts from users this user follows
  { $match: {
    user_id: { $in: followedUserIds },
    created_at: { $gt: new Date(Date.now() - 7*24*60*60*1000) } // Last 7 days
  }},
  // Sort by creation time, newest first
  { $sort: { created_at: -1 } },
  // Limit to page size
  { $limit: 20 },
  // Optional: Lookup comments
  { $lookup: {
    from: "comments",
    localField: "_id",
    foreignField: "post_id",
    as: "recent_comments"
  }},
  // Limit comments to most recent
  { $project: {
    // Keep all post fields
    "recent_comments": { $slice: ["$recent_comments", 3] }
  }}
])

// Critical indexes
db.posts.createIndex({ user_id: 1, created_at: -1 })
db.posts.createIndex({ created_at: -1 }) // For trending
```

PostgreSQL Approach:

```
Pros:
- Excellent for complex joining of related data
- Strong consistency guarantees
- Advanced filtering capabilities
- Built-in full-text search for content

Cons:
- Higher latency than NoSQL options
- Complex queries can be costly at scale
- More overhead for schema changes

Implementation Strategy:
```

```
-- Schema design
CREATE TABLE users (
  user_id SERIAL PRIMARY KEY,
  username VARCHAR(50) UNIQUE,
  profile_pic_url VARCHAR(255)
);

CREATE TABLE posts (
  post_id SERIAL PRIMARY KEY,
```

```
  user_id INTEGER REFERENCES users(user_id),
  content TEXT,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  -- Other post attributes
);

CREATE TABLE follows (
  follower_id INTEGER REFERENCES users(user_id),
  followed_id INTEGER REFERENCES users(user_id),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  PRIMARY KEY (follower_id, followed_id)
);

CREATE TABLE engagements (
  post_id INTEGER REFERENCES posts(post_id),
  engagement_type VARCHAR(20), -- 'like', 'comment', 'share'
  user_id INTEGER REFERENCES users(user_id),
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
  content TEXT, -- For comments
  PRIMARY KEY (post_id, user_id, engagement_type)
);

-- Feed query
SELECT
  p.post_id,
  p.content,
  p.created_at,
  u.username,
  u.profile_pic_url,
  (SELECT COUNT(*) FROM engagements WHERE post_id = p.post_id AND engagement_type =
'like') AS like_count,
  (SELECT COUNT(*) FROM engagements WHERE post_id = p.post_id AND engagement_type =
'comment') AS comment_count
FROM posts p
JOIN users u ON p.user_id = u.user_id
JOIN follows f ON p.user_id = f.followed_id
WHERE f.follower_id = 123 -- User viewing the feed
ORDER BY p.created_at DESC
LIMIT 20 OFFSET 0;

-- Critical indexes
CREATE INDEX idx_posts_user_created ON posts(user_id, created_at DESC);
CREATE INDEX idx_follows_follower ON follows(follower_id);
CREATE INDEX idx_engagements_post_type ON engagements(post_id, engagement_type);
```

Hybrid Solution:

For optimal performance, I would implement a hybrid approach:

1. **Redis for Feed Caching and Delivery**:
    - Store pre-computed feed entries in Redis sorted sets
    - Maintain engagement counters in Redis for real-time updates
    - Cache most recent/active feeds for fast retrieval

2. **MongoDB for Content Storage**:
    - Store the actual post content and metadata
    - Use for historical feed retrieval beyond cache
    - Leverage for complex aggregations and analytics
3. **PostgreSQL for Relationship Management**:
    - Maintain social graph (follows, blocks, etc.)
    - Handle transactional aspects (user accounts, settings)
    - Enable complex filter rules and privacy controls

Implementation flow:

```
1. User creates post → Store in MongoDB → Fan out post IDs to Redis feeds
2. User views feed → Fetch IDs from Redis → Get content from MongoDB
3. For cold/historical feeds → Generate from MongoDB directly
4. Social graph changes → Update in PostgreSQL → Trigger feed rebuilds
```

Recommendation Based on Performance Requirements:

For a social media application with performance issues in feed generation, I would recommend:

1. **Primary Solution: Redis + MongoDB**
    - Redis handles the feed composition and delivery (solving immediate performance issues)
    - MongoDB stores the actual content with flexibility for different post types
    - This approach provides sub-millisecond feed loading times while maintaining flexibility
2. **Scaling Considerations:**
    - Implement Redis Cluster for larger user bases
    - Shard MongoDB by user_id for horizontal scaling
    - Use read replicas for both systems
3. **Optimization Techniques:**
    - Lazy loading of content not visible in the initial viewport
    - Incremental feed updates rather than full regeneration
    - Time-based partitioning for historical content

This hybrid approach balances the performance benefits of Redis with the flexibility of MongoDB, addressing the immediate performance issues while allowing for future growth and feature development.

Q: When would it make sense to migrate from PostgreSQL to a NoSQL solution like MongoDB or Redis, and what would be your migration strategy?

A: Deciding to migrate from PostgreSQL to a NoSQL solution like MongoDB or Redis requires careful evaluation of specific triggers, use cases, and a well-planned migration strategy.

When Migration Makes Sense

Technical Triggers:

1. **Schema Flexibility Requirements**:
    - Your application requires frequent schema changes
    - You're storing diverse data types that don't fit a rigid table structure
    - Example: A content management system where different content types have widely varying attributes
2. **Scaling Challenges**:
    - You're approaching vertical scaling limits with PostgreSQL
    - Read/write throughput requirements exceed what's reasonable for PostgreSQL
    - Cross-shard joins in a sharded PostgreSQL setup are causing performance issues
    - Example: A social media application where user-generated content is growing exponentially
3. **Performance Bottlenecks**:
    - High-volume, simple read/write operations are dominating your workload
    - Specific query patterns are consistently slow despite optimization
    - Example: Session management with millions of concurrent users
4. **Specialized Data Access Patterns**:
    - Time-series data with high write throughput
    - Geographic or graph data requiring specialized operations
    - Example: IoT applications generating massive sensor readings

Business Triggers:

1. **Development Velocity**:
    - Schema changes are slowing down feature development
    - Teams need greater agility for rapid prototyping
    - Example: Startup pivoting its product and needing to iterate quickly
2. **Cost Considerations**:
    - Operational costs for scaling PostgreSQL become prohibitive
    - Lower consistency requirements could allow more cost-effective solutions
    - Example: Analytics system where eventual consistency is acceptable
3. **New Use Cases**:
    - New business requirements better align with NoSQL models
    - Example: Expanding into recommendation systems requiring document storage

Migration Strategy

1. Assessment Phase:

```
a. Workload Analysis:
   - Query patterns (read/write ratio, query complexity)
```

```
   - Transaction requirements
   - Data volume and growth projections
   - Performance metrics and bottlenecks

b. Data Modeling:
   - Map relational schema to NoSQL models
   - Identify denormalization opportunities
   - Address relationship handling

c. Consistency Requirements:
   - Identify operations requiring strong consistency
   - Areas where eventual consistency is acceptable
```

Example Workload Analysis:

```
-- PostgreSQL query to analyze query patterns
SELECT
  queryid,
  calls,
  total_time,
  mean_time,
  rows,
  query
FROM pg_stat_statements
ORDER BY total_time DESC
LIMIT 20;
```

2. Proof of Concept:

```
a. Select Critical Subset:
   - Identify bounded context for initial migration
   - Choose representative but non-critical workload

b. Dual-Write Implementation:
   - Write to both PostgreSQL and NoSQL during testing
   - Compare performance and functionality

c. Validation:
   - Ensure data consistency between systems
   - Benchmark performance improvements
   - Validate application behavior
```

Example Dual-Write Pattern:

```
public void saveCustomer(Customer customer) {
    // Write to PostgreSQL
    jdbcTemplate.update(
        "INSERT INTO customers (id, name, email) VALUES (?, ?, ?)",
        customer.getId(), customer.getName(), customer.getEmail()
```

```
        );

        // Write to MongoDB
        Document customerDoc = new Document()
            .append("_id", customer.getId())
            .append("name", customer.getName())
            .append("email", customer.getEmail())
            .append("preferences", customer.getPreferencesAsMap());

        mongoCollection.insertOne(customerDoc);
    }
```

3. Data Migration Strategy:

```
a. For MongoDB Migration:
   - Use change data capture (CDC) tools like Debezium
   - Set up replication from PostgreSQL to MongoDB
   - Validate data integrity after initial load
```

Example using Debezium:

```
{
  "name": "postgresql-connector",
  "config": {
    "connector.class": "io.debezium.connector.postgresql.PostgresConnector",
    "database.hostname": "postgres",
    "database.port": "5432",
    "database.user": "debezium",
    "database.password": "dbz",
    "database.dbname": "inventory",
    "database.server.name": "dbserver1",
    "table.include.list": "public.customers",
    "transforms": "unwrap",
    "transforms.unwrap.type": "io.debezium.transforms.ExtractNewRecordState",
    "transforms.unwrap.drop.tombstones": "false",
    "transforms.unwrap.delete.handling.mode": "rewrite"
  }
}
```

```
b. For Redis Migration:
   - Identify cacheable data
   - Pre-populate Redis from PostgreSQL
   - Implement cache warming strategies
```

Example Redis pre-population:

```
def populate_redis_cache():
    # Connect to PostgreSQL
    pg_conn = psycopg2.connect("dbname=myapp user=user password=pass")
    cursor = pg_conn.cursor()

    # Get frequently accessed products
    cursor.execute("""
        SELECT product_id, name, price, inventory
        FROM products
        WHERE last_accessed > NOW() - INTERVAL '7 days'
    """)

    # Populate Redis
    for product in cursor.fetchall():
        product_key = f"product:{product[0]}"
        redis_client.hmset(product_key, {
            "name": product[1],
            "price": product[2],
            "inventory": product[3]
        })
        # Set expiration
        redis_client.expire(product_key, 3600)  # 1 hour cache
```

4. Application Adaptation:

```
a. Data Access Layer Refactoring:
   - Abstract database access behind interfaces
   - Implement adapters for PostgreSQL and NoSQL
   - Feature flags for gradual cutover
```

Example Adapter Pattern:

```
// Interface
public interface CustomerRepository {
    Customer findById(String id);
    void save(Customer customer);
    List<Customer> findByRegion(String region);
}

// PostgreSQL Implementation
public class PostgresCustomerRepository implements CustomerRepository {
    private JdbcTemplate jdbcTemplate;

    @Override
    public Customer findById(String id) {
        return jdbcTemplate.queryForObject(
            "SELECT * FROM customers WHERE id = ?",
            new CustomerRowMapper(), id
        );
    }
```

```
    // Other methods...
}

// MongoDB Implementation
public class MongoCustomerRepository implements CustomerRepository {
    private MongoCollection<Document> collection;

    @Override
    public Customer findById(String id) {
        Document doc = collection.find(eq("_id", id)).first();
        return mapToCustomer(doc);
    }

    // Other methods...
}

// Factory with feature flag
public class CustomerRepositoryFactory {
    public CustomerRepository getRepository() {
        if (featureFlagService.isEnabled("use-mongodb")) {
            return new MongoCustomerRepository();
        } else {
            return new PostgresCustomerRepository();
        }
    }
}
```

5. Rollout Strategy:

```
a. Phased Approach:
   - Start with read-only workloads
   - Gradually move write operations
   - Monitor and validate at each step

b. Traffic Shifting:
   - Canary testing with subset of users
   - A/B testing different database technologies
   - Gradual traffic increase to NoSQL
```

Example Canary Deployment:

```
public class DatabaseRouter {
    private Random random = new Random();
    private double noSqlPercentage = 0.05; // Start with 5% of traffic

    public CustomerRepository getRepository(String userId) {
        // Deterministic routing based on user ID for consistent experience
        int userHash = userId.hashCode();
        boolean useNoSql = (Math.abs(userHash % 100) < noSqlPercentage * 100);
```

```
        if (useNoSql) {
            return new MongoCustomerRepository();
        } else {
            return new PostgresCustomerRepository();
        }
    }

    // Method to increase NoSQL percentage gradually
    public void increaseNoSqlPercentage(double increment) {
        this.noSqlPercentage = Math.min(1.0, this.noSqlPercentage + increment);
    }
}
```

6. Monitoring and Optimization:

```
a. Key Metrics:
   - Response times for critical operations
   - Error rates and consistency issues
   - Resource utilization for both systems

b. Performance Tuning:
   - Index optimization for NoSQL
   - Query pattern adjustments
   - Connection pooling configuration
```

Example Monitoring Setup (Prometheus metrics):

```
@Component
public class DatabaseMetrics {
    private final MeterRegistry registry;

    public DatabaseMetrics(MeterRegistry registry) {
        this.registry = registry;
    }

    public Timer.Sample startQueryTimer() {
        return Timer.start(registry);
    }

    public void recordQueryTime(Timer.Sample sample, String database, String
operation) {
        sample.stop(registry.timer("database.query.time",
            "database", database,
            "operation", operation));
    }

    public void incrementErrorCount(String database, String operation, String
errorType) {
        registry.counter("database.error.count",
            "database", database,
            "operation", operation,
```

```
                "error", errorType).increment();
        }
    }
```

Migration Examples for Specific Use Cases

1. Session Management Migration to Redis:

```
Before (PostgreSQL):
```

```
CREATE TABLE sessions (
    session_id VARCHAR(128) PRIMARY KEY,
    user_id INTEGER REFERENCES users(id),
    data JSONB,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    expires_at TIMESTAMP NOT NULL
);

CREATE INDEX idx_sessions_expires ON sessions(expires_at);
```

```
After (Redis):
```

```
def store_session(session_id, user_id, data, ttl_seconds=3600):
    session_key = f"session:{session_id}"
    session_data = {
        "user_id": user_id,
        "data": json.dumps(data),
        "created_at": time.time()
    }

    # Store with expiration
    redis_client.hmset(session_key, session_data)
    redis_client.expire(session_key, ttl_seconds)

def get_session(session_id):
    session_key = f"session:{session_id}"
    session_data = redis_client.hgetall(session_key)

    if not session_data:
        return None

    # Parse JSON data
    if "data" in session_data:
        session_data["data"] = json.loads(session_data["data"])

    return session_data
```

2. Product Catalog Migration to MongoDB:

```
Before (PostgreSQL):
```

```
CREATE TABLE products (
    product_id SERIAL PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    category_id INTEGER REFERENCES categories(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE product_attributes (
    product_id INTEGER REFERENCES products(product_id),
    attribute_name VARCHAR(100) NOT NULL,
    attribute_value TEXT NOT NULL,
    PRIMARY KEY (product_id, attribute_name)
);

-- Query for products with specific attributes
SELECT p.*, array_agg(pa.attribute_name || ':' || pa.attribute_value) as attributes
FROM products p
LEFT JOIN product_attributes pa ON p.product_id = pa.product_id
WHERE p.category_id = 5
GROUP BY p.product_id;
```

```
After (MongoDB):
```

```
// Document structure
{
  "_id": ObjectId("..."),
  "name": "Smartphone XYZ",
  "price": 799.99,
  "category": {
    "id": 5,
    "name": "Electronics"
  },
  "attributes": {
    "color": "black",
    "memory": "128GB",
    "processor": "Snapdragon 8 Gen 1"
  },
  "created_at": ISODate("..."),
  "updated_at": ISODate("...")
}
```

```
// Query for products with specific attributes
db.products.find({
  "category.id": 5,
  "attributes.memory": "128GB"
})
```

Final Recommendations

When considering migration from PostgreSQL to NoSQL:

1. **Be selective**: Migrate specific workloads that benefit most from NoSQL characteristics
2. **Consider a polyglot persistence approach**:
    - Keep transactional data in PostgreSQL
    - Move suitable workloads to appropriate NoSQL databases
    - Use Redis for caching and real-time features
    - Use MongoDB for flexible schema requirements
3. **Evaluate the trade-offs**:
    - Consistency vs. performance
    - Developer productivity vs. operational complexity
    - Query flexibility vs. specialized capabilities
4. **Plan for potential challenges**:
    - Loss of transactions spanning multiple operations
    - Increased complexity in data consistency
    - New operational skills and monitoring requirements

By taking a thoughtful, phased approach to migration, you can leverage the strengths of NoSQL databases while minimizing disruption and risk.

2.5 Use Case Scenarios

Different database types excel in different scenarios. Here's a breakdown of ideal use cases for each type of database.

Relational Databases (RDBMS)

1. Financial Systems and Banking

- **Why suitable**: ACID transactions, data integrity, complex reporting
- **Example implementation**: Core banking system with account management, transfers, and ledger
- **Technical considerations**: Transaction isolation levels, stored procedures for critical operations

```
-- Bank transfer with transaction
BEGIN;
-- Check sufficient funds
SELECT balance FROM accounts WHERE account_id = 123 FOR UPDATE;
```

```
-- If sufficient, update both accounts
UPDATE accounts SET balance = balance - 1000 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 1000 WHERE account_id = 456;
-- Record the transaction
INSERT INTO transactions (from_account, to_account, amount, type)
VALUES (123, 456, 1000, 'TRANSFER');
COMMIT;
```

2. Enterprise Resource Planning (ERP)

- **Why suitable**: Complex relationships, reporting, consistency requirements
- **Example implementation**: Inventory management tied to orders, invoicing, and purchasing
- **Technical considerations**: Stored procedures, triggers for data integrity

3. Customer Relationship Management (CRM)

- **Why suitable**: Structured customer data, transaction history, reporting
- **Example implementation**: Customer database with interactions, purchases, and support tickets
- **Technical considerations**: Indexing for search performance, partitioning for large datasets

4. E-commerce Order Processing

- **Why suitable**: Order integrity, inventory management, financial transactions
- **Example implementation**: Order management system with inventory, payments, and shipping
- **Technical considerations**: Locking strategies, transaction management

Document Databases

1. Content Management Systems

- **Why suitable**: Variable content structure, flexible attributes, good query performance
- **Example implementation**: Blog platform with different content types and attributes
- **Technical considerations**: Indexes for text search, schema validation for consistency

```
// MongoDB content structure
{
  "_id": ObjectId("..."),
  "type": "article",
  "title": "MongoDB Use Cases",
  "content": "...",
  "author": {
    "id": 123,
    "name": "Jane Smith",
    "email": "jane@example.com"
  },
  "tags": ["database", "nosql", "mongodb"],
  "comments": [
    { "user": "user123", "text": "Great article!", "date": ISODate("...") }
  ],
  "metadata": {
    "published_date": ISODate("..."),
    "reading_time": 5,
```

```
    "featured": true
  }
}

// Query for featured articles with specific tag
db.content.find({
  "type": "article",
  "tags": "mongodb",
  "metadata.featured": true
}).sort({"metadata.published_date": -1})
```

2. Product Catalogs

- **Why suitable**: Varying product attributes, schema flexibility, query performance
- **Example implementation**: E-commerce catalog with diverse product categories
- **Technical considerations**: Indexing for search, denormalization for performance

3. User Profiles and Preferences

- **Why suitable**: Evolving user attributes, schema flexibility
- **Example implementation**: Social media profiles with customizable fields
- **Technical considerations**: Indexing for lookup performance, field-level updates

4. Event Logging and Monitoring

- **Why suitable**: Schemaless events, high write throughput
- **Example implementation**: Application monitoring storing diverse event types
- **Technical considerations**: Time-based partitioning, TTL indexes for expiration

Key-Value Stores

1. Caching Layer

- **Why suitable**: Extremely fast access, simple data model, expiration capabilities
- **Example implementation**: Database query cache, API response cache
- **Technical considerations**: Eviction policies, memory management

```
# Redis caching example
def get_user_profile(user_id):
    cache_key = f"user:{user_id}"

    # Try to get from cache
    cached_data = redis_client.get(cache_key)
    if cached_data:
        return json.loads(cached_data)

    # Cache miss - get from database
    user_data = database.execute_query(
        "SELECT * FROM users WHERE id = %s", (user_id,)
    )

    # Store in cache for future requests (with 30-minute TTL)
```

```
    redis_client.setex(cache_key, 1800, json.dumps(user_data))

    return user_data
```

2. Session Storage

- **Why suitable**: Fast access, expiration support, simple structure
- **Example implementation**: Web application session management
- **Technical considerations**: TTL for session expiration, data serialization

3. Distributed Locks and Semaphores

- **Why suitable**: Atomic operations, high availability, TTL support
- **Example implementation**: Distributed lock manager for microservices
- **Technical considerations**: Lock expiration, retry mechanisms

```
# Redis distributed lock implementation
def acquire_lock(lock_name, timeout=10):
    lock_key = f"lock:{lock_name}"
    # Set NX means "only set if not exists"
    return redis_client.set(lock_key, "locked", nx=True, ex=timeout)

def release_lock(lock_name):
    lock_key = f"lock:{lock_name}"
    redis_client.delete(lock_key)

# Usage
if acquire_lock("critical_section"):
    try:
        # Perform critical operation
        process_data()
    finally:
        release_lock("critical_section")
else:
    # Could not acquire lock
    print("Operation in progress by another process")
```

4. Real-time Leaderboards and Counters

- **Why suitable**: Atomic operations, sorted sets, high performance
- **Example implementation**: Gaming leaderboards, view counters
- **Technical considerations**: Sorted set operations, expiry for time-windowed data

Column-Family Stores

1. Time-Series Data Storage

- **Why suitable**: Optimized for time-based data, high write throughput
- **Example implementation**: IoT device readings, log data
- **Technical considerations**: Time-based partitioning, TTL for data expiration

```
-- Cassandra time-series model
CREATE TABLE sensor_data (
  sensor_id uuid,
  date text,
  timestamp timestamp,
  temperature float,
  humidity float,
  pressure float,
  PRIMARY KEY ((sensor_id, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);

-- Query for sensor data in a time range
SELECT * FROM sensor_data
WHERE sensor_id = 123e4567-e89b-12d3-a456-426614174000
AND date = '2023-11-01'
AND timestamp >= '2023-11-01 00:00:00'
AND timestamp < '2023-11-01 01:00:00';
```

2. Log Data Management

- **Why suitable**: High write throughput, time-based querying, compression
- **Example implementation**: Application logging infrastructure
- **Technical considerations**: Partitioning strategy, compaction policies

3. High-Volume Counters and Metrics

- **Why suitable**: High write throughput, time-based aggregation
- **Example implementation**: Real-time analytics system
- **Technical considerations**: Counter columns, consistency levels

4. Large-Scale Content and Asset Storage

- **Why suitable**: Distributed architecture, scalability for large objects
- **Example implementation**: Content delivery backend
- **Technical considerations**: Partition key design, replication factors

Graph Databases

1. Social Networks

- **Why suitable**: Optimized for relationship traversal, complex connection queries
- **Example implementation**: Friend/follower relationships, content sharing
- **Technical considerations**: Graph modeling, index-free adjacency

```
// Neo4j social graph queries
// Find friends of friends who like a specific genre
MATCH (user:User {name: 'Alice'})-[:FRIEND]->(friend)-[:FRIEND]->(fof)
WHERE fof <> user AND NOT (user)-[:FRIEND]->(fof)
AND (fof)-[:LIKES]->(:Genre {name: 'Jazz'})
RETURN fof.name, COUNT(friend) AS common_friends
ORDER BY common_friends DESC;
```

```
// Find the shortest path between two users
MATCH path = shortestPath((user1:User {name: 'Alice'})-[:FRIEND*]-(user2:User {name:
'Bob'}))
RETURN path;
```

2. Fraud Detection

- **Why suitable**: Pattern detection in connected data, path analysis
- **Example implementation**: Financial transaction analysis
- **Technical considerations**: Temporal aspects, path algorithms

3. Recommendation Engines

- **Why suitable**: Relationship-based recommendations, path analysis
- **Example implementation**: Product recommendations based on user similarity
- **Technical considerations**: Similarity algorithms, weighting strategies

4. Knowledge Graphs

- **Why suitable**: Semantic relationships, inferencing capabilities
- **Example implementation**: Enterprise knowledge base
- **Technical considerations**: Ontology design, inference rules

Time-Series Databases

1. IoT Sensor Data

- **Why suitable**: Optimized for time-based data, high ingest rate
- **Example implementation**: Manufacturing equipment monitoring
- **Technical considerations**: Downsampling, retention policies

```
-- TimescaleDB example
CREATE TABLE sensor_data (
  time TIMESTAMPTZ NOT NULL,
  sensor_id INTEGER,
  temperature FLOAT,
  humidity FLOAT,
  battery FLOAT
);

-- Convert to hypertable (partitioning by time)
SELECT create_hypertable('sensor_data', 'time');

-- Query for average temperature by hour
SELECT
  time_bucket('1 hour', time) AS hour,
  sensor_id,
  AVG(temperature) AS avg_temp
FROM sensor_data
WHERE time > NOW() - INTERVAL '1 day'
```

```
GROUP BY hour, sensor_id
ORDER BY hour DESC, sensor_id;
```

2. Application Monitoring and APM

- **Why suitable**: High write throughput, time-based aggregation, retention policies
- **Example implementation**: Microservices monitoring system
- **Technical considerations**: Pre-aggregation, continuous queries

3. Financial Market Data

- **Why suitable**: Time precision, custom aggregations, performance
- **Example implementation**: Stock market data analysis
- **Technical considerations**: High availability, query optimization

4. Infrastructure Monitoring

- **Why suitable**: Metrics collection, alerting, visualization integration
- **Example implementation**: Server and network monitoring
- **Technical considerations**: Downsampling for historical data, retention policies

Search Engines (Search-Oriented Databases)

1. Full-Text Search Applications

- **Why suitable**: Inverted indexes, relevance scoring, faceted search
- **Example implementation**: Site search, document search
- **Technical considerations**: Analyzer configuration, stopwords, synonyms

```
// Elasticsearch mapping for articles
{
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "english",
        "fields": {
          "keyword": { "type": "keyword" }
        }
      },
      "content": { "type": "text", "analyzer": "english" },
      "tags": { "type": "keyword" },
      "author": { "type": "keyword" },
      "published_date": { "type": "date" }
    }
  }
}

// Complex search query with filters and highlighting
{
  "query": {
    "bool": {
```

```
        "must": [
          { "match": { "content": "database performance optimization" } }
        ],
        "filter": [
          { "term": { "tags": "nosql" } },
          { "range": { "published_date": { "gte": "2023-01-01" } } }
        ]
      }
    },
    "highlight": {
      "fields": {
        "content": { "number_of_fragments": 3, "fragment_size": 150 }
      }
    },
    "aggs": {
      "popular_tags": {
        "terms": { "field": "tags", "size": 10 }
      }
    }
  }
```

2. Log Analysis and Monitoring

- **Why suitable**: Text analysis, pattern matching, aggregations
- **Example implementation**: Log aggregation and analysis platform
- **Technical considerations**: Index lifecycle management, shard allocation

3. E-commerce Search

- **Why suitable**: Faceted navigation, filtering, boosting/relevance
- **Example implementation**: Product search with filters and categories
- **Technical considerations**: Custom scoring, synonyms, search-as-you-type

4. Content Discovery Platforms

- **Why suitable**: Relevance algorithms, content analysis
- **Example implementation**: News aggregator, article recommendation
- **Technical considerations**: Natural language processing, text classification

In-Memory Databases

1. Real-Time Bidding Systems

- **Why suitable**: Microsecond latency, high throughput
- **Example implementation**: Ad tech bidding platform
- **Technical considerations**: Memory optimization, data persistence strategy

```
// Redis-based bidding system example
public class BiddingSystem {
    private Jedis redis;

    public BiddingSystem() {
```

```
        this.redis = new Jedis("localhost");
    }

    public void recordBid(String auctionId, String bidderId, double amount) {
        // Store bid in sorted set, sorted by amount
        redis.zadd("auction:" + auctionId, amount, bidderId);

        // Add auction to active auctions with TTL
        redis.sadd("active_auctions", auctionId);
        redis.expire("auction:" + auctionId, 30); // 30-second auction
    }

    public String getHighestBidder(String auctionId) {
        // Get highest bidder (last element in sorted set)
        Set<String> bidders = redis.zrevrange("auction:" + auctionId, 0, 0);
        return bidders.isEmpty() ? null : bidders.iterator().next();
    }

    public Double getHighestBid(String auctionId) {
        // Get highest bid amount
        String highestBidder = getHighestBidder(auctionId);
        if (highestBidder == null) return 0.0;

        return redis.zscore("auction:" + auctionId, highestBidder);
    }
}
```

2. Gaming Leaderboards and State

- **Why suitable**: Fast updates and reads, sorted sets
- **Example implementation**: Multiplayer game leaderboards
- **Technical considerations**: Memory usage optimization, expiry policies

3. Real-Time Analytics Dashboards

- **Why suitable**: Fast aggregations, pub/sub capabilities
- **Example implementation**: Real-time business metrics dashboard
- **Technical considerations**: Data persistence, memory constraints

4. High-Frequency Trading Systems

- **Why suitable**: Microsecond latency, in-memory computations
- **Example implementation**: Algorithmic trading platform
- **Technical considerations**: Durability, fault tolerance

Multi-Model Databases

1. Unified Customer View Systems

- **Why suitable**: Different data models for different aspects of customer data
- **Example implementation**: 360-degree customer view platform
- **Technical considerations**: Query optimization across models, consistency

2. Content Management with Search

- **Why suitable**: Document storage with search capabilities
- **Example implementation**: Media asset management system
- **Technical considerations**: Index synchronization, query routing

3. IoT Platforms with Mixed Workloads

- **Why suitable**: Time-series for metrics, documents for device info
- **Example implementation**: Industrial IoT management platform
- **Technical considerations**: Model selection for different data types

4. Fraud Detection Systems

- **Why suitable**: Graph for relationships, document for entity data
- **Example implementation**: Financial fraud analysis platform
- **Technical considerations**: Cross-model queries, performance optimization

Interview Questions

Q: How would you choose the right database technology for a high-scale e-commerce platform and why?

A: Choosing the right database technology for a high-scale e-commerce platform requires a decomposition of the system into distinct workloads, each with unique requirements. I would implement a polyglot persistence architecture with specialized databases for different components:

1. Product Catalog and Inventory:

Database Choice: Document Database (MongoDB)

- **Rationale:**
  - Products have varying attributes across categories (flexible schema)
  - Product information is mostly read (read-heavy workload)
  - Catalog needs frequent schema updates as new product types are added
  - Supports rich querying for product search and filtering

Implementation Details:

```
// Product document structure in MongoDB
{
  "_id": ObjectId("..."),
  "sku": "ABC123",
  "name": "Ultra HD Smart TV",
  "brand": "TechBrand",
  "category": "Electronics",
  "subcategory": "Televisions",
  "price": {
    "current": 799.99,
    "msrp": 999.99,
    "currency": "USD"
  },
  "attributes": {
    "size": "55 inch",
```

```
    "resolution": "4K",
    "refreshRate": "120Hz",
    "smartFeatures": ["WiFi", "Bluetooth", "Voice Control"]
  },
  "inventory": {
    "quantity": 45,
    "warehouseLocations": ["EAST-5", "WEST-3"]
  },
  "images": ["url1", "url2", "url3"],
  "rating": 4.7,
  "reviewCount": 128
}

// Indexing strategy
db.products.createIndex({ "name": "text", "brand": "text", "category": 1 });
db.products.createIndex({ "price.current": 1 });
db.products.createIndex({ "inventory.quantity": 1 });
```

2. Order Processing and Payment:

Database Choice: Relational Database (PostgreSQL)

- **Rationale:**
  - Order transactions require ACID properties
  - Complex relational data between orders, items, payments, shipping
  - Financial records need strong consistency
  - Mature tooling for reporting and analysis

Implementation Details:

```
-- Simplified schema for order management
CREATE TABLE orders (
  order_id UUID PRIMARY KEY,
  customer_id UUID NOT NULL REFERENCES customers(customer_id),
  order_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  status VARCHAR(50) NOT NULL DEFAULT 'pending',
  shipping_address_id UUID NOT NULL REFERENCES addresses(address_id),
  payment_method_id UUID NOT NULL REFERENCES payment_methods(payment_method_id),
  subtotal DECIMAL(12,2) NOT NULL,
  tax DECIMAL(12,2) NOT NULL,
  shipping_fee DECIMAL(12,2) NOT NULL,
  total_amount DECIMAL(12,2) NOT NULL,
  promo_code VARCHAR(50)
);

CREATE TABLE order_items (
  order_item_id UUID PRIMARY KEY,
  order_id UUID NOT NULL REFERENCES orders(order_id),
  product_id VARCHAR(100) NOT NULL, -- References MongoDB product
  quantity INTEGER NOT NULL,
  unit_price DECIMAL(12,2) NOT NULL,
  total_price DECIMAL(12,2) NOT NULL
);
```

```
-- Critical transaction for order placement
BEGIN TRANSACTION;
  -- Insert the order
  INSERT INTO orders (...) VALUES (...) RETURNING order_id;

  -- Insert order items
  INSERT INTO order_items (...) VALUES (...);
  INSERT INTO order_items (...) VALUES (...);

  -- Update inventory (via API call to inventory service)
  -- Process payment (via API call to payment service)

  -- If successful, commit
  COMMIT;
```

3. Customer Profiles and Authentication:

Database Choice: Relational Database (PostgreSQL)

- **Rationale:**
  - User authentication requires strong consistency
  - Profile information has a relatively stable structure
  - Integration with order history and payment information
  - Complex queries across customer data

Implementation Details:

```
CREATE TABLE customers (
  customer_id UUID PRIMARY KEY,
  email VARCHAR(255) UNIQUE NOT NULL,
  password_hash VARCHAR(255) NOT NULL,
  first_name VARCHAR(100) NOT NULL,
  last_name VARCHAR(100) NOT NULL,
  phone VARCHAR(20),
  created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
  last_login TIMESTAMP
);

CREATE TABLE addresses (
  address_id UUID PRIMARY KEY,
  customer_id UUID NOT NULL REFERENCES customers(customer_id),
  address_type VARCHAR(20) NOT NULL, -- 'billing' or 'shipping'
  is_default BOOLEAN NOT NULL DEFAULT false,
  street_address VARCHAR(255) NOT NULL,
  city VARCHAR(100) NOT NULL,
  state VARCHAR(100) NOT NULL,
  postal_code VARCHAR(20) NOT NULL,
  country VARCHAR(100) NOT NULL
);
```

4. Session Management and Shopping Cart:

Database Choice: In-Memory Database (Redis)

- **Rationale:**
  - Session data requires very low latency access
  - Shopping carts need high availability and quick updates
  - Temporary data benefits from automatic expiration
  - High concurrent access during peak shopping periods

Implementation Details:

```
// Redis data structures for cart management

// Cart contents (Hash)
HMSET cart:user123 product:123 2 product:456 1 product:789 3

// Cart metadata (Hash)
HMSET cart:user123:meta last_updated 1636988400 currency USD

// Cart expiration (if abandoned)
EXPIRE cart:user123 86400      // Expire in 24 hours
EXPIRE cart:user123:meta 86400

// Session data
HSET session:abc123 user_id user123 logged_in true
EXPIRE session:abc123 3600     // Expire in 1 hour
```

5. Product Search:

Database Choice: Search Engine (Elasticsearch)

- **Rationale:**
  - Full-text search capabilities
  - Faceted navigation for filtering
  - Relevance scoring for search results
  - Support for synonyms and search suggestions

Implementation Details:

```
// Elasticsearch mapping for products
{
  "mappings": {
    "properties": {
      "name": {
        "type": "text",
        "analyzer": "english",
        "fields": {
          "keyword": { "type": "keyword" }
        }
      },
      "description": { "type": "text", "analyzer": "english" },
      "brand": { "type": "keyword" },
```

```
      "category": { "type": "keyword" },
      "subcategory": { "type": "keyword" },
      "price": { "type": "float" },
      "attributes": {
        "properties": {
          "size": { "type": "keyword" },
          "color": { "type": "keyword" },
          "material": { "type": "keyword" }
          // Other common attributes
        }
      },
      "in_stock": { "type": "boolean" },
      "rating": { "type": "float" }
    }
  }
}

// Complex search query
{
  "query": {
    "bool": {
      "must": [
        { "match": { "name": "smartphone waterproof" } }
      ],
      "filter": [
        { "term": { "category": "Electronics" } },
        { "term": { "in_stock": true } },
        { "range": { "price": { "lte": 1000 } } }
      ]
    }
  },
  "sort": [
    { "_score": "desc" },
    { "rating": "desc" }
  ],
  "aggs": {
    "brands": {
      "terms": { "field": "brand", "size": 10 }
    },
    "price_ranges": {
      "range": {
        "field": "price",
        "ranges": [
          { "to": 300 },
          { "from": 300, "to": 600 },
          { "from": 600, "to": 1000 },
          { "from": 1000 }
        ]
      }
    }
  }
}
```

6. Recommendations and Personalization:

Database Choice: Graph Database (Neo4j)

- **Rationale:**
  - Complex relationship modeling (users, products, categories)
  - Efficient traversal for finding product relationships
  - Pattern recognition for recommendation algorithms
  - Adaptability to changing recommendation strategies

Implementation Details:

```
// Neo4j recommendation queries

// Find products purchased by similar customers
MATCH (u:User {id: 'user123'})-[:PURCHASED]->(p:Product)<-[:PURCHASED]-(other:User),
      (other)-[:PURCHASED]->(rec:Product)
WHERE NOT (u)-[:PURCHASED]->(rec)
RETURN rec.name, COUNT(distinct other) AS frequency
ORDER BY frequency DESC
LIMIT 10;

// Find products frequently purchased together
MATCH (p:Product {id: 'prod123'})<-[:CONTAINS]-(o:Order)-[:CONTAINS]->(rec:Product)
WHERE p <> rec
RETURN rec.name, COUNT(o) AS frequency
ORDER BY frequency DESC
LIMIT 5;
```

7. Analytics and Reporting:

Database Choice: Columnar Database (Snowflake/BigQuery)

- **Rationale:**
  - Optimized for analytical queries
  - Efficient for large dataset aggregations
  - Decoupled from operational databases
  - Scalable for growing data volumes

Implementation Details:

```
-- Example Snowflake analytics query
SELECT
  DATE_TRUNC('month', o.order_date) AS month,
  p.category,
  p.brand,
  SUM(oi.quantity) AS units_sold,
  SUM(oi.total_price) AS revenue,
  COUNT(DISTINCT o.customer_id) AS unique_customers
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products_dim p ON oi.product_id = p.product_id
WHERE o.order_date BETWEEN '2023-01-01' AND '2023-12-31'
```

```
  AND o.status = 'completed'
GROUP BY 1, 2, 3
ORDER BY 1, 4 DESC;
```

8. Caching Layer:

Database Choice: Redis

- **Rationale:**
  - Extremely fast response times for frequently accessed data
  - Support for various data structures
  - Automatic expiration for stale data
  - Reduces load on primary databases

Implementation Details:

```
# Example caching implementation
def get_product_details(product_id):
    cache_key = f"product:{product_id}"
    cached_data = redis_client.get(cache_key)

    if cached_data:
        return json.loads(cached_data)

    # Cache miss - get from MongoDB
    product = mongo_db.products.find_one({"_id": ObjectId(product_id)})

    # Cache for 10 minutes
    redis_client.setex(cache_key, 600, json.dumps(product))
    return product
```

Integration Strategy:

1. **Data Synchronization:**
   - Use Change Data Capture (CDC) to keep product information synchronized
   - Maintain a single source of truth for each data domain
   - Implement eventual consistency where appropriate
2. **Service-Oriented Architecture:**
   - Implement domain-specific microservices around each database
   - Use API gateways to compose data from multiple sources
   - Enforce access patterns through service interfaces
3. **Scalability Approach:**
   - Horizontal scaling for MongoDB and Elasticsearch
   - Read replicas for PostgreSQL
   - Redis cluster for distributed caching
   - Sharding strategies based on customer regions

4. **Resilience Strategy:**
    - Circuit breakers between services
    - Fallback mechanisms when dependencies are unavailable
    - Asynchronous processing for non-critical operations

This polyglot persistence approach leverages the strengths of each database type for specific workloads, resulting in a high-performance, scalable e-commerce platform that can handle diverse requirements and growing data volumes.

Q: Describe how you would design a database solution for a real-time analytics platform tracking user behavior across multiple channels.

A: Designing a database solution for a real-time analytics platform tracking cross-channel user behavior requires an architecture that can handle high ingestion rates, perform real-time analysis, and support both streaming and historical analytics. Here's my comprehensive approach:

System Requirements Analysis

1. **Data Characteristics**:
    - High volume: Millions of events per minute
    - Diverse event types across web, mobile, and other channels
    - Variable schema as new event types emerge
    - Temporal nature with recent data being most frequently accessed
2. **Query Patterns**:
    - Real-time dashboards with sub-second latency
    - Historical trend analysis with flexible time ranges
    - User journey analysis across channels
    - Funnel analysis and conversion tracking
    - Ad-hoc exploration and segmentation
3. **Performance Requirements**:
    - Sub-second query response for real-time dashboards
    - High write throughput for event ingestion
    - Scalability to handle traffic spikes
    - Cost-effective storage for historical data

Multi-Tier Database Architecture

I propose a multi-tier database architecture that separates concerns while optimizing for different workloads:

```
[Event Sources] → [Data Ingestion] → [Hot Storage] → [Warm Storage] → [Cold Storage]
                          ↓                ↓              ↓               ↓
                  [Stream Processing] → [Real-time Analytics] → [Historical
Analytics]
```

1. Data Ingestion Layer

Technology Choice: Apache Kafka

- Acts as a buffer to absorb traffic spikes
- Enables parallel processing of event streams
- Provides event replay capabilities
- Decouples producers from consumers

Schema Management:

```
// Kafka message schema using Avro
{
  "type": "record",
  "name": "UserEvent",
  "fields": [
    { "name": "event_id", "type": "string" },
    { "name": "user_id", "type": "string" },
    { "name": "session_id", "type": "string" },
    { "name": "timestamp", "type": "long" },
    { "name": "channel", "type": "string" },
    { "name": "event_type", "type": "string" },
    {
      "name": "device_info",
      "type": {
        "type": "record",
        "name": "DeviceInfo",
        "fields": [
          { "name": "type", "type": "string" },
          { "name": "os", "type": "string" },
          { "name": "browser", "type": ["null", "string"] },
          { "name": "app_version", "type": ["null", "string"] }
        ]
      }
    },
    {
      "name": "geo_location",
      "type": {
        "type": "record",
        "name": "GeoLocation",
        "fields": [
          { "name": "country", "type": "string" },
          { "name": "region", "type": ["null", "string"] },
          { "name": "city", "type": ["null", "string"] },
          { "name": "latitude", "type": ["null", "double"] },
          { "name": "longitude", "type": ["null", "double"] }
        ]
      }
    },
    {
      "name": "event_properties",
      "type": "map",
      "values": ["null", "string", "int", "double", "boolean"]
    }
  ]
}
```

Partitioning Strategy:

- Partition by user_id to maintain event order per user
- Configure retention policy based on downstream processing needs

2. Stream Processing Layer

Technology Choice: Apache Flink

- Provides stateful stream processing
- Enables complex event processing and pattern detection
- Supports event-time processing with late-event handling
- Allows for windowed aggregations

Processing Examples:

```
// Flink stream processing for session analytics
DataStream<UserEvent> events = ...

// Compute session metrics in 1-minute tumbling windows
events
    .keyBy(event -> event.getSessionId())
    .window(TumblingEventTimeWindows.of(Time.minutes(1)))
    .process(new SessionMetricsProcessor())
    .addSink(new ClickhouseSessionSink());

// Detect conversion funnels
Pattern<UserEvent, ?> conversionPattern = Pattern.<UserEvent>begin("pageView")
    .where(event -> event.getEventType().equals("page_view") &&
                   event.getEventProperties().get("page").equals("product"))
    .followedBy("addToCart")
    .where(event -> event.getEventType().equals("add_to_cart"))
    .followedBy("checkout")
    .where(event -> event.getEventType().equals("checkout"))
    .within(Time.hours(24));

PatternStream<UserEvent> patternStream = CEP.pattern(events, conversionPattern);
patternStream.select(new ConversionPatternProcessor())
             .addSink(new RedisConversionSink());
```

3. Hot Storage Layer (Real-time Analytics)

Technology Choice: ClickHouse

- Columnar database optimized for analytics
- Excellent insert and query performance
- Support for real-time aggregations
- Efficient storage with high compression

Schema Design:

```
-- ClickHouse table for event data
CREATE TABLE events (
    event_id String,
    user_id String,
    session_id String,
    timestamp DateTime64(3),
    date Date DEFAULT toDate(timestamp),
    channel String,
    event_type String,
    device_type String,
    device_os String,
    device_browser Nullable(String),
    device_app_version Nullable(String),
    country String,
    region Nullable(String),
    city Nullable(String),
    latitude Nullable(Float64),
    longitude Nullable(Float64),
    -- Store event properties in a nested structure
    properties Nested (
        key String,
        value String
    )
) ENGINE = MergeTree()
PARTITION BY toYYYYMM(date)
ORDER BY (date, user_id, timestamp)
TTL date + INTERVAL 7 DAY TO VOLUME 'hot',
    date + INTERVAL 30 DAY TO VOLUME 'warm',
    date + INTERVAL 180 DAY TO VOLUME 'cold';

-- Materialized view for real-time event counts
CREATE MATERIALIZED VIEW event_counts_mv
ENGINE = SummingMergeTree()
PARTITION BY toYYYYMM(date)
ORDER BY (date, hour, event_type, channel)
POPULATE
AS SELECT
    date,
    toHour(timestamp) AS hour,
    event_type,
    channel,
    count() AS events_count,
    uniqCombined(user_id) AS unique_users
FROM events
GROUP BY date, hour, event_type, channel;
```

Query Examples:

```
-- Real-time event count for the last hour
SELECT
    event_type,
    channel,
```

```
    sum(events_count) AS total_events,
    sum(unique_users) AS total_unique_users
FROM event_counts_mv
WHERE date = today() AND hour >= toHour(now()) - 1
GROUP BY event_type, channel
ORDER BY total_events DESC;

-- User journey analysis (single user)
SELECT
    event_type,
    timestamp,
    channel,
    device_type,
    arrayJoin(arrayZip(properties.key, properties.value)) AS property
FROM events
WHERE user_id = 'user123'
  AND date BETWEEN today() - 7 AND today()
ORDER BY timestamp;

-- Conversion rate by channel (last 24 hours)
WITH
page_views AS (
    SELECT channel, uniqCombined(user_id) AS users
    FROM events
    WHERE date >= today() - 1
      AND event_type = 'page_view'
      AND has(properties.key, 'page') AND
          arrayElement(properties.value, indexOf(properties.key, 'page')) =
'product'
    GROUP BY channel
),
purchases AS (
    SELECT channel, uniqCombined(user_id) AS users
    FROM events
    WHERE date >= today() - 1
      AND event_type = 'purchase'
    GROUP BY channel
)
SELECT
    pv.channel,
    pv.users AS view_users,
    p.users AS purchase_users,
    round(p.users / pv.users * 100, 2) AS conversion_rate
FROM page_views pv
LEFT JOIN purchases p ON pv.channel = p.channel
ORDER BY conversion_rate DESC;
```

4. Warm/Cold Storage Layer (Historical Analytics)

Technology Choice: Combination of ClickHouse (Warm) and S3/Parquet (Cold)

- ClickHouse for warm storage (1-6 months)
- S3 with Parquet for cold storage (6+ months)
- Integration with query engines like Presto/Athena for cold data

Data Lifecycle Management:

```
-- ClickHouse tiered storage configuration
ALTER TABLE events MODIFY SETTING storage_policy = 'tiered';

-- Automatically move older data to cold storage
CREATE MATERIALIZED VIEW events_to_s3_mv TO s3_engine
AS SELECT * FROM events
WHERE date < today() - 180;

-- External table definition for cold storage
CREATE TABLE events_cold (
    -- Same schema as events table
)
ENGINE = S3('s3://analytics-bucket/events/{partition_id}/*.parquet',
            'AccessKeyID', 'SecretAccessKey', 'Parquet');
```

5. In-Memory Layer (Serving Layer)

Technology Choice: Redis + Time Series Module

- Ultra-low latency for dashboard metrics
- Support for time-series data
- Pub/sub for real-time updates
- Automatic downsampling

Implementation Details:

```
# Store real-time metrics in Redis Time Series
def update_dashboard_metrics(metrics_batch):
    redis_pipe = redis_client.pipeline()

    for metric in metrics_batch:
        # Create time-series if doesn't exist
        try:
            redis_client.ts().create(
                f"metric:{metric['name']}:{metric['dimensions']}",
                retention_msecs=86400000,  # 24 hours
                labels={
                    "name": metric["name"],
                    **parse_dimensions(metric["dimensions"])
                }
            )
        except:
            pass  # Already exists

        # Add data point
        redis_pipe.ts().add(
            f"metric:{metric['name']}:{metric['dimensions']}",
            metric["timestamp"],
            metric["value"]
```

```
            )

        redis_pipe.execute()

    # Get dashboard data
    def get_dashboard_metrics(metric_name, dimensions, start_time, end_time,
    aggregation="avg", bucket_size_ms=60000):
        # Get raw time series or aggregation
        if aggregation:
            return redis_client.ts().range(
                f"metric:{metric_name}:{dimensions}",
                start_time,
                end_time,
                aggregation_type=aggregation,
                bucket_size_msec=bucket_size_ms
            )
        else:
            return redis_client.ts().range(
                f"metric:{metric_name}:{dimensions}",
                start_time,
                end_time
            )
```

Integration and Data Flow

The complete data flow looks like this:

1. **Event Collection**:
    - Web/mobile events sent to collection endpoints
    - Validation and enrichment at the edge
    - Batched and sent to Kafka topics
2. **Stream Processing**:
    - Flink reads from Kafka topics
    - Performs windowed aggregations and pattern detection
    - Outputs to both ClickHouse and Redis
3. **Storage and Query**:
    - Real-time metrics stored in Redis Time Series
    - Recent event data stored in ClickHouse hot storage
    - Historical data automatically moved to warm/cold storage
    - Query federation across storage tiers for analytics spanning different time ranges
4. **Dashboard and Analysis**:
    - Real-time dashboards read from Redis
    - Interactive analytics query ClickHouse
    - Long-term analysis uses federated queries across tiers

Handling Cross-Channel User Identification

A critical aspect of cross-channel analytics is identifying the same user across different channels and devices:

```
-- ClickHouse table for identity resolution
CREATE TABLE user_identities (
    anonymous_id String,
    user_id String,
    identity_type String, -- email, phone, device_id, etc.
    identity_value String,
    first_seen DateTime64(3),
    last_seen DateTime64(3),
    confidence Float64
) ENGINE = ReplacingMergeTree(last_seen)
ORDER BY (identity_type, identity_value, anonymous_id, user_id);

-- Query to get unified user profile
WITH user_devices AS (
    SELECT user_id, anonymous_id
    FROM user_identities
    WHERE identity_value = 'john.doe@example.com'
      AND identity_type = 'email'
)
SELECT
    event_type,
    timestamp,
    channel,
    device_type
FROM events
WHERE user_id IN (SELECT user_id FROM user_devices)
   OR session_id IN (SELECT anonymous_id FROM user_devices)
ORDER BY timestamp DESC
LIMIT 100;
```

Scaling Considerations

1. **Write Scaling**:
    - Kafka cluster sized for peak ingest with buffer capacity
    - ClickHouse sharded by user_id or date ranges
    - Automatic partitioning for time-based data
2. **Query Scaling**:
    - Redis cluster for in-memory metrics
    - ClickHouse read replicas for query-heavy workloads
    - Query routing based on time range and freshness requirements
3. **Cost Optimization**:
    - Tiered storage with automated data lifecycle
    - Materialized views for common query patterns
    - Aggregation tables for historical data

Monitoring and Operations

1. **Data Quality Monitoring**:
    - Schema validation at ingestion
    - Data freshness and completeness metrics
    - Anomaly detection on event volumes
2. **Performance Monitoring**:
    - Query latency tracking
    - Resource utilization metrics
    - Alerting on SLA violations
3. **Operational Procedures**:
    - Automatic scaling based on load
    - Disaster recovery procedures with regular testing
    - Data retention policy enforcement

This architecture provides a robust foundation for a real-time analytics platform capable of tracking user behavior across multiple channels, delivering both real-time insights and historical analysis while efficiently managing large volumes of data.

3. SQL and Relational Databases

3.1 SQL Fundamentals

SQL (Structured Query Language) is the standard language for interacting with relational databases. Understanding SQL fundamentals is essential for working with any relational database system.

Basic SQL Syntax

SELECT Statement:

```
-- Basic SELECT statement
SELECT column1, column2
FROM table_name
WHERE condition
ORDER BY column1 [ASC|DESC]
LIMIT number;

-- Example: Get all products with price greater than $100, sorted by price
SELECT product_name, price, category
FROM products
WHERE price > 100
ORDER BY price DESC
LIMIT 10;
```

INSERT Statement:

```sql
-- Basic INSERT statement
INSERT INTO table_name (column1, column2, column3)
VALUES (value1, value2, value3);

-- Example: Add a new product
INSERT INTO products (product_name, price, category, in_stock)
VALUES ('Wireless Headphones', 89.99, 'Electronics', true);

-- Insert multiple rows
INSERT INTO products (product_name, price, category)
VALUES
  ('Laptop Stand', 29.99, 'Accessories'),
  ('USB-C Cable', 12.99, 'Accessories'),
  ('Wireless Mouse', 24.99, 'Electronics');
```

UPDATE Statement:

```sql
-- Basic UPDATE statement
UPDATE table_name
SET column1 = value1, column2 = value2
WHERE condition;

-- Example: Update product price and stock status
UPDATE products
SET price = 79.99, in_stock = false
WHERE product_id = 123;
```

DELETE Statement:

```sql
-- Basic DELETE statement
DELETE FROM table_name
WHERE condition;

-- Example: Delete discontinued products
DELETE FROM products
WHERE discontinued = true AND last_ordered < '2022-01-01';
```

Data Types

Common SQL data types across most relational databases:

Numeric Types:

- **INTEGER/INT**: Whole numbers
- **SMALLINT**: Small-range whole numbers
- **BIGINT**: Large-range whole numbers
- **DECIMAL/NUMERIC(p,s)**: Exact decimal numbers with precision p and scale s
- **FLOAT/REAL**: Approximate floating-point numbers

- **DOUBLE PRECISION**: Higher precision floating-point numbers

String Types:

- **CHAR(n)**: Fixed-length character strings
- **VARCHAR(n)**: Variable-length character strings
- **TEXT**: Variable-length character strings with large capacity

Date and Time Types:

- **DATE**: Calendar date (year, month, day)
- **TIME**: Time of day
- **TIMESTAMP**: Date and time
- **INTERVAL**: Period of time

Boolean Type:

- **BOOLEAN**: True/false values

Other Types:

- **BLOB/BINARY**: Binary data
- **JSON**: JSON data (in modern RDBMSs)
- **UUID**: Universally unique identifiers
- **ARRAY**: Array of values (in some RDBMSs)

Example of Type Usage:

```
CREATE TABLE employees (
    employee_id INTEGER PRIMARY KEY,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    birth_date DATE,
    hire_date DATE NOT NULL,
    salary DECIMAL(10, 2) NOT NULL,
    is_active BOOLEAN DEFAULT true,
    department VARCHAR(100),
    job_description TEXT,
    profile_photo BLOB,
    last_login TIMESTAMP,
    settings JSON
);
```

Filtering Data

WHERE Clause Operators:

```
-- Comparison operators
SELECT * FROM products WHERE price > 100;
SELECT * FROM products WHERE category = 'Electronics';
SELECT * FROM products WHERE launch_date < '2023-01-01';
```

```
-- Logical operators
SELECT * FROM products WHERE price > 100 AND category = 'Electronics';
SELECT * FROM products WHERE category = 'Electronics' OR category = 'Accessories';
SELECT * FROM products WHERE NOT discontinued;

-- BETWEEN operator
SELECT * FROM products WHERE price BETWEEN 50 AND 150;

-- IN operator
SELECT * FROM products WHERE category IN ('Electronics', 'Computers',
'Accessories');

-- LIKE operator (pattern matching)
SELECT * FROM products WHERE product_name LIKE 'Wireless%'; -- Starts with
'Wireless'
SELECT * FROM products WHERE product_name LIKE '%Headphone%'; -- Contains
'Headphone'
SELECT * FROM products WHERE product_name LIKE '_Phone'; -- Exactly 6 characters,
ending with 'Phone'

-- NULL checks
SELECT * FROM customers WHERE phone IS NULL;
SELECT * FROM customers WHERE phone IS NOT NULL;
```

Sorting and Limiting Results

```
-- Basic sorting
SELECT * FROM products ORDER BY price ASC; -- Ascending order (default)
SELECT * FROM products ORDER BY price DESC; -- Descending order

-- Multiple sort criteria
SELECT * FROM products ORDER BY category ASC, price DESC;

-- Limiting results
SELECT * FROM products ORDER BY price DESC LIMIT 10; -- Top 10 most expensive
products

-- Pagination
SELECT * FROM products ORDER BY product_id LIMIT 20 OFFSET 40; -- Items 41-60
```

Aggregation Functions

```
-- COUNT
SELECT COUNT(*) FROM orders; -- Count all rows
SELECT COUNT(customer_id) FROM orders; -- Count non-NULL values
SELECT COUNT(DISTINCT customer_id) FROM orders; -- Count unique values

-- SUM
SELECT SUM(order_total) FROM orders WHERE order_date >= '2023-01-01';
```

```sql
-- AVG
SELECT AVG(order_total) FROM orders;

-- MIN/MAX
SELECT MIN(price) AS lowest_price, MAX(price) AS highest_price FROM products;

-- GROUP BY
SELECT category, COUNT(*) AS product_count, AVG(price) AS avg_price
FROM products
GROUP BY category
ORDER BY product_count DESC;

-- HAVING (filters on aggregated results)
SELECT category, COUNT(*) AS product_count, AVG(price) AS avg_price
FROM products
GROUP BY category
HAVING COUNT(*) > 10 AND AVG(price) < 100
ORDER BY product_count DESC;
```

Joining Tables

```sql
-- INNER JOIN
SELECT o.order_id, o.order_date, c.customer_name, c.email
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id;

-- LEFT JOIN
SELECT p.product_name, p.price, COALESCE(SUM(oi.quantity), 0) AS units_sold
FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name, p.price;

-- RIGHT JOIN
SELECT e.employee_name, d.department_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id;

-- FULL OUTER JOIN
SELECT e.employee_name, p.project_name
FROM employees e
FULL OUTER JOIN projects p ON e.project_id = p.project_id;

-- CROSS JOIN
SELECT e.employee_name, s.skill_name
FROM employees e
CROSS JOIN skills s;

-- Self JOIN
SELECT e1.employee_name AS employee, e2.employee_name AS manager
FROM employees e1
JOIN employees e2 ON e1.manager_id = e2.employee_id;
```

Subqueries

```
-- Subquery in WHERE clause
SELECT product_name, price
FROM products
WHERE price > (SELECT AVG(price) FROM products);

-- Subquery in FROM clause
SELECT category, avg_price
FROM (
    SELECT category, AVG(price) AS avg_price
    FROM products
    GROUP BY category
) AS category_averages
WHERE avg_price > 100;

-- Subquery with IN
SELECT customer_name, email
FROM customers
WHERE customer_id IN (
    SELECT DISTINCT customer_id
    FROM orders
    WHERE order_date >= '2023-01-01'
);

-- Correlated subquery
SELECT p.product_name, p.price
FROM products p
WHERE p.price > (
    SELECT AVG(price)
    FROM products
    WHERE category = p.category
);

-- EXISTS
SELECT c.customer_name
FROM customers c
WHERE EXISTS (
    SELECT 1
    FROM orders o
    WHERE o.customer_id = c.customer_id
    AND o.order_date >= '2023-01-01'
);
```

SET Operations

```
-- UNION (removes duplicates)
SELECT product_id, product_name FROM discontinued_products
UNION
SELECT product_id, product_name FROM current_products;
```

```
-- UNION ALL (keeps duplicates)
SELECT product_id, product_name FROM discontinued_products
UNION ALL
SELECT product_id, product_name FROM current_products;

-- INTERSECT
SELECT product_id FROM products_2022
INTERSECT
SELECT product_id FROM products_2023;

-- EXCEPT/MINUS
SELECT product_id FROM products_2022
EXCEPT  -- Use MINUS in Oracle
SELECT product_id FROM products_2023;
```

Interview Questions

Q: What is the difference between WHERE and HAVING clauses in SQL?

A: The WHERE and HAVING clauses serve different purposes in SQL queries:

WHERE Clause:

- Filters rows before any grouping occurs
- Applied directly to individual rows in the base tables
- Cannot reference aggregate functions (like COUNT, SUM, AVG)
- Processed before GROUP BY
- More efficient as it reduces the number of rows early in query processing

HAVING Clause:

- Filters groups after the GROUP BY operation
- Applied to the results of grouping
- Can reference aggregate functions
- Processed after GROUP BY
- Less efficient as filtering happens after rows are already grouped

Example:

```
SELECT department_id, department_name, AVG(salary) AS avg_salary, COUNT(*) AS
employee_count
FROM employees
WHERE hire_date > '2020-01-01'  -- Filters individual employees before grouping
GROUP BY department_id, department_name
HAVING AVG(salary) > 50000      -- Filters departments after grouping
   AND COUNT(*) >= 5;           -- Only departments with 5+ employees
```

In this example:

1. The WHERE clause filters out employees hired before 2020-01-01

2. Remaining employees are grouped by department
3. The HAVING clause filters out departments where the average salary is 50000 or less, or with fewer than 5 employees

Using WHERE is more efficient when you can apply the filter before grouping, as it reduces the amount of data that needs to be processed in the GROUP BY operation.

Q: Explain the different types of JOINs in SQL with examples.

A: SQL supports several types of JOINs for combining data from multiple tables:

1. INNER JOIN:

- Returns only rows where there is a match in both tables
- Most common join type
- Syntax: `FROM table1 INNER JOIN table2 ON table1.column = table2.column`

```
-- Find all orders with customer information
SELECT o.order_id, o.order_date, c.customer_name, c.email
FROM orders o
INNER JOIN customers c ON o.customer_id = c.customer_id;
```

2. LEFT JOIN (or LEFT OUTER JOIN):

- Returns all rows from the left table and matching rows from the right table
- If no match exists in right table, NULL values are returned
- Useful for finding "missing" relationships

```
-- Find all products and their order quantities (including products never ordered)
SELECT p.product_id, p.product_name,
       COALESCE(SUM(oi.quantity), 0) AS total_ordered
FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name;
```

3. RIGHT JOIN (or RIGHT OUTER JOIN):

- Returns all rows from the right table and matching rows from the left table
- If no match exists in left table, NULL values are returned
- Less commonly used (can usually be rewritten as LEFT JOIN)

```
-- Find all departments and employees assigned to them (including empty departments)
SELECT d.department_name, e.employee_name
FROM employees e
RIGHT JOIN departments d ON e.department_id = d.department_id;
```

4. FULL JOIN (or FULL OUTER JOIN):

- Returns all rows when there is a match in either left or right table
- If no match exists, NULL values are returned for columns from the table without a match
- Used when you need to see all possible combinations

```
-- Match students with courses, showing all students and all courses
SELECT s.student_name, c.course_name
FROM students s
FULL OUTER JOIN enrollments e ON s.student_id = e.student_id
FULL OUTER JOIN courses c ON e.course_id = c.course_id;
```

5. CROSS JOIN:

- Returns the Cartesian product of both tables
- Every row from first table is paired with every row from second table
- Does not use a join condition
- Results in m × n rows (where m and n are the numbers of rows in the two tables)

```
-- Create all possible product-color combinations
SELECT p.product_name, c.color_name
FROM products p
CROSS JOIN colors c;
```

6. SELF JOIN:

- Not a distinct join type but a special use case
- Joins a table to itself
- Useful for hierarchical data or comparing rows within the same table

```
-- Find employees and their managers
SELECT e.employee_name AS employee,
       m.employee_name AS manager
FROM employees e
LEFT JOIN employees m ON e.manager_id = m.employee_id;
```

7. NATURAL JOIN:

- Automatically joins tables based on columns with the same name
- No explicit join condition required
- Generally avoided in practice due to lack of control

```
-- Implicitly joins on all columns with same name
SELECT employee_name, department_name
FROM employees
NATURAL JOIN departments;
```

Visual Representation:

```
Table A        Table B
  1 FOO          1 BAR
  2 ABC          3 XYZ
  3 DEF          5 QWE

INNER JOIN: Rows where A.id = B.id
  1 FOO BAR
  3 DEF XYZ

LEFT JOIN: All rows from A, matching rows from B
  1 FOO BAR
  2 ABC NULL
  3 DEF XYZ

RIGHT JOIN: All rows from B, matching rows from A
  1 FOO BAR
  3 DEF XYZ
  5 NULL QWE

FULL JOIN: All rows from both tables
  1 FOO BAR
  2 ABC NULL
  3 DEF XYZ
  5 NULL QWE

CROSS JOIN: Every possible combination
  1 FOO 1 BAR
  1 FOO 3 XYZ
  1 FOO 5 QWE
  2 ABC 1 BAR
  2 ABC 3 XYZ
  2 ABC 5 QWE
  3 DEF 1 BAR
  3 DEF 3 XYZ
  3 DEF 5 QWE
```

The choice of join type depends on your specific requirements and the relationship between the tables. INNER JOIN is most commonly used when you only need matching data, while outer joins are valuable when you need to include rows that don't have matches.

3.2 Advanced SQL Features

Beyond the basics, advanced SQL features enable more complex and efficient data manipulation and analysis.

Window Functions

Window functions perform calculations across a set of rows related to the current row, without grouping the rows into a single output row.

```sql
-- Basic window function syntax
SELECT column1, column2,
       FUNCTION() OVER ([PARTITION BY column1] [ORDER BY column2] [frame_clause])
FROM table_name;

-- ROW_NUMBER(): Assigns unique numbers to rows
SELECT product_name, category, price,
       ROW_NUMBER() OVER (ORDER BY price DESC) AS price_rank
FROM products;

-- RANK() and DENSE_RANK(): Assign ranks to rows (with and without gaps)
SELECT product_name, category, price,
       RANK() OVER (PARTITION BY category ORDER BY price DESC) AS
category_price_rank,
       DENSE_RANK() OVER (PARTITION BY category ORDER BY price DESC) AS
category_price_dense_rank
FROM products;

-- NTILE(): Divides rows into specified number of groups
SELECT product_name, price,
       NTILE(4) OVER (ORDER BY price) AS price_quartile
FROM products;

-- Window aggregate functions
SELECT product_name, category, price,
       AVG(price) OVER (PARTITION BY category) AS category_avg_price,
       price - AVG(price) OVER (PARTITION BY category) AS diff_from_avg
FROM products;

-- LAG() and LEAD(): Access previous or next row values
SELECT product_name, launch_date, price,
       LAG(price) OVER (ORDER BY launch_date) AS previous_product_price,
       LEAD(price) OVER (ORDER BY launch_date) AS next_product_price
FROM products;

-- FIRST_VALUE() and LAST_VALUE(): Get first/last value in window
SELECT product_name, category, price,
       FIRST_VALUE(product_name) OVER (PARTITION BY category ORDER BY price DESC) AS
most_expensive_in_category
FROM products;

-- Running totals and moving averages
SELECT order_date, order_total,
       SUM(order_total) OVER (ORDER BY order_date) AS running_total,
       AVG(order_total) OVER (ORDER BY order_date ROWS BETWEEN 2 PRECEDING AND
CURRENT ROW) AS moving_avg_3days
FROM orders;
```

Common Table Expressions (CTEs)

CTEs provide a way to create named temporary result sets that exist for the duration of a query.

```sql
-- Basic CTE syntax
WITH cte_name AS (
    SELECT column1, column2
    FROM table_name
    WHERE condition
)
SELECT * FROM cte_name;

-- Using multiple CTEs
WITH category_stats AS (
    SELECT category, COUNT(*) AS product_count, AVG(price) AS avg_price
    FROM products
    GROUP BY category
),
high_value_categories AS (
    SELECT category, product_count, avg_price
    FROM category_stats
    WHERE avg_price > 100
)
SELECT * FROM high_value_categories
ORDER BY avg_price DESC;

-- Recursive CTEs (hierarchical data)
WITH RECURSIVE employee_hierarchy AS (
    -- Base case: top-level employees (no manager)
    SELECT employee_id, employee_name, manager_id, 1 AS level
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive case: employees with managers
    SELECT e.employee_id, e.employee_name, e.manager_id, eh.level + 1
    FROM employees e
    JOIN employee_hierarchy eh ON e.manager_id = eh.employee_id
)
SELECT * FROM employee_hierarchy
ORDER BY level, employee_name;
```

Stored Procedures and Functions

Stored procedures and functions are pre-compiled SQL code that can be reused.

```sql
-- PostgreSQL function
CREATE OR REPLACE FUNCTION get_product_stats(
    category_param VARCHAR
) RETURNS TABLE (
    category VARCHAR,
    product_count BIGINT,
    avg_price NUMERIC,
    min_price NUMERIC,
    max_price NUMERIC
```

```
) AS $$
BEGIN
    RETURN QUERY
    SELECT
        p.category,
        COUNT(*) AS product_count,
        AVG(p.price) AS avg_price,
        MIN(p.price) AS min_price,
        MAX(p.price) AS max_price
    FROM products p
    WHERE p.category = category_param
    GROUP BY p.category;
END;
$$ LANGUAGE plpgsql;

-- Usage
SELECT * FROM get_product_stats('Electronics');

-- MySQL stored procedure
DELIMITER //
CREATE PROCEDURE update_product_price(
    IN product_id_param INT,
    IN new_price DECIMAL(10,2)
)
BEGIN
    DECLARE current_price DECIMAL(10,2);

    -- Get current price
    SELECT price INTO current_price
    FROM products
    WHERE product_id = product_id_param;

    -- Update price
    UPDATE products
    SET price = new_price,
        last_updated = NOW()
    WHERE product_id = product_id_param;

    -- Log price change
    INSERT INTO price_change_log (product_id, old_price, new_price, change_date)
    VALUES (product_id_param, current_price, new_price, NOW());
END //
DELIMITER ;

-- Usage
CALL update_product_price(123, 49.99);
```

Transactions

Transactions allow multiple operations to be executed as a single unit of work.

```
-- Basic transaction
START TRANSACTION;
```

```
-- or BEGIN;

-- Perform operations
UPDATE accounts SET balance = balance - 100 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 100 WHERE account_id = 456;

-- Check conditions
SELECT @new_balance := balance FROM accounts WHERE account_id = 123;
IF @new_balance < 0 THEN
    ROLLBACK;
ELSE
    COMMIT;
END IF;

-- Transaction with savepoint
BEGIN;
UPDATE inventory SET quantity = quantity - 5 WHERE product_id = 101;

SAVEPOINT after_inventory_update;

INSERT INTO order_items (order_id, product_id, quantity, price)
VALUES (1001, 101, 5, 25.99);

-- Oops, something went wrong
ROLLBACK TO SAVEPOINT after_inventory_update;

-- Try a different product
INSERT INTO order_items (order_id, product_id, quantity, price)
VALUES (1001, 102, 3, 19.99);

COMMIT;
```

Triggers

Triggers are special stored procedures that automatically execute when specific events occur.

```
-- MySQL trigger example
DELIMITER //
CREATE TRIGGER after_product_insert
AFTER INSERT ON products
FOR EACH ROW
BEGIN
    -- Log product creation
    INSERT INTO product_audit (product_id, action, action_date, user)
    VALUES (NEW.product_id, 'INSERT', NOW(), CURRENT_USER());

    -- Update product count
    UPDATE product_categories
    SET product_count = product_count + 1
    WHERE category_id = NEW.category_id;
END //
DELIMITER ;
```

```
-- PostgreSQL trigger example
CREATE OR REPLACE FUNCTION update_last_modified()
RETURNS TRIGGER AS $$
BEGIN
    NEW.last_modified = NOW();
    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER update_product_last_modified
BEFORE UPDATE ON products
FOR EACH ROW
EXECUTE FUNCTION update_last_modified();
```

Views

Views are virtual tables based on the result of a SQL statement.

```
-- Simple view
CREATE VIEW high_value_products AS
SELECT product_id, product_name, price, category
FROM products
WHERE price > 1000;

-- Updatable view (with limitations)
CREATE VIEW electronics_products AS
SELECT product_id, product_name, price, in_stock
FROM products
WHERE category = 'Electronics';

-- Using WITH CHECK OPTION to enforce constraints
CREATE VIEW active_customers AS
SELECT customer_id, customer_name, email, status
FROM customers
WHERE status = 'active'
WITH CHECK OPTION;

-- Materialized view (PostgreSQL)
CREATE MATERIALIZED VIEW product_sales_summary AS
SELECT
    p.product_id,
    p.product_name,
    p.category,
    COUNT(oi.order_id) AS order_count,
    SUM(oi.quantity) AS units_sold,
    SUM(oi.quantity * oi.price) AS total_revenue
FROM products p
LEFT JOIN order_items oi ON p.product_id = oi.product_id
GROUP BY p.product_id, p.product_name, p.category;

-- Refreshing a materialized view
REFRESH MATERIALIZED VIEW product_sales_summary;
```

Indexes and Constraints

Advanced indexing and constraint techniques ensure data integrity and performance.

```
-- Unique constraint
ALTER TABLE products
ADD CONSTRAINT uk_product_sku UNIQUE (sku);

-- Composite primary key
CREATE TABLE order_items (
    order_id INT,
    product_id INT,
    quantity INT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (order_id, product_id)
);

-- Foreign key with cascade actions
ALTER TABLE orders
ADD CONSTRAINT fk_orders_customer
FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
ON DELETE RESTRICT
ON UPDATE CASCADE;

-- Check constraint
ALTER TABLE products
ADD CONSTRAINT chk_price_positive CHECK (price > 0);

-- Partial/Filtered index (PostgreSQL)
CREATE INDEX idx_active_high_value_customers
ON customers (customer_name, email)
WHERE status = 'active' AND lifetime_value > 1000;

-- Functional/Expression index
CREATE INDEX idx_lower_email
ON customers (LOWER(email));

-- Covering index
CREATE INDEX idx_products_cat_price
ON products (category, price)
INCLUDE (product_name, in_stock);

-- Bitmap index (Oracle)
CREATE BITMAP INDEX idx_product_category
ON products(category);

-- Full-text search index (MySQL)
CREATE FULLTEXT INDEX idx_product_search
ON products(product_name, description);
```

JSON Support

Modern relational databases offer strong support for JSON data.

```
-- PostgreSQL JSON operations
CREATE TABLE customer_data (
    customer_id INT PRIMARY KEY,
    data JSONB
);

-- Insert JSON data
INSERT INTO customer_data (customer_id, data)
VALUES (
    1,
    '{"name": "John Smith", "email": "john@example.com", "preferences": {"theme":
"dark", "notifications": true}, "addresses": [{"type": "home", "street": "123 Main
St"}, {"type": "work", "street": "456 Market St"}]}'
);

-- Query JSON fields
SELECT
    customer_id,
    data->>'name' AS name,
    data->>'email' AS email,
    data->'preferences'->>'theme' AS theme
FROM customer_data;

-- Filter by JSON values
SELECT * FROM customer_data
WHERE data->'preferences'->>'theme' = 'dark';

-- Update JSON fields
UPDATE customer_data
SET data = jsonb_set(data, '{preferences,theme}', '"light"')
WHERE customer_id = 1;

-- Array operations
SELECT customer_id, jsonb_array_elements(data->'addresses')
FROM customer_data;
```

Pivot and Unpivot Operations

Transforming data between row and column formats.

```
-- PostgreSQL CROSSTAB (requires tablefunc extension)
CREATE EXTENSION IF NOT EXISTS tablefunc;

SELECT * FROM crosstab(
  'SELECT product_category, quarter, SUM(sales)
   FROM quarterly_sales
   GROUP BY product_category, quarter
   ORDER BY product_category, quarter',
  'SELECT DISTINCT quarter FROM quarterly_sales ORDER BY quarter'
) AS ct (
```

```
  product_category TEXT,
  "Q1" NUMERIC,
  "Q2" NUMERIC,
  "Q3" NUMERIC,
  "Q4" NUMERIC
);

-- SQL Server PIVOT
SELECT product_category, [Q1], [Q2], [Q3], [Q4]
FROM (
    SELECT product_category, quarter, sales
    FROM quarterly_sales
) AS source
PIVOT (
    SUM(sales)
    FOR quarter IN ([Q1], [Q2], [Q3], [Q4])
) AS pivot_table;

-- SQL Server UNPIVOT
SELECT product_category, quarter, sales
FROM quarterly_results
UNPIVOT (
    sales
    FOR quarter IN (Q1, Q2, Q3, Q4)
) AS unpivot_table;
```

Interview Questions

Q: How do window functions differ from aggregate functions with GROUP BY, and when would you use each?

A: Window functions and aggregate functions with GROUP BY serve different purposes and produce different result sets, despite both performing calculations across rows.

Aggregate Functions with GROUP BY:

- **Result structure**: Collapses multiple rows into a single row per group
- **Row context**: Each row in the result represents an entire group
- **Common usage**: For summarizing data and producing reports
- **Output size**: Reduces the number of rows in the result set
- **Calculations**: Apply once per group

Window Functions:

- **Result structure**: Maintains all original rows
- **Row context**: Each row maintains its identity while incorporating calculations from related rows
- **Common usage**: For data analysis, ranking, comparisons to groups/peers
- **Output size**: Preserves the original number of rows
- **Calculations**: Apply to a "window" of rows related to each row

Example scenario:

Consider a table of monthly sales by product category:

```
-- Sample data
CREATE TABLE monthly_sales (
    month DATE,
    category VARCHAR(50),
    sales_amount DECIMAL(10,2)
);
```

Using GROUP BY:

```
-- Get total sales per category
SELECT
    category,
    SUM(sales_amount) AS total_sales
FROM monthly_sales
GROUP BY category;
```

This returns one row per category, showing total sales. You lose individual month data.

Using Window Functions:

```
-- Get both individual month sales and running totals
SELECT
    month,
    category,
    sales_amount,
    SUM(sales_amount) OVER (PARTITION BY category ORDER BY month) AS running_total,
    SUM(sales_amount) OVER (PARTITION BY category) AS category_total,
    sales_amount / SUM(sales_amount) OVER (PARTITION BY category) * 100 AS
percentage_of_category
FROM monthly_sales
ORDER BY category, month;
```

This retains all original rows (month + category combinations) while adding analytical calculations.

When to use GROUP BY:

1. When you need summary reports (e.g., total sales by category)
2. When individual row details aren't needed
3. For aggregating data before presentation
4. When you need to reduce the result set size

When to use Window Functions:

1. When you need both detail and aggregated data together
2. For ranking operations (e.g., top 3 products per category)
3. For running totals or moving averages
4. To compare each row to its group/peer values (e.g., % of department total)
5. For time-based analysis with prior/next period comparisons

Hybrid Approach: You can also combine both techniques:

```
-- Get product ranking within each category and show only top 3
WITH ranked_products AS (
    SELECT
        product_name,
        category,
        sales,
        RANK() OVER (PARTITION BY category ORDER BY sales DESC) AS category_rank
    FROM product_sales
)
SELECT category, COUNT(*) AS product_count, SUM(sales) AS category_sales
FROM ranked_products
WHERE category_rank <= 3  -- Only include top 3 products per category
GROUP BY category;
```

The key difference is that GROUP BY collapses rows while window functions enhance rows with additional calculations while preserving the original data. The choice between them depends on whether you need detail-level data in your results or just summary information.

Q: Explain CTEs (Common Table Expressions) and how they can be used to simplify complex queries. Provide an example of a recursive CTE.

A: Common Table Expressions (CTEs) are named temporary result sets that exist only within the scope of a single SQL statement. They act as "virtual tables" that you can reference in your main query, making complex SQL more readable, maintainable, and easier to debug.

Key benefits of CTEs:

1. **Improved readability**: Break complex queries into logical, named components
2. **Reusability within a query**: Reference the same subquery multiple times
3. **Recursive capabilities**: Solve hierarchical or iterative problems
4. **Query modularization**: Builds complex queries in manageable steps
5. **Self-documentation**: Meaningful names explain query components

Basic CTE Syntax:

```
WITH cte_name AS (
    -- CTE query definition
    SELECT columns FROM tables WHERE conditions
)
-- Main query using the CTE
SELECT columns FROM cte_name WHERE conditions;
```

Simple CTE Example:

Instead of nesting subqueries:

```
-- Without CTE (nested subqueries)
SELECT product_name, price,
       price - (SELECT AVG(price) FROM products) AS diff_from_avg
FROM products
WHERE category IN (
    SELECT category
    FROM product_categories
    WHERE active = true
);
```

With CTEs, the query becomes more readable:

```
-- With CTEs
WITH active_categories AS (
    SELECT category
    FROM product_categories
    WHERE active = true
),
avg_price AS (
    SELECT AVG(price) AS value
    FROM products
)
SELECT
    product_name,
    price,
    price - (SELECT value FROM avg_price) AS diff_from_avg
FROM products
WHERE category IN (SELECT category FROM active_categories);
```

Recursive CTE Example:

Recursive CTEs are especially powerful for handling hierarchical data like organizational charts, bill of materials, or category trees. They consist of:

1. An anchor member (base case)
2. A recursive member that references the CTE itself
3. A UNION ALL between these parts

Here's an example of an employee hierarchy with managers:

```
-- Table structure
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100),
    manager_id INT NULL,
    title VARCHAR(100),
    department VARCHAR(100),
    hire_date DATE
);
```

```
-- Recursive CTE to find all levels of management
WITH RECURSIVE employee_hierarchy AS (
    -- Anchor member (base case): top-level employees with no manager
    SELECT
        employee_id,
        name,
        manager_id,
        title,
        0 AS level,
        CAST(name AS VARCHAR(1000)) AS hierarchy_path
    FROM employees
    WHERE manager_id IS NULL

    UNION ALL

    -- Recursive member: employees with managers
    SELECT
        e.employee_id,
        e.name,
        e.manager_id,
        e.title,
        eh.level + 1,
        CONCAT(eh.hierarchy_path, ' > ', e.name) AS hierarchy_path
    FROM employees e
    JOIN employee_hierarchy eh ON e.manager_id = eh.employee_id
)
SELECT
    employee_id,
    name,
    title,
    level,
    hierarchy_path
FROM employee_hierarchy
ORDER BY hierarchy_path;
```

This recursive CTE starts with top-level employees (those with no manager) and then repeatedly finds their subordinates, building a complete hierarchy with levels and paths.

Another Practical Example: Product Category Tree

```
-- Category table with parent-child relationships
CREATE TABLE product_categories (
    category_id INT PRIMARY KEY,
    name VARCHAR(100),
    parent_id INT NULL REFERENCES product_categories(category_id)
);

-- Finding all categories and their breadcrumb paths
WITH RECURSIVE category_tree AS (
    -- Anchor: Top-level categories
    SELECT
        category_id,
        name,
```

```
        parent_id,
        CAST(name AS VARCHAR(1000)) AS breadcrumb,
        1 AS level
    FROM product_categories
    WHERE parent_id IS NULL

    UNION ALL

    -- Recursive: Sub-categories
    SELECT
        c.category_id,
        c.name,
        c.parent_id,
        CONCAT(ct.breadcrumb, ' > ', c.name),
        ct.level + 1
    FROM product_categories c
    JOIN category_tree ct ON c.parent_id = ct.category_id
)
SELECT * FROM category_tree
ORDER BY breadcrumb;
```

Practical Applications of Recursive CTEs:

1. **Hierarchical data traversal**: Organization charts, category trees
2. **Graph traversal**: Finding paths in networks
3. **Series generation**: Creating date ranges or number sequences
4. **Bill of Materials expansion**: Product components and subcomponents
5. **Iterative calculations**: Calculating compound interest or propagating values

Limitations and Considerations:

1. Performance can degrade with deeply nested hierarchies
2. Recursive CTEs must have termination conditions to avoid infinite loops
3. Some database systems limit recursion depth
4. Include appropriate indexing on join columns for better performance

CTEs can transform complex, hard-to-maintain queries into well-structured, self-documenting code. The recursive capability adds a dimension of functionality that's difficult to achieve with standard SQL, allowing relational databases to handle hierarchical and graph-like data structures effectively.

3.3 Database Design Best Practices

Good database design is crucial for performance, scalability, and maintainability. Following established best practices helps create robust database schemas.

Database Design Process

1. **Requirements Gathering**
    - Identify entities and their attributes
    - Determine relationships between entities
    - Define business rules and constraints

    - Document performance requirements

2. **Conceptual Design**
    - Create entity-relationship (ER) diagrams
    - Define high-level entities and relationships
    - Identify primary keys and attributes

3. **Logical Design**
    - Transform ER diagrams into relations (tables)
    - Apply normalization rules
    - Define keys (primary, foreign, candidate)
    - Establish constraints

4. **Physical Design**
    - Map logical schema to database objects
    - Design indexing strategy
    - Define storage parameters
    - Plan for partitioning if needed
    - Consider denormalization where appropriate

Naming Conventions

Consistent naming conventions make databases more maintainable and self-documenting.

Table Naming:

- Use singular or plural nouns consistently (e.g., `customer` or `customers`)
- Avoid reserved SQL keywords
- Use lowercase with underscores for readability (`order_items` vs. `OrderItems`)
- Consider table name prefixes for large systems (`hr_employees`, `fin_accounts`)

Column Naming:

- Use descriptive names that indicate purpose
- Primary keys: `id` or `[table_name]_id`
- Foreign keys: `[referenced_table]_id`
- Boolean fields: Use `is_` or `has_` prefix (`is_active`, `has_subscription`)
- Date/time fields: Use suffixes like `_date`, `_time`, `_at` (`created_at`, `expiry_date`)

Constraint Naming:

- Primary keys: `pk_[table_name]`
- Foreign keys: `fk_[table_name]_[referenced_table]`
- Unique constraints: `uk_[table_name]_[column(s)]`
- Check constraints: `chk_[table_name]_[description]`
- Indexes: `idx_[table_name]_[column(s)]`

Effective Use of Data Types

Choosing appropriate data types improves performance, reduces storage, and enforces data integrity.

Numeric Types:

- Use `INT` or `INTEGER` for most IDs and counts
- Use `SMALLINT` for small ranges (-32,768 to 32,767)
- Use `BIGINT` only when necessary (large values or IDs)
- Use `DECIMAL` or `NUMERIC` for exact monetary values
- Use `FLOAT` or `REAL` only for scientific/approximate values

String Types:

- Use `CHAR(n)` only for fixed-length strings (e.g., country codes)
- Use `VARCHAR(n)` for most variable-length strings
- Set appropriate length limits (don't default to `VARCHAR(255)`)
- Use `TEXT` for long, unstructured content

Date and Time Types:

- Use `DATE` for dates without time components
- Use `TIME` for time without date components
- Use `TIMESTAMP` for date and time together
- Consider time zones: `TIMESTAMP WITH TIME ZONE` vs. `TIMESTAMP WITHOUT TIME ZONE`

Boolean Type:

- Use `BOOLEAN` for true/false values (not `CHAR(1)` or `INT`)

Special Types:

- Use `UUID` for globally unique identifiers
- Use `ENUM` or `CHECK` constraints for limited value sets
- Use `JSON` or `JSONB` for semi-structured data
- Use specialized types for geometry, arrays, etc. when appropriate

Primary Keys

Proper primary key selection is fundamental to good database design.

Primary Key Options:

1. **Natural Keys**: Existing attributes that uniquely identify entities (e.g., SSN, ISBN)
    - Pros: Meaningful to users, no additional storage
    - Cons: May change, may be complex (composite keys)
2. **Surrogate Keys**: Generated values with no business meaning (e.g., auto-increment IDs)
    - Pros: Never change, simple, consistent across tables
    - Cons: Additional storage, no inherent meaning

Best Practices:

- Prefer surrogate keys for most tables (INT or UUID)
- Keep primary keys simple (avoid complex composite keys)
- Ensure primary keys never change

- Index foreign keys referencing primary keys
- Consider using UUIDs for distributed systems or when privacy is a concern

Example:

```
-- Using auto-increment integer (traditional)
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,
    email VARCHAR(100) UNIQUE NOT NULL,
    -- other columns
);

-- Using UUID (useful for distributed systems)
CREATE TABLE customers (
    customer_id UUID DEFAULT gen_random_uuid() PRIMARY KEY,
    email VARCHAR(100) UNIQUE NOT NULL,
    -- other columns
);
```

Relationships and Foreign Keys

Properly modeling relationships ensures data integrity and supports efficient queries.

One-to-Many Relationships:

```
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100) NOT NULL
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    department_id INT,
    employee_name VARCHAR(100) NOT NULL,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);
```

Many-to-Many Relationships:

```
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100) NOT NULL
);

CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100) NOT NULL
);
```

```
CREATE TABLE enrollments (
    student_id INT,
    course_id INT,
    enrollment_date DATE NOT NULL,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);
```

One-to-One Relationships:

```
CREATE TABLE users (
    user_id INT PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL
);

CREATE TABLE user_profiles (
    user_id INT PRIMARY KEY,
    bio TEXT,
    avatar_url VARCHAR(255),
    FOREIGN KEY (user_id) REFERENCES users(user_id)
);
```

Self-Referential Relationships:

```
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100) NOT NULL,
    manager_id INT,
    FOREIGN KEY (manager_id) REFERENCES employees(employee_id)
);
```

Cascading Actions:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
        ON DELETE RESTRICT
        ON UPDATE CASCADE
);

CREATE TABLE order_items (
    order_id INT,
    product_id INT,
    quantity INT NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
```

```
        ON DELETE CASCADE,
    FOREIGN KEY (product_id) REFERENCES products(product_id)
        ON DELETE RESTRICT
);
```

Normalization Best Practices

Normalization reduces redundancy and improves data integrity, but should be applied thoughtfully.

When to Normalize:

- When data integrity is the highest priority
- For OLTP (transactional) systems
- When data is frequently updated
- When storage space is a concern
- When the data model needs to be flexible for future changes

When to Denormalize:

- For OLAP (analytical) systems
- When read performance is critical
- For reporting and data warehousing
- When complex joins impact performance
- For specific high-traffic queries

Common Normalization Mistakes:

- Over-normalization (creating too many small tables)
- Treating lookup tables and junction tables the same way
- Ignoring performance implications of deep normalization
- Not considering query patterns when normalizing

Practical Normalization Example:

Unnormalized Table:

```
Orders(order_id, customer_name, customer_email, customer_address, product_name,
product_category, product_price, quantity, order_date)
```

After Normalization:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    address TEXT NOT NULL
);

CREATE TABLE product_categories (
    category_id INT PRIMARY KEY,
```

```
    category_name VARCHAR(50) UNIQUE NOT NULL
);

CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    category_id INT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    FOREIGN KEY (category_id) REFERENCES product_categories(category_id)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

CREATE TABLE order_items (
    order_id INT,
    product_id INT,
    quantity INT NOT NULL,
    price_at_time DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Indexing Strategy

A well-planned indexing strategy balances query performance and maintenance costs.

Index Selection Guidelines:

- Index foreign key columns
- Index columns used in WHERE clauses frequently
- Index columns used in JOIN conditions
- Index columns used in ORDER BY and GROUP BY
- Consider covering indexes for high-priority queries
- Avoid over-indexing (each index adds overhead to writes)

Example Indexing Strategy:

```
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    status VARCHAR(20) NOT NULL,
    total_amount DECIMAL(10, 2) NOT NULL,
    shipping_address_id INT NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id),
    FOREIGN KEY (shipping_address_id) REFERENCES addresses(address_id)
```

```
);

-- Indexes based on access patterns
CREATE INDEX idx_orders_customer ON orders(customer_id);
CREATE INDEX idx_orders_date ON orders(order_date);
CREATE INDEX idx_orders_status ON orders(status);
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
CREATE INDEX idx_orders_status_date ON orders(status, order_date);
```

Constraints for Data Integrity

Constraints enforce business rules at the database level, ensuring data integrity.

Types of Constraints:

1. **Primary Key**: Uniquely identifies each row
2. **Foreign Key**: Ensures referential integrity
3. **Unique**: Prevents duplicate values
4. **Check**: Enforces domain rules
5. **Not Null**: Ensures required values
6. **Default**: Provides default values

Example Constraints:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    sku VARCHAR(20) UNIQUE NOT NULL,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL CHECK (price > 0),
    cost DECIMAL(10, 2) CHECK (cost > 0),
    category_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    is_active BOOLEAN DEFAULT true,
    weight DECIMAL(8, 2) CHECK (weight > 0),
    CONSTRAINT fk_product_category FOREIGN KEY (category_id)
        REFERENCES categories(category_id),
    CONSTRAINT chk_price_above_cost CHECK (price >= cost)
);
```

Temporal Data Handling

Properly modeling time-based data is crucial for many applications.

Effective Date Pattern:

```
CREATE TABLE employee_positions (
    employee_id INT NOT NULL,
    position VARCHAR(100) NOT NULL,
    department VARCHAR(100) NOT NULL,
    salary DECIMAL(10, 2) NOT NULL,
```

```
    effective_from DATE NOT NULL,
    effective_to DATE,
    PRIMARY KEY (employee_id, effective_from),
    FOREIGN KEY (employee_id) REFERENCES employees(employee_id),
    CHECK (effective_to IS NULL OR effective_from < effective_to)
);

-- Query for employee position at a specific date
SELECT position, department, salary
FROM employee_positions
WHERE employee_id = 123
  AND effective_from <= '2023-05-15'
  AND (effective_to IS NULL OR effective_to > '2023-05-15')
```

Audit Trail Pattern:

```
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    created_by VARCHAR(50) NOT NULL,
    updated_by VARCHAR(50) NOT NULL
);

-- Trigger to update the updated_at timestamp
CREATE TRIGGER update_product_timestamp
BEFORE UPDATE ON products
FOR EACH ROW
SET NEW.updated_at = CURRENT_TIMESTAMP;
```

History Table Pattern:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    status VARCHAR(20) NOT NULL,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE customer_history (
    history_id INT AUTO_INCREMENT PRIMARY KEY,
    customer_id INT NOT NULL,
    name VARCHAR(100) NOT NULL,
    email VARCHAR(100) NOT NULL,
    status VARCHAR(20) NOT NULL,
    changed_at TIMESTAMP NOT NULL,
    changed_by VARCHAR(50) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
```

```
);

-- Trigger to capture changes
DELIMITER //
CREATE TRIGGER customer_history_trigger
AFTER UPDATE ON customers
FOR EACH ROW
BEGIN
    INSERT INTO customer_history
    (customer_id, name, email, status, changed_at, changed_by)
    VALUES
    (NEW.customer_id, NEW.name, NEW.email, NEW.status, NOW(), CURRENT_USER());
END //
DELIMITER ;
```

Soft Delete Pattern

The soft delete pattern marks records as deleted rather than physically removing them.

```
CREATE TABLE articles (
    article_id INT PRIMARY KEY,
    title VARCHAR(200) NOT NULL,
    content TEXT NOT NULL,
    author_id INT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    deleted_at TIMESTAMP NULL,
    FOREIGN KEY (author_id) REFERENCES users(user_id)
);

-- Mark as deleted
UPDATE articles SET deleted_at = CURRENT_TIMESTAMP WHERE article_id = 123;

-- Query only active articles
SELECT * FROM articles WHERE deleted_at IS NULL;

-- Restore a deleted article
UPDATE articles SET deleted_at = NULL WHERE article_id = 123;
```

Interview Questions

Q: What are the key considerations when designing a database schema for a large-scale application?

A: Designing a database schema for a large-scale application requires careful planning across multiple dimensions. Here are the key considerations:

1. Data Volume and Growth Projections

- Estimate initial data volume and growth rate
- Plan for horizontal and vertical scaling
- Consider partitioning and sharding strategies

- Implement data archiving policies for historical data

2. Performance Requirements

- Identify read vs. write pattern ratios
- Determine acceptable query response times
- Plan for peak load scenarios
- Balance normalization with performance needs
- Design appropriate indexing strategy based on query patterns

3. Data Integrity and Consistency

- Implement constraints at the database level
- Define appropriate transaction boundaries
- Choose suitable isolation levels
- Consider eventual consistency vs. strong consistency trade-offs
- Plan for referential integrity across shards if applicable

4. Scalability Architecture

- Decide between vertical and horizontal scaling approaches
- Plan for read replicas and connection pooling
- Consider database federation (functional partitioning)
- Design for zero-downtime schema changes
- Implement caching strategies at appropriate levels

5. Security Considerations

- Implement row-level security where needed
- Plan for data encryption (at rest and in transit)
- Design secure authentication and authorization
- Consider data masking for sensitive information
- Plan for audit logging of critical operations

6. Schema Design Best Practices

- Apply appropriate normalization levels (typically 3NF)
- Use consistent naming conventions
- Choose appropriate data types and constraints
- Implement versioning or temporal patterns for historical tracking
- Consider soft delete patterns for recoverable data

7. Operational Concerns

- Plan for backup and recovery
- Implement monitoring and alerting
- Design for routine maintenance with minimal downtime
- Consider automated failover and high availability
- Plan for disaster recovery scenarios

8. Query Optimization

- Design schema to support efficient query patterns

- Implement materialized views for complex aggregations
- Consider denormalization for read-heavy workloads
- Create appropriate indexes based on query patterns
- Plan for query monitoring and optimization

9. Technology Selection

- Choose appropriate database type (RDBMS, NoSQL, specialized DB)
- Consider multi-model database options if appropriate
- Evaluate managed services vs. self-hosted solutions
- Consider open source vs. commercial options
- Assess team expertise with different technologies

10. Future-Proofing

- Design for schema evolution
- Implement forward and backward compatibility
- Consider API versioning strategies
- Plan for data migration paths
- Allow for business rule changes

Real-World Example:

For an e-commerce application that's expected to scale to millions of users:

```
-- Core Tables with Appropriate Types and Constraints
CREATE TABLE users (
    user_id BIGINT PRIMARY KEY,                  -- BIGINT for large user base
    email VARCHAR(255) UNIQUE NOT NULL,
    password_hash CHAR(60) NOT NULL,             -- Fixed length for BCrypt hashes
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    last_login_at TIMESTAMP NULL,
    status ENUM('active', 'inactive', 'suspended') NOT NULL DEFAULT 'active',
    tier VARCHAR(20) NOT NULL DEFAULT 'standard',
    deleted_at TIMESTAMP NULL                    -- Soft delete pattern
);

-- Partitioning for large tables
CREATE TABLE orders (
    order_id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    order_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    status VARCHAR(20) NOT NULL,
    total_amount DECIMAL(12,2) NOT NULL,
    shipping_address_id BIGINT NOT NULL,
    billing_address_id BIGINT NOT NULL,
    FOREIGN KEY (user_id) REFERENCES users(user_id),
    FOREIGN KEY (shipping_address_id) REFERENCES addresses(address_id),
    FOREIGN KEY (billing_address_id) REFERENCES addresses(address_id),
```

```
    INDEX idx_orders_user_date (user_id, order_date),
    INDEX idx_orders_status (status, order_date)
) PARTITION BY RANGE (YEAR(order_date)) (
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION p2024 VALUES LESS THAN (2025),
    PARTITION pfuture VALUES LESS THAN MAXVALUE
);

-- Implementing history tracking
CREATE TABLE product_price_history (
    history_id BIGINT AUTO_INCREMENT PRIMARY KEY,
    product_id BIGINT NOT NULL,
    price DECIMAL(12,2) NOT NULL,
    effective_from TIMESTAMP NOT NULL,
    effective_to TIMESTAMP NULL,
    created_by VARCHAR(50) NOT NULL,
    FOREIGN KEY (product_id) REFERENCES products(product_id),
    INDEX idx_product_date (product_id, effective_from)
);

-- Scalable product catalog with JSON for flexible attributes
CREATE TABLE products (
    product_id BIGINT PRIMARY KEY,
    sku VARCHAR(50) UNIQUE NOT NULL,
    name VARCHAR(255) NOT NULL,
    description TEXT NOT NULL,
    base_price DECIMAL(12,2) NOT NULL,
    category_id BIGINT NOT NULL,
    attributes JSON NOT NULL,          -- Flexible schema for varying attributes
    created_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP ON UPDATE
CURRENT_TIMESTAMP,
    FOREIGN KEY (category_id) REFERENCES categories(category_id),
    INDEX idx_products_category (category_id),
    INDEX idx_products_name ((name(50))),
    FULLTEXT INDEX idx_products_search (name, description)
);
```

This approach:

- Uses appropriate data types for scale (BIGINT for IDs)
- Implements table partitioning for large tables (orders)
- Uses soft delete pattern for data that shouldn't be permanently deleted
- Tracks history with temporal tables (product_price_history)
- Uses JSON for flexible attributes where schema varies
- Implements comprehensive indexing strategy
- Uses full-text indexing for search functionality

Ultimately, database design for large-scale applications requires balancing current needs with future growth, while carefully considering performance, scalability, and maintainability.

Q: Compare and contrast the use of surrogate keys versus natural keys in database design. When would you choose one over the other?

A: Surrogate keys and natural keys represent two fundamental approaches to primary key selection in database design, each with distinct characteristics and use cases.

Surrogate Keys vs. Natural Keys

Surrogate Keys:

- Artificial identifiers with no business meaning
- Typically auto-incremented integers, UUIDs, or system-generated values
- Examples: `customer_id = 12345`, `order_id = 'a1b2c3d4-e5f6'`

Natural Keys:

- Existing attributes (or combinations) that uniquely identify entities
- Derived from the data's inherent properties
- Examples: SSN, ISBN, email address, combination of name+DOB+address

Detailed Comparison

Aspect	Surrogate Keys	Natural Keys
Immutability	Never change	May change (email addresses, names, etc.)
Simplicity	Simple, consistent	May be complex (composite keys)
Performance	Optimized (integer lookups are fast)	May be less efficient (especially composite or string keys)
Meaning	No business meaning	Inherent business meaning
Foreign Keys	Simple references	May require multi-column foreign keys
Storage	Additional column	Uses existing columns
Implementation	Easy to implement	Requires careful attribute selection
Consistency	Uniform across database	May vary between entities
Privacy	More private (opaque IDs)	May expose sensitive information
Distributed Systems	UUIDs work well across systems	May cause conflicts across systems

Implementation Examples

Surrogate Key Example:

```
CREATE TABLE customers (
    customer_id INT AUTO_INCREMENT PRIMARY KEY,  -- Surrogate key
    email VARCHAR(255) UNIQUE NOT NULL,
    first_name VARCHAR(50) NOT NULL,
    last_name VARCHAR(50) NOT NULL,
```

```
    phone VARCHAR(20)
);

CREATE TABLE orders (
    order_id INT AUTO_INCREMENT PRIMARY KEY,      -- Surrogate key
    customer_id INT NOT NULL,
    order_date TIMESTAMP NOT NULL DEFAULT CURRENT_TIMESTAMP,
    total_amount DECIMAL(10,2) NOT NULL,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);
```

Natural Key Example:

```
CREATE TABLE countries (
    country_code CHAR(2) PRIMARY KEY,  -- Natural key (ISO country code)
    country_name VARCHAR(100) NOT NULL
);

CREATE TABLE universities (
    university_code VARCHAR(20) PRIMARY KEY,  -- Natural key (established code)
    university_name VARCHAR(255) NOT NULL,
    country_code CHAR(2) NOT NULL,
    FOREIGN KEY (country_code) REFERENCES countries(country_code)
);

CREATE TABLE courses (
    university_code VARCHAR(20),       -- Part of composite natural key
    course_code VARCHAR(20),           -- Part of composite natural key
    course_name VARCHAR(255) NOT NULL,
    department VARCHAR(100) NOT NULL,
    PRIMARY KEY (university_code, course_code),
    FOREIGN KEY (university_code) REFERENCES universities(university_code)
);
```

When to Choose Surrogate Keys

1. **When natural attributes may change:**
    - Customer email addresses, phone numbers, or names can change
    - Business identifiers might be reassigned over time
2. **For performance optimization:**
    - Integer surrogate keys are more efficient for indexing and joining
    - Shorter keys reduce index size and improve memory usage
3. **When natural keys would be composite:**
    - Avoid complex multi-column primary and foreign keys
    - Simplifies query writing and index design
4. **For distributed or merged systems:**

- UUIDs prevent key collisions when combining data from multiple sources
- Allows independent generation without coordination

5. **For privacy and security:**
   - Avoids exposing meaningful business data in URLs or logs
   - Prevents information leakage through sequential ID analysis
6. **When no suitable natural key exists:**
   - Some entities lack inherent unique identifiers
   - Natural candidates might have exceptions or special cases

When to Choose Natural Keys

1. **For reference or lookup data:**
   - Country codes, currency codes, language codes
   - Industry standard identifiers that never change
2. **When external systems require them:**
   - When integrating with systems that expect natural keys
   - When data is frequently exchanged with external parties
3. **For data that has well-established, immutable identifiers:**
   - ISBN for books
   - Chemical formulas
   - Standard product codes (UPC, EAN)
4. **When the overhead of surrogate keys isn't justified:**
   - Small, stable lookup tables
   - Join tables that map fixed identifiers
5. **For enforcing business uniqueness rules:**
   - Although you can use a surrogate key as PK and add a unique constraint

Hybrid Approach

Many real-world systems use a hybrid approach:

```
CREATE TABLE products (
    product_id INT AUTO_INCREMENT PRIMARY KEY,  -- Surrogate key for internal use
    sku VARCHAR(50) UNIQUE NOT NULL,            -- Natural key for business use
    upc VARCHAR(14) UNIQUE,                     -- Another natural identifier
    name VARCHAR(255) NOT NULL,
    price DECIMAL(10,2) NOT NULL
);
```

This approach:

- Uses surrogate keys for technical operations (joins, indexing)
- Preserves natural keys via unique constraints
- Offers the best of both worlds

Real-World Example

For a library management system:

```
-- Books table with both surrogate and natural keys
CREATE TABLE books (
    book_id INT AUTO_INCREMENT PRIMARY KEY,      -- Surrogate key
    isbn VARCHAR(13) UNIQUE NOT NULL,            -- Natural key
    title VARCHAR(255) NOT NULL,
    author_id INT NOT NULL,
    publisher_id INT NOT NULL,
    publication_year SMALLINT NOT NULL,
    FOREIGN KEY (author_id) REFERENCES authors(author_id),
    FOREIGN KEY (publisher_id) REFERENCES publishers(publisher_id)
);

-- Book copies (multiple physical copies of the same book)
CREATE TABLE book_copies (
    copy_id INT AUTO_INCREMENT PRIMARY KEY,      -- Surrogate key
    book_id INT NOT NULL,                        -- Foreign key to surrogate key
    acquisition_date DATE NOT NULL,
    status ENUM('available', 'checked_out', 'in_repair', 'lost') NOT NULL,
    FOREIGN KEY (book_id) REFERENCES books(book_id)
);

-- Book checkouts
CREATE TABLE checkouts (
    checkout_id INT AUTO_INCREMENT PRIMARY KEY,  -- Surrogate key
    copy_id INT NOT NULL,                        -- Foreign key to surrogate key
    patron_id INT NOT NULL,
    checkout_date DATE NOT NULL,
    due_date DATE NOT NULL,
    return_date DATE,
    FOREIGN KEY (copy_id) REFERENCES book_copies(copy_id),
    FOREIGN KEY (patron_id) REFERENCES patrons(patron_id)
);
```

In this example:

- ISBN is a perfect natural key for books (standardized, unique)
- But surrogate keys simplify the relationships between entities
- The system maintains both for different purposes

Conclusion

The choice between surrogate and natural keys should be driven by:

1. The nature of the data and its stability

2. Performance requirements
3. Integration needs
4. Complexity management
5. Privacy concerns

In modern database design, surrogate keys are generally preferred for most entities, with natural keys maintained through unique constraints. However, for reference data with stable, standardized identifiers, natural keys can be the more logical choice.

3.4 Performance Tuning

Database performance tuning is the process of optimizing database performance to achieve faster response times and higher throughput.

Query Optimization Techniques

Query optimization involves rewriting queries to execute more efficiently.

Filter Early:

```
-- Less efficient: Filtering after join
SELECT o.order_id, o.order_date, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date > '2023-01-01';

-- More efficient: Filtering before join
SELECT o.order_id, o.order_date, c.customer_name
FROM (SELECT order_id, order_date, customer_id
      FROM orders
      WHERE order_date > '2023-01-01') o
JOIN customers c ON o.customer_id = c.customer_id;
```

Avoid SELECT *:

```
-- Less efficient: Retrieving all columns
SELECT * FROM products WHERE category_id = 5;

-- More efficient: Retrieving only needed columns
SELECT product_id, product_name, price FROM products WHERE category_id = 5;
```

Use Appropriate JOINs:

```
-- Less efficient: Using subqueries
SELECT p.product_name, p.price,
       (SELECT category_name FROM categories WHERE category_id = p.category_id) AS
category
FROM products p
WHERE p.price > 100;
```

```
-- More efficient: Using joins
SELECT p.product_name, p.price, c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.price > 100;
```

Use EXISTS Instead of IN for Subqueries:

```
-- Less efficient with large subquery results
SELECT customer_id, customer_name
FROM customers
WHERE customer_id IN (SELECT customer_id FROM orders WHERE order_date > '2023-01-
01');

-- More efficient with EXISTS
SELECT customer_id, customer_name
FROM customers c
WHERE EXISTS (SELECT 1 FROM orders o WHERE o.customer_id = c.customer_id AND
o.order_date > '2023-01-01');
```

Avoid Functions in WHERE Clauses:

```
-- Less efficient: Function on indexed column
SELECT * FROM orders WHERE YEAR(order_date) = 2023;

-- More efficient: Direct comparison allows index usage
SELECT * FROM orders WHERE order_date >= '2023-01-01' AND order_date < '2024-01-01';
```

Use UNION ALL Instead of UNION When Duplicates Don't Matter:

```
-- Less efficient: Eliminates duplicates (requires sorting)
SELECT product_id, product_name FROM active_products
UNION
SELECT product_id, product_name FROM discontinued_products;

-- More efficient: Keeps duplicates (avoids sorting)
SELECT product_id, product_name FROM active_products
UNION ALL
SELECT product_id, product_name FROM discontinued_products;
```

Indexing Optimization

Proper indexing is crucial for query performance.

Index Types and Use Cases:

1. **Single-Column Index:**

```
CREATE INDEX idx_products_price ON products(price);
```

2. **Composite Index:**

```
-- For queries filtering on both columns
CREATE INDEX idx_orders_customer_date ON orders(customer_id, order_date);
```

3. **Covering Index:**

```
-- Index includes all columns needed by the query
CREATE INDEX idx_products_category_price_name
ON products(category_id, price) INCLUDE (product_name);
```

4. **Partial/Filtered Index:**

```
-- PostgreSQL: Index only active products
CREATE INDEX idx_active_products ON products(product_name)
WHERE status = 'active';
```

5. **Functional Index:**

```
-- Index for case-insensitive searches
CREATE INDEX idx_customers_lower_email ON customers(LOWER(email));
```

Index Analysis and Maintenance:

```
-- PostgreSQL: Find unused indexes
SELECT s.schemaname,
       s.relname AS tablename,
       s.indexrelname AS indexname,
       pg_size_pretty(pg_relation_size(s.indexrelid)) AS index_size,
       s.idx_scan AS index_scans
FROM pg_stat_user_indexes s
JOIN pg_index i ON s.indexrelid = i.indexrelid
WHERE s.idx_scan = 0 -- No scans
  AND NOT i.indisprimary -- Not a primary key
  AND NOT i.indisunique -- Not a unique key
ORDER BY pg_relation_size(s.indexrelid) DESC;

-- MySQL: Find unused indexes
SELECT
```

```
  table_name,
  index_name,
  stat_value AS index_usage
FROM
  mysql.innodb_index_stats
WHERE
  stat_name = 'rows_read'
  AND index_name != 'PRIMARY'
ORDER BY
  stat_value ASC;

-- Rebuild indexes to reduce fragmentation
-- SQL Server
ALTER INDEX idx_products_category ON products REBUILD;

-- PostgreSQL
REINDEX INDEX idx_products_category;

-- MySQL
OPTIMIZE TABLE products;
```

Query Execution Plans

Analyzing execution plans helps identify performance bottlenecks.

Getting Execution Plans:

```
-- PostgreSQL
EXPLAIN ANALYZE
SELECT p.product_name, c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.price > 100
ORDER BY p.price DESC;

-- MySQL
EXPLAIN
SELECT p.product_name, c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.price > 100
ORDER BY p.price DESC;

-- SQL Server
SET STATISTICS IO, TIME ON;
GO
SELECT p.product_name, c.category_name
FROM products p
JOIN categories c ON p.category_id = c.category_id
WHERE p.price > 100
ORDER BY p.price DESC;
GO
```

Common Execution Plan Issues:

1. **Table Scans**: When no suitable index is used
   - Solution: Add appropriate indexes
2. **Inefficient Joins**: Nested loop joins for large tables
   - Solution: Ensure join columns are indexed
3. **Sorting Operations**: External sorts requiring disk I/O
   - Solution: Add indexes that support ORDER BY
4. **Temporary Tables**: Created for complex operations
   - Solution: Simplify queries or add covering indexes

Database Configuration Tuning

Optimizing database server parameters can significantly improve performance.

Memory Configuration:

```
# PostgreSQL memory settings
shared_buffers = 2GB                    # 25% of RAM for dedicated servers
work_mem = 64MB                         # Per-sort operation memory
maintenance_work_mem = 256MB            # For maintenance operations
effective_cache_size = 6GB              # Estimate of OS cache (75% of RAM)

# MySQL/MariaDB memory settings
innodb_buffer_pool_size = 6G            # 70-80% of RAM for dedicated servers
innodb_buffer_pool_instances = 8        # Multiple instances for concurrency
innodb_log_buffer_size = 32M            # Transaction log buffer
key_buffer_size = 256M                  # For MyISAM tables
query_cache_size = 0                    # Disable query cache for high throughput
```

Concurrency Settings:

```
# PostgreSQL concurrency
max_connections = 100
max_worker_processes = 8
max_parallel_workers_per_gather = 4

# MySQL/MariaDB concurrency
max_connections = 500
innodb_thread_concurrency = 16          # 0 = unlimited, or 2x CPU cores
thread_cache_size = 32
```

I/O Settings:

```
# PostgreSQL I/O
wal_buffers = 16MB
checkpoint_timeout = 15min
checkpoint_completion_target = 0.9
random_page_cost = 1.1                # For SSD storage (4.0 is default for HDD)

# MySQL/MariaDB I/O
innodb_flush_log_at_trx_commit = 2    # Slightly less durability, better performance
innodb_flush_method = O_DIRECT        # Bypass OS cache
innodb_io_capacity = 2000             # Higher for SSD
innodb_write_io_threads = 16
innodb_read_io_threads = 16
```

Partitioning for Performance

Partitioning large tables can improve query performance and manageability.

Range Partitioning:

```
-- MySQL: Partition orders by date range
CREATE TABLE orders (
    order_id INT NOT NULL,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (order_id, order_date)
)
PARTITION BY RANGE (YEAR(order_date)) (
    PARTITION p2021 VALUES LESS THAN (2022),
    PARTITION p2022 VALUES LESS THAN (2023),
    PARTITION p2023 VALUES LESS THAN (2024),
    PARTITION pfuture VALUES LESS THAN MAXVALUE
);

-- PostgreSQL: Partition orders by date range
CREATE TABLE orders (
    order_id INT NOT NULL,
    customer_id INT NOT NULL,
    order_date DATE NOT NULL,
    total_amount DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (order_id, order_date)
) PARTITION BY RANGE (order_date);

CREATE TABLE orders_2021 PARTITION OF orders
    FOR VALUES FROM ('2021-01-01') TO ('2022-01-01');

CREATE TABLE orders_2022 PARTITION OF orders
    FOR VALUES FROM ('2022-01-01') TO ('2023-01-01');

CREATE TABLE orders_2023 PARTITION OF orders
    FOR VALUES FROM ('2023-01-01') TO ('2024-01-01');
```

```
CREATE TABLE orders_future PARTITION OF orders
    FOR VALUES FROM ('2024-01-01') TO (MAXVALUE);
```

List Partitioning:

```
-- MySQL: Partition customers by region
CREATE TABLE customers (
    customer_id INT NOT NULL,
    customer_name VARCHAR(100) NOT NULL,
    region VARCHAR(20) NOT NULL,
    PRIMARY KEY (customer_id, region)
)
PARTITION BY LIST (region) (
    PARTITION p_east VALUES IN ('East', 'Northeast', 'Southeast'),
    PARTITION p_west VALUES IN ('West', 'Northwest', 'Southwest'),
    PARTITION p_central VALUES IN ('Central', 'North', 'South'),
    PARTITION p_other VALUES IN ('Other', 'Unknown')
);
```

Hash Partitioning:

```
-- MySQL: Hash partition for even distribution
CREATE TABLE order_items (
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    price DECIMAL(10, 2) NOT NULL,
    PRIMARY KEY (order_id, product_id)
)
PARTITION BY HASH (order_id)
PARTITIONS 8;
```

Caching Strategies

Implementing caching at different levels can significantly improve performance.

Database Query Cache:

```
# PostgreSQL: Shared buffers are primary cache
shared_buffers = 2GB

# MySQL: Query cache (in MySQL 5.7 and earlier)
query_cache_type = 1
query_cache_size = 128M
query_cache_limit = 2M
```

Application-Level Cache:

```
# Python example with Redis cache
import redis
import json

redis_client = redis.Redis(host='localhost', port=6379, db=0)

def get_product(product_id):
    # Try to get from cache
    cache_key = f"product:{product_id}"
    cached_product = redis_client.get(cache_key)

    if cached_product:
        return json.loads(cached_product)

    # Not in cache, get from database
    query = "SELECT product_id, name, price, category_id FROM products WHERE
product_id = %s"
    product = db.execute_query(query, (product_id,))

    # Store in cache (with 1-hour expiration)
    redis_client.setex(cache_key, 3600, json.dumps(product))

    return product
```

Result Set Caching:

```
-- PostgreSQL: Materialized view for expensive queries
CREATE MATERIALIZED VIEW sales_by_month AS
SELECT
    DATE_TRUNC('month', order_date) AS month,
    SUM(total_amount) AS total_sales,
    COUNT(DISTINCT customer_id) AS customer_count
FROM orders
GROUP BY DATE_TRUNC('month', order_date)
ORDER BY month;

-- Refresh when needed
REFRESH MATERIALIZED VIEW sales_by_month;
```

Connection Pooling

Connection pooling reduces the overhead of establishing database connections.

PostgreSQL Connection Pooling with PgBouncer:

```
# pgbouncer.ini
[databases]
mydb = host=localhost port=5432 dbname=mydb

[pgbouncer]
```

```
listen_addr = *
listen_port = 6432
auth_type = md5
auth_file = /etc/pgbouncer/userlist.txt
pool_mode = transaction
max_client_conn = 1000
default_pool_size = 100
```

Connection Pool in Application Code:

```
// Java connection pooling with HikariCP
HikariConfig config = new HikariConfig();
config.setJdbcUrl("jdbc:postgresql://localhost:5432/mydb");
config.setUsername("user");
config.setPassword("password");
config.setMaximumPoolSize(10);
config.setMinimumIdle(5);
config.setIdleTimeout(300000);
config.setConnectionTimeout(10000);

HikariDataSource dataSource = new HikariDataSource(config);

// Get a connection from the pool
try (Connection conn = dataSource.getConnection()) {
    // Use the connection
}
```

Interview Questions

Q: How would you approach optimizing a slow-performing database query in a production environment?

A: Optimizing a slow query in a production environment requires a methodical approach that balances performance improvement with system stability. Here's my comprehensive approach:

1. Identify and Understand the Problem

Gather Performance Metrics:

```
-- PostgreSQL: Find slow queries
SELECT
    query,
    calls,
    total_exec_time,
    mean_exec_time,
    rows,
    100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS
hit_percent
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;
```

```
-- MySQL: Using the slow query log
SELECT
    query_time,
    rows_sent,
    rows_examined,
    sql_text
FROM mysql.slow_log
WHERE start_time > DATE_SUB(NOW(), INTERVAL 1 DAY)
ORDER BY query_time DESC
LIMIT 10;
```

Analyze the Query:

- Obtain the exact query text and typical parameters
- Understand the business purpose of the query
- Identify tables and relationships involved
- Note frequency of execution and typical result size

2. Analyze the Execution Plan

```
-- PostgreSQL
EXPLAIN ANALYZE SELECT * FROM orders
JOIN customers ON orders.customer_id = customers.customer_id
WHERE order_date BETWEEN '2023-01-01' AND '2023-01-31'
ORDER BY total_amount DESC;

-- MySQL
EXPLAIN FORMAT=JSON SELECT * FROM orders
JOIN customers ON orders.customer_id = customers.customer_id
WHERE order_date BETWEEN '2023-01-01' AND '2023-01-31'
ORDER BY total_amount DESC;
```

Look for Common Issues:

- Table scans (missing indexes)
- Inefficient join algorithms
- Sorting operations
- Temporary tables or filesorts
- High row estimates vs. actual rows processed
- Inefficient index usage

3. Check Existing Indexes

```
-- PostgreSQL: View indexes on a table
SELECT
    indexname,
    indexdef
FROM pg_indexes
WHERE tablename = 'orders';
```

```
-- PostgreSQL: Check index usage
SELECT
    relname AS table_name,
    indexrelname AS index_name,
    idx_scan AS index_scans,
    idx_tup_read AS tuples_read,
    idx_tup_fetch AS tuples_fetched
FROM pg_stat_user_indexes
JOIN pg_index USING (indexrelid)
WHERE relname = 'orders';

-- MySQL: View indexes
SHOW INDEX FROM orders;
```

4. Develop an Optimization Strategy

Based on the analysis, create a prioritized list of potential improvements:

Index Optimizations:

```
-- Add missing index
CREATE INDEX idx_orders_date ON orders(order_date);

-- Create composite index for multiple conditions
CREATE INDEX idx_orders_date_amount ON orders(order_date, total_amount);

-- Create covering index
CREATE INDEX idx_orders_customer_date_amount
ON orders(customer_id, order_date, total_amount)
INCLUDE (status);
```

Query Rewriting:

```
-- Original query
SELECT o.*, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE EXTRACT(YEAR FROM o.order_date) = 2023;

-- Rewritten query
SELECT o.*, c.customer_name
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
WHERE o.order_date >= '2023-01-01' AND o.order_date < '2024-01-01';
```

Schema Optimizations:

```
-- Add materialized view for common aggregations
CREATE MATERIALIZED VIEW monthly_order_summary AS
```

```
SELECT
    DATE_TRUNC('month', order_date) AS month,
    COUNT(*) AS order_count,
    SUM(total_amount) AS total_sales
FROM orders
GROUP BY DATE_TRUNC('month', order_date);

-- Create index on materialized view
CREATE INDEX idx_monthly_summary_month ON monthly_order_summary(month);
```

5. Test in Non-Production Environment

Before applying changes to production:

1. **Create a Test Environment:**
    - Use a copy of production data (or representative subset)
    - Replicate production configuration
2. **Benchmark Existing Performance:**

```
-- PostgreSQL
EXPLAIN ANALYZE SELECT /* benchmark original */ ...
```

3. **Apply Optimizations and Measure Improvement:**

```
-- Apply index
CREATE INDEX idx_test_optimization ON orders(order_date);

-- Benchmark after change
EXPLAIN ANALYZE SELECT /* benchmark optimized */ ...
```

4. **Assess Side Effects:**
    - Check impact on write operations
    - Verify other queries aren't negatively affected
    - Measure index size and storage impact

6. Implement in Production with Safeguards

Preparation:

- Schedule during low-traffic period if possible
- Prepare rollback plan
- Set up monitoring for query performance
- Alert team members about the change

Implementation Approaches:

For Index Creation:

```
-- PostgreSQL: Create index concurrently (minimal locking)
CREATE INDEX CONCURRENTLY idx_orders_date ON orders(order_date);

-- MySQL: Create index with algorithm option
ALTER TABLE orders ADD INDEX idx_orders_date (order_date),
ALGORITHM=INPLACE, LOCK=NONE;
```

For Query Changes:

- Implement in application code
- Update stored procedures
- Use query hints if needed

7. Monitor After Implementation

Short-term Monitoring:

- Watch for errors or exceptions
- Verify query performance improvement
- Check system resource utilization (CPU, memory, I/O)

Long-term Monitoring:

```
-- Create a baseline of query performance
CREATE TABLE query_performance_baseline (
    query_id TEXT,
    execution_time NUMERIC,
    rows_processed INTEGER,
    captured_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Track periodic performance
INSERT INTO query_performance_baseline (query_id, execution_time, rows_processed)
SELECT 'order_report_query', total_exec_time, rows
FROM pg_stat_statements
WHERE query LIKE '%/* order_report_query */%'
LIMIT 1;
```

8. Document Changes and Lessons Learned

Create documentation that includes:

- Original problem and symptoms
- Root cause analysis
- Changes made and rationale
- Performance improvement metrics
- Best practices for similar queries

Real-World Example

Problem: A slow dashboard query showing customer order summaries:

```
-- Original slow query
SELECT c.customer_name,
       COUNT(o.order_id) AS order_count,
       SUM(o.total_amount) AS total_spent,
       MAX(o.order_date) AS last_order_date
FROM customers c
LEFT JOIN orders o ON c.customer_id = o.customer_id
WHERE o.order_date >= '2023-01-01'
GROUP BY c.customer_name
ORDER BY total_spent DESC
LIMIT 100;
```

Analysis:

- EXPLAIN shows full table scan on orders (10M+ rows)
- No index on order_date
- Sorting large result set for ORDER BY

Optimization Steps:

1. Add index for the date filter:

```
CREATE INDEX CONCURRENTLY idx_orders_date ON orders(order_date);
```

2. Add composite index to support both filter and sort:

```
CREATE INDEX CONCURRENTLY idx_orders_date_amount
ON orders(order_date, total_amount DESC);
```

3. Rewrite query to filter early and use more efficient GROUP BY:

```
-- Optimized query
SELECT c.customer_name,
       COUNT(o.order_id) AS order_count,
       SUM(o.total_amount) AS total_spent,
       MAX(o.order_date) AS last_order_date
FROM customers c
LEFT JOIN (
    SELECT customer_id, order_id, total_amount, order_date
    FROM orders
    WHERE order_date >= '2023-01-01'
) o ON c.customer_id = o.customer_id
GROUP BY c.customer_id, c.customer_name
```

```
ORDER BY total_spent DESC NULLS LAST
LIMIT 100;
```

4. For more dramatic improvement, create a summary table with daily refresh:

```
CREATE TABLE customer_order_summary (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    order_count INT,
    total_spent DECIMAL(12,2),
    last_order_date DATE
);

-- Refresh procedure
CREATE PROCEDURE refresh_customer_summary()
LANGUAGE SQL
AS $$
    TRUNCATE TABLE customer_order_summary;

    INSERT INTO customer_order_summary
    SELECT c.customer_id,
           c.customer_name,
           COUNT(o.order_id),
           COALESCE(SUM(o.total_amount), 0),
           MAX(o.order_date)
    FROM customers c
    LEFT JOIN orders o ON c.customer_id = o.customer_id
    GROUP BY c.customer_id, c.customer_name;

    CREATE INDEX idx_summary_spent ON customer_order_summary(total_spent DESC);
$$;

-- Schedule refresh
SELECT cron.schedule('0 1 * * *', 'CALL refresh_customer_summary()');
```

Results:

- Original query: 15 seconds execution time
- After indexes: 2.5 seconds execution time
- With summary table: 0.05 seconds execution time

This methodical approach ensures that we not only improve performance but do so safely and with proper validation, minimizing risks while maximizing benefits.

Q: What are the most important metrics to monitor for database performance, and how would you set up alerting for performance issues?

A: Monitoring and alerting for database performance requires tracking key metrics across multiple dimensions. Here's a comprehensive approach to database performance monitoring:

Key Database Performance Metrics

1. Query Performance Metrics

Response Time:

- Average query execution time
- 95th/99th percentile query times
- Slow query count and frequency

Throughput:

- Queries per second
- Transactions per second
- Reads vs. writes ratio

Resource Utilization per Query:

- Logical reads (buffer gets)
- Physical reads (disk reads)
- Rows processed

Implementation:

```
-- PostgreSQL: Create extension if not exists
CREATE EXTENSION IF NOT EXISTS pg_stat_statements;

-- PostgreSQL: Monitor top queries by execution time
SELECT
    substring(query, 1, 100) AS query_sample,
    calls,
    total_exec_time / 1000 AS total_exec_time_sec,
    total_exec_time / calls / 1000 AS avg_exec_time_sec,
    rows / calls AS avg_rows,
    100.0 * shared_blks_hit / nullif(shared_blks_hit + shared_blks_read, 0) AS
hit_percent
FROM pg_stat_statements
ORDER BY total_exec_time DESC
LIMIT 10;

-- MySQL: Enable performance schema
SET GLOBAL performance_schema = ON;

-- MySQL: Monitor query performance
SELECT
    DIGEST_TEXT AS query_sample,
    COUNT_STAR AS execution_count,
    SUM_TIMER_WAIT / 1000000000 AS total_exec_time_sec,
    AVG_TIMER_WAIT / 1000000000 AS avg_exec_time_sec,
    SUM_ROWS_EXAMINED / COUNT_STAR AS avg_rows_examined,
    SUM_ROWS_SENT / COUNT_STAR AS avg_rows_returned
FROM performance_schema.events_statements_summary_by_digest
ORDER BY sum_timer_wait DESC
LIMIT 10;
```

2. Server Resource Metrics

CPU Usage:

- System CPU utilization
- Database process CPU utilization
- CPU queue length
- Context switches

Memory Usage:

- Buffer/cache hit ratio
- Memory used by database processes
- Swap usage
- Page faults

Disk I/O:

- IOPS (reads and writes per second)
- Throughput (MB/s)
- Latency (ms per operation)
- Queue length
- Read vs. write ratio

Network:

- Network throughput (MB/s)
- Connection count
- Network latency
- Packet errors/drops

Implementation:

```
# Linux system-level monitoring
iostat -xdm 5  # Disk I/O stats every 5 seconds
vmstat 5       # Memory and CPU stats
sar -n DEV 5   # Network statistics

# PostgreSQL memory usage
SELECT
    pg_size_pretty(pg_database_size(current_database())) AS db_size,
    pg_size_pretty(sum(pg_relation_size(C.oid))::bigint) AS tables_size,
    pg_size_pretty(sum(pg_relation_size(I.oid))::bigint) AS indexes_size
FROM pg_class C
LEFT JOIN pg_index X ON C.oid = X.indrelid
LEFT JOIN pg_class I ON I.oid = X.indexrelid
WHERE C.relkind = 'r' AND C.relnamespace NOT IN
    (SELECT oid FROM pg_namespace WHERE nspname IN ('pg_catalog',
'information_schema'))
GROUP BY 1;

# MySQL resource usage
SHOW GLOBAL STATUS LIKE 'Innodb_buffer_pool%';
```

```
SHOW GLOBAL STATUS LIKE 'Threads_%';
SHOW GLOBAL STATUS LIKE 'Handler_%';
```

3. Database-Specific Metrics

Connection Metrics:

- Current connections
- Connection utilization (%)
- Failed connections
- Connection wait time
- Idle connections

Cache Efficiency:

- Buffer cache hit ratio
- Shared pool hit ratio (Oracle)
- InnoDB buffer pool hit ratio (MySQL)
- Plan cache hit ratio (SQL Server)

Lock Metrics:

- Lock wait time
- Lock timeouts
- Deadlocks
- Blocked transactions

Transaction Metrics:

- Commits per second
- Rollbacks per second
- Transaction duration
- Active transactions

Implementation:

```
-- PostgreSQL connection stats
SELECT
    max_conn.setting AS max_connections,
    current_conn.count AS current_connections,
    (current_conn.count::float / max_conn.setting::float) * 100 AS
connection_utilization
FROM
    (SELECT setting FROM pg_settings WHERE name = 'max_connections') max_conn,
    (SELECT count(*) FROM pg_stat_activity) current_conn;

-- PostgreSQL lock monitoring
SELECT
    blocked_locks.pid AS blocked_pid,
    blocked_activity.usename AS blocked_user,
    blocking_locks.pid AS blocking_pid,
    blocking_activity.usename AS blocking_user,
```

```
    blocked_activity.query AS blocked_statement,
    blocking_activity.query AS blocking_statement
FROM pg_catalog.pg_locks blocked_locks
JOIN pg_catalog.pg_stat_activity blocked_activity ON blocked_activity.pid =
blocked_locks.pid
JOIN pg_catalog.pg_locks blocking_locks ON blocking_locks.locktype =
blocked_locks.locktype
    AND blocking_locks.database IS NOT DISTINCT FROM blocked_locks.database
    AND blocking_locks.relation IS NOT DISTINCT FROM blocked_locks.relation
    AND blocking_locks.page IS NOT DISTINCT FROM blocked_locks.page
    AND blocking_locks.tuple IS NOT DISTINCT FROM blocked_locks.tuple
    AND blocking_locks.virtualxid IS NOT DISTINCT FROM blocked_locks.virtualxid
    AND blocking_locks.transactionid IS NOT DISTINCT FROM
blocked_locks.transactionid
    AND blocking_locks.classid IS NOT DISTINCT FROM blocked_locks.classid
    AND blocking_locks.objid IS NOT DISTINCT FROM blocked_locks.objid
    AND blocking_locks.objsubid IS NOT DISTINCT FROM blocked_locks.objsubid
    AND blocking_locks.pid != blocked_locks.pid
JOIN pg_catalog.pg_stat_activity blocking_activity ON blocking_activity.pid =
blocking_locks.pid
WHERE NOT blocked_locks.granted;

-- MySQL InnoDB buffer pool stats
SELECT
    ROUND(100 * (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')),
2) AS buffer_pool_hit_ratio;
```

4. Storage Metrics

Tablespace Usage:

- Total space used
- Free space available
- Growth rate
- Auto-extend events

Table Growth:

- Size of largest tables
- Fastest growing tables
- Index size vs. table size

WAL/Binlog Metrics:

- Write-ahead log generation rate
- Checkpoint frequency
- Replication lag
- Binlog size

Implementation:

```
-- PostgreSQL: Table sizes
SELECT
    table_name,
    pg_size_pretty(pg_total_relation_size(quote_ident(table_name))) AS total_size,
    pg_size_pretty(pg_relation_size(quote_ident(table_name))) AS table_size,
    pg_size_pretty(pg_total_relation_size(quote_ident(table_name)) -
                   pg_relation_size(quote_ident(table_name))) AS index_size
FROM information_schema.tables
WHERE table_schema = 'public'
ORDER BY pg_total_relation_size(quote_ident(table_name)) DESC
LIMIT 10;

-- MySQL: Table sizes
SELECT
    table_name,
    ROUND(data_length/1024/1024, 2) AS data_size_mb,
    ROUND(index_length/1024/1024, 2) AS index_size_mb,
    ROUND((data_length + index_length)/1024/1024, 2) AS total_size_mb
FROM information_schema.tables
WHERE table_schema = 'your_database'
ORDER BY (data_length + index_length) DESC
LIMIT 10;
```

Setting Up a Monitoring and Alerting System

1. Multi-Tiered Alert Levels

Info (Blue):

- Metrics approaching warning thresholds
- Notable changes in trends
- Automated reports on growth rates

Warning (Yellow):

- Resource utilization over 70%
- Response time degradation (50% increase)
- Slow query count increase
- Connection pool utilization over 70%

Critical (Red):

- Resource utilization over 90%
- Response time degradation (100%+ increase)
- Deadlocks or prolonged blocking
- Replication lag exceeding thresholds
- Free space below critical threshold

2. Implementing Alerts Using Prometheus and Alertmanager

Prometheus Configuration:

```
# prometheus.yml
scrape_configs:
  - job_name: 'postgresql'
    static_configs:
      - targets: ['postgres_exporter:9187']
  - job_name: 'node'
    static_configs:
      - targets: ['node_exporter:9100']
```

Alert Rules:

```
# alerts.yml
groups:
  - name: database_alerts
    rules:
      # CPU Usage Alert
      - alert: DatabaseHighCPUUsage
        expr: rate(process_cpu_seconds_total{job="postgresql"}[5m]) * 100 > 80
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: 'High CPU usage on database server'
          description: 'Database server has {{ $value }}% CPU usage for more than 5
minutes'

      # Connection Saturation
      - alert: DatabaseConnectionSaturation
        expr: pg_stat_activity_count{datname!~"template.*|postgres"} /
pg_settings_max_connections * 100 > 80
        for: 5m
        labels:
          severity: warning
        annotations:
          summary: 'Database connection pool nearly full'
          description: 'Database {{ $labels.datname }} is using {{ $value }}% of
available connections'

      # Slow Queries Increase
      - alert: SlowQueriesIncreasing
        expr: rate(pg_stat_statements_total_time_seconds[5m]) > 10
        for: 10m
        labels:
          severity: warning
        annotations:
          summary: 'Increase in slow queries detected'
          description: 'Database is experiencing slow query execution'

      # Replication Lag
      - alert: ReplicationLagHigh
```

```
        expr: pg_stat_replication_lag_bytes > 50000000
        for: 5m
        labels:
          severity: critical
        annotations:
          summary: 'High replication lag'
          description: 'Replication lag is {{ $value | humanizeBytes }} and
increasing'
```

3. Dashboard and Visualization

Grafana Dashboard Panels:

1. **Overview Dashboard:**
   - Server resource utilization (CPU, Memory, Disk, Network)
   - Query throughput and response time
   - Active connections
   - Error rates
2. **Query Performance Dashboard:**
   - Top 10 queries by execution time
   - Slow query count over time
   - Query cache hit ratio
   - Plan changes
3. **Storage Dashboard:**
   - Database size growth
   - Tablespace utilization
   - Top 10 tables by size
   - Index size vs. table size
4. **Availability Dashboard:**
   - Uptime
   - Replication status and lag
   - Backup status
   - Recovery metrics

4. Anomaly Detection

Statistical Anomaly Detection:

```
-- Create baseline table
CREATE TABLE query_performance_baseline (
    query_hash TEXT,
    avg_execution_time NUMERIC,
    std_dev NUMERIC,
    last_updated TIMESTAMP
```

```
);

-- Populate and update baseline (scheduled job)
INSERT INTO query_performance_baseline
SELECT
    md5(query) AS query_hash,
    AVG(mean_exec_time) AS avg_execution_time,
    STDDEV(mean_exec_time) AS std_dev,
    NOW() AS last_updated
FROM pg_stat_statements
GROUP BY md5(query)
ON CONFLICT (query_hash) DO UPDATE
SET
    avg_execution_time = (query_performance_baseline.avg_execution_time * 0.7) +
(EXCLUDED.avg_execution_time * 0.3),
    std_dev = (query_performance_baseline.std_dev * 0.7) + (EXCLUDED.std_dev * 0.3),
    last_updated = NOW();

-- Detect anomalies
SELECT
    s.query,
    s.mean_exec_time AS current_avg_time,
    b.avg_execution_time AS baseline_avg_time,
    (s.mean_exec_time - b.avg_execution_time) / NULLIF(b.std_dev, 0) AS z_score
FROM pg_stat_statements s
JOIN query_performance_baseline b ON md5(s.query) = b.query_hash
WHERE (s.mean_exec_time - b.avg_execution_time) / NULLIF(b.std_dev, 0) > 3
  AND s.calls > 10;
```

5. Automated Response Actions

Read-Only Mode Protection:

```
-- PostgreSQL: Function to activate read-only mode when resources critical
CREATE OR REPLACE FUNCTION activate_read_only_mode()
RETURNS void AS $$
BEGIN
    -- Log the event
    INSERT INTO admin_events (event_type, description, created_at)
    VALUES ('READ_ONLY_MODE_ACTIVATED', 'Activated due to resource constraints',
NOW());

    -- Make database read-only
    ALTER SYSTEM SET default_transaction_read_only = on;
    SELECT pg_reload_conf();
END;
$$ LANGUAGE plpgsql;
```

Auto-Analyze for Statistics:

```
-- PostgreSQL: Function to identify and analyze tables with stale statistics
CREATE OR REPLACE FUNCTION auto_analyze_stale_tables()
RETURNS void AS $$
DECLARE
    stale_table record;
BEGIN
    FOR stale_table IN
        SELECT schemaname, relname
        FROM pg_stat_user_tables
        WHERE n_mod_since_analyze > 1000
        ORDER BY n_mod_since_analyze DESC
        LIMIT 5
    LOOP
        EXECUTE 'ANALYZE ' || quote_ident(stale_table.schemaname) || '.' ||
quote_ident(stale_table.relname);
        RAISE NOTICE 'Analyzed table: %', stale_table.relname;
    END LOOP;
END;
$$ LANGUAGE plpgsql;
```

6. Long-Term Trend Analysis

Storing Historical Metrics:

```
-- Create metrics history table
CREATE TABLE database_metrics_history (
    captured_at TIMESTAMP NOT NULL,
    metric_name VARCHAR(100) NOT NULL,
    metric_value NUMERIC NOT NULL,
    tags JSONB NOT NULL DEFAULT '{}'
);

-- Create time-based index
CREATE INDEX idx_metrics_history_time ON database_metrics_history(captured_at);
CREATE INDEX idx_metrics_history_name ON database_metrics_history(metric_name);

-- Implement partitioning for efficient storage
CREATE TABLE database_metrics_history_y2023m01 PARTITION OF database_metrics_history
    FOR VALUES FROM ('2023-01-01') TO ('2023-02-01');

-- Capture periodic snapshots
INSERT INTO database_metrics_history (captured_at, metric_name, metric_value, tags)
SELECT
    current_timestamp,
    'buffer_hit_ratio',
    (SELECT blks_hit::numeric / (blks_hit + blks_read) * 100
     FROM pg_stat_database
     WHERE datname = current_database()),
    '{"database": "' || current_database() || '"}';
```

Implementation Approach

1. **Start with Core Metrics:**
    - Server resources (CPU, memory, disk, network)
    - Query performance metrics
    - Connection utilization
    - Error rates and availability
2. **Establish Baselines:**
    - Collect at least 2 weeks of data
    - Identify daily and weekly patterns
    - Document peak and average values
3. **Set Initial Thresholds:**
    - Warning: 2 standard deviations from baseline
    - Critical: 3 standard deviations from baseline
    - Adjust based on business impact
4. **Implement Alerting Channels:**
    - Email for non-urgent issues
    - SMS/push notifications for critical issues
    - Ticketing system integration
    - Team chat integration (Slack/Teams)
5. **Reduce Alert Fatigue:**
    - Group related alerts
    - Implement alert suppression during maintenance
    - Use exponential backoff for repeated alerts
    - Create different severity levels
6. **Document Response Procedures:**
    - Create runbooks for common alerts
    - Define escalation paths
    - Set SLAs for different alert severities

Real-World Example: E-commerce Database Monitoring

For an e-commerce platform with peak holiday traffic:

Critical Metrics:

1. Transaction throughput during checkout process
2. Product catalog query response time
3. Inventory update lock contention
4. Order processing queue length
5. Payment processing success rate

Custom Alerts:

```
# Checkout process latency alert
- alert: CheckoutLatencyHigh
  expr: histogram_quantile(0.95, rate(checkout_query_duration_seconds_bucket[5m])) >
0.5
  for: 2m
  labels:
    severity: critical
    team: ecommerce
  annotations:
    summary: 'Checkout queries are slow'
    description: '95th percentile checkout query latency is {{ $value }}s
(threshold: 0.5s)'

# Inventory contention alert
- alert: InventoryLockContention
  expr: rate(postgres_locks_count{relation="inventory"}[5m]) > 10
  for: 2m
  labels:
    severity: warning
    team: inventory
  annotations:
    summary: 'High lock contention on inventory table'
    description: 'Inventory table experiencing {{ $value }} locks per second'

# Product search performance
- alert: ProductSearchDegradation
  expr: rate(product_search_latency_sum[5m]) /
rate(product_search_latency_count[5m]) > 0.2
  for: 5m
  labels:
    severity: warning
    team: search
  annotations:
    summary: 'Product search performance degraded'
    description: 'Average product search taking {{ $value }}s to complete'
```

Holiday Season Adjustments:

1. Lower thresholds temporarily during peak hours
2. Increase alert specificity to reduce noise
3. Add capacity-focused predictive alerts
4. Implement auto-scaling triggers
5. Enable more aggressive query caching

This comprehensive monitoring approach provides visibility into database performance from multiple angles, enabling both proactive optimization and rapid response to issues. By combining real-time alerting with trend analysis, you can ensure database reliability while continuously improving performance.

3.5 Popular RDBMS Systems

There are several popular relational database management systems (RDBMS), each with unique features, strengths, and use cases.

MySQL

MySQL is one of the most widely used open-source relational database systems, known for its performance, reliability, and ease of use.

Key Features:

- Multiple storage engines (InnoDB, MyISAM, Memory, etc.)
- Replication (master-slave, master-master)
- Partitioning
- Stored procedures and triggers
- Window functions (8.0+)
- JSON support
- Common Table Expressions (8.0+)

Architecture:

- Client-server architecture
- Thread-based model (one thread per connection)
- Storage engine architecture (pluggable storage engines)
- Query cache (deprecated in 8.0)
- Buffer pool for caching data and indexes

Strengths:

- Easy to set up and use
- Good performance for read-heavy workloads
- Widely supported by applications and tools
- Active community and extensive documentation
- Good performance-to-cost ratio

Limitations:

- Less advanced features compared to PostgreSQL
- Limited support for complex queries until 8.0
- Subquery performance can be suboptimal
- Not fully ACID compliant with some storage engines

Best Use Cases:

- Web applications
- Online transaction processing (OLTP)
- Content management systems
- E-commerce platforms
- Small to medium-sized businesses

Example Configuration (my.cnf):

```
[mysqld]
# Basic Settings
port = 3306
socket = /var/run/mysqld/mysqld.sock
```

```
user = mysql
pid-file = /var/run/mysqld/mysqld.pid
datadir = /var/lib/mysql

# Buffer Pool Settings
innodb_buffer_pool_size = 4G            # 70-80% of RAM for dedicated DB server
innodb_buffer_pool_instances = 4        # Multiple instances for concurrency

# InnoDB Settings
innodb_file_per_table = 1               # Separate file for each table
innodb_flush_log_at_trx_commit = 1      # Full ACID (0 or 2 for better performance)
innodb_log_file_size = 512M             # Larger log files for write-heavy workloads
innodb_log_buffer_size = 16M            # Log buffer size

# Connection Settings
max_connections = 500                   # Maximum concurrent connections
thread_cache_size = 128                 # Thread cache
max_allowed_packet = 64M                # Maximum packet size

# Query Cache (for MySQL 5.7 and earlier)
query_cache_type = 0                    # Disable query cache for high-throughput
query_cache_size = 0                    # No memory allocated for query cache

# Logging
log_error = /var/log/mysql/error.log
slow_query_log = 1
slow_query_log_file = /var/log/mysql/mysql-slow.log
long_query_time = 2                     # Log queries taking more than 2 seconds
```

Example SQL Features:

```
-- JSON Operations (MySQL 8.0+)
SELECT id, JSON_EXTRACT(data, '$.name') AS name
FROM users
WHERE JSON_EXTRACT(data, '$.age') > 30;

-- Window Functions (MySQL 8.0+)
SELECT
    product_id,
    category_id,
    price,
    AVG(price) OVER (PARTITION BY category_id) AS category_avg_price,
    price - AVG(price) OVER (PARTITION BY category_id) AS diff_from_avg
FROM products;

-- Common Table Expressions (8.0+)
WITH monthly_sales AS (
    SELECT
        DATE_FORMAT(order_date, '%Y-%m') AS month,
        SUM(total_amount) AS sales
    FROM orders
    GROUP BY DATE_FORMAT(order_date, '%Y-%m')
)
```

```
SELECT month, sales,
       LAG(sales, 1) OVER (ORDER BY month) AS prev_month_sales,
       (sales - LAG(sales, 1) OVER (ORDER BY month)) / LAG(sales, 1) OVER (ORDER BY
month) * 100 AS growth_percent
FROM monthly_sales
ORDER BY month;
```

PostgreSQL

PostgreSQL is a powerful, open-source object-relational database system with a strong reputation for reliability, feature robustness, and performance.

Key Features:

- Advanced indexing (B-tree, Hash, GiST, SP-GiST, GIN, BRIN)
- Materialized views
- Table inheritance
- Foreign data wrappers
- Rich JSON/JSONB support
- Full-text search
- Multi-Version Concurrency Control (MVCC)
- Extensibility with custom data types and functions
- Highly extensible with extensions (PostGIS, TimescaleDB, etc.)

Architecture:

- Process-based architecture (one process per connection)
- MVCC for transaction isolation
- Write-Ahead Logging for durability
- Buffer cache for data caching
- Background processes for maintenance tasks

Strengths:

- Strong compliance with SQL standards
- Advanced features for complex queries
- Excellent data integrity and reliability
- Robust transaction support
- Strong support for geographic data (PostGIS)
- Extensibility and customization
- Support for advanced data types

Limitations:

- Can be more complex to configure and tune
- Slower for simple read operations compared to MySQL
- Higher memory usage per connection
- Replication was historically more complex (improved in recent versions)

Best Use Cases:

- Complex, data-intensive applications
- Geographic information systems (GIS)
- Systems requiring data integrity and complex validation
- Data warehousing and analytics
- Applications needing custom data types or functions

Example Configuration (postgresql.conf):

```
# Basic Settings
listen_addresses = '*'
port = 5432
max_connections = 200
shared_buffers = 2GB                    # 25% of RAM for dedicated DB server
work_mem = 64MB                         # Memory for sort operations
maintenance_work_mem = 256MB            # Memory for maintenance operations

# WAL Settings
wal_level = replica                     # Minimum for replication
max_wal_size = 1GB                      # Maximum WAL size before checkpoint
min_wal_size = 80MB                     # Minimum WAL size
checkpoint_timeout = 15min              # Max time between checkpoints

# Query Tuning
random_page_cost = 1.1                  # Lower for SSD storage (default 4.0)
effective_cache_size = 6GB              # Estimate of available memory (75% of RAM)
default_statistics_target = 100         # Statistics target (higher for complex
queries)

# Logging
log_destination = 'stderr'              # Log destination
logging_collector = on                  # Collect logs to files
log_directory = 'pg_log'                # Directory for log files
log_filename = 'postgresql-%Y-%m-%d_%H%M%S.log'
log_min_duration_statement = 1000       # Log queries taking longer than 1 second

# Replication
wal_keep_segments = 64                  # Number of WAL files to keep for
replication
hot_standby = on                        # Allow queries on standby servers
max_standby_archive_delay = 30s         # Max delay for standby servers
max_standby_streaming_delay = 30s       # Max delay for standby servers
```

Example SQL Features:

```
-- Common Table Expressions with Recursion
WITH RECURSIVE category_tree AS (
    -- Base case: top-level categories
    SELECT category_id, name, parent_id, 1 AS level, ARRAY[name] AS path
    FROM categories
    WHERE parent_id IS NULL
```

```
    UNION ALL

    -- Recursive case: child categories
    SELECT c.category_id, c.name, c.parent_id, ct.level + 1, array_append(ct.path,
c.name)
    FROM categories c
    JOIN category_tree ct ON c.parent_id = ct.category_id
)
SELECT category_id, name, level, array_to_string(path, ' > ') AS breadcrumb
FROM category_tree
ORDER BY path;

-- JSON Operations
CREATE TABLE user_data (
    id SERIAL PRIMARY KEY,
    data JSONB
);

INSERT INTO user_data (data) VALUES
('{"name": "John", "email": "john@example.com", "preferences": {"theme": "dark",
"notifications": true}}');

SELECT
    id,
    data->>'name' AS name,
    data->>'email' AS email,
    data->'preferences'->>'theme' AS theme
FROM user_data
WHERE data->'preferences'->>'notifications' = 'true';

-- Full-Text Search
CREATE TABLE articles (
    id SERIAL PRIMARY KEY,
    title TEXT,
    content TEXT
);

-- Create a tsvector column for full-text search
ALTER TABLE articles ADD COLUMN search_vector TSVECTOR;

-- Create an index on the tsvector column
CREATE INDEX idx_articles_search ON articles USING GIN(search_vector);

-- Update the tsvector column when inserting or updating
CREATE TRIGGER articles_search_update
BEFORE INSERT OR UPDATE ON articles
FOR EACH ROW EXECUTE FUNCTION
    tsvector_update_trigger(search_vector, 'pg_catalog.english', title, content);

-- Search for articles with specific terms
SELECT id, title, ts_headline(content, query) AS content_excerpt
FROM articles,
     to_tsquery('database & performance') AS query
WHERE search_vector @@ query
ORDER BY ts_rank(search_vector, query) DESC;
```

```
-- Range Types
CREATE TABLE reservations (
    id SERIAL PRIMARY KEY,
    room_id INTEGER,
    reserved_during TSRANGE,
    EXCLUDE USING GIST (room_id WITH =, reserved_during WITH &&)
);

-- The EXCLUDE constraint prevents overlapping reservations for the same room
INSERT INTO reservations (room_id, reserved_during)
VALUES (101, tsrange('2023-06-01', '2023-06-05'));

-- This will fail if it overlaps with an existing reservation
INSERT INTO reservations (room_id, reserved_during)
VALUES (101, tsrange('2023-06-03', '2023-06-07'));
```

Oracle Database

Oracle Database is a commercial, enterprise-grade relational database management system known for its performance, scalability, and comprehensive feature set.

Key Features:

- Advanced partitioning options
- Real Application Clusters (RAC) for high availability
- Automatic Storage Management (ASM)
- Data Guard for disaster recovery
- Advanced security features
- In-memory database option
- Multitenant architecture (pluggable databases)
- Parallel query execution
- Materialized views with query rewrite

Architecture:

- Shared everything architecture
- System Global Area (SGA) for shared memory
- Program Global Area (PGA) for session memory
- Background processes for database management
- Redo logs for recovery
- Table and index storage in tablespaces

Strengths:

- Excellent performance for large enterprise workloads
- Strong scalability features (vertical and horizontal)
- Advanced high availability options
- Comprehensive management and monitoring tools
- Strong security features
- Enterprise-grade support

Limitations:

- Higher cost (licensing and administration)
- Complex administration and tuning
- Resource-intensive
- Steep learning curve

Best Use Cases:

- Large enterprise applications
- Mission-critical systems requiring high availability
- Complex OLTP and OLAP workloads
- Financial systems
- Systems with stringent security requirements

Example Configuration (init.ora):

```
# Memory Configuration
memory_target = 8G                      # Automatic memory management
sga_target = 6G                         # System Global Area target
pga_aggregate_target = 2G               # Program Global Area target

# Connection Settings
processes = 300                         # Maximum number of processes
sessions = 330                          # Maximum number of sessions
transactions = 363                      # Maximum number of transactions

# Database Settings
db_name = PRODDB                        # Database name
db_block_size = 8192                    # Database block size
db_recovery_file_dest_size = 10G        # Recovery file destination size
db_recovery_file_dest = '/oracle/fast_recovery_area'

# Redo Log Settings
log_buffer = 64M                        # Redo log buffer size
fast_start_mttr_target = 300            # Mean time to recover target (seconds)

# Query Tuning
optimizer_mode = ALL_ROWS               # Optimizer mode for throughput
statistics_level = TYPICAL              # Statistics level for optimizer
```

Example SQL Features:

```
-- Analytic Functions
SELECT
    department_id,
    employee_id,
    salary,
    RANK() OVER (PARTITION BY department_id ORDER BY salary DESC) AS
dept_salary_rank,
    CUME_DIST() OVER (PARTITION BY department_id ORDER BY salary) * 100 AS
```

```
percentile
FROM employees;

-- Materialized Views with Query Rewrite
CREATE MATERIALIZED VIEW sales_summary
REFRESH COMPLETE ON DEMAND
ENABLE QUERY REWRITE
AS
SELECT
    product_id,
    customer_id,
    SUM(quantity) AS total_quantity,
    SUM(amount) AS total_amount
FROM sales
GROUP BY product_id, customer_id;

-- Interval Partitioning
CREATE TABLE sales (
    sale_id NUMBER,
    sale_date DATE,
    amount NUMBER,
    customer_id NUMBER
)
PARTITION BY RANGE (sale_date)
INTERVAL (NUMTOYMINTERVAL(1, 'MONTH'))
(
    PARTITION sales_initial VALUES LESS THAN (TO_DATE('01-JAN-2023', 'DD-MON-YYYY'))
);

-- Result Cache for Frequently Used Functions
CREATE OR REPLACE FUNCTION get_customer_level(
    p_customer_id IN NUMBER
) RETURN VARCHAR2
RESULT_CACHE RELIES_ON (customers)
AS
    v_total_purchases NUMBER;
    v_level VARCHAR2(20);
BEGIN
    SELECT SUM(amount) INTO v_total_purchases
    FROM purchases
    WHERE customer_id = p_customer_id;

    IF v_total_purchases > 10000 THEN
        v_level := 'PLATINUM';
    ELSIF v_total_purchases > 5000 THEN
        v_level := 'GOLD';
    ELSIF v_total_purchases > 1000 THEN
        v_level := 'SILVER';
    ELSE
        v_level := 'BRONZE';
    END IF;

    RETURN v_level;
END;
/
```

```
-- Flashback Query
SELECT * FROM employees
AS OF TIMESTAMP (SYSTIMESTAMP - INTERVAL '1' HOUR)
WHERE employee_id = 123;
```

Microsoft SQL Server

SQL Server is Microsoft's enterprise database platform, offering a comprehensive set of features for transaction processing, business intelligence, and analytics.

Key Features:

- Always On Availability Groups for high availability
- In-Memory OLTP (Hekaton)
- Columnstore indexes for analytics
- Temporal tables for historical data
- Query Store for performance tuning
- PolyBase for external data access
- Transparent Data Encryption (TDE)
- Row-level security
- Graph database capabilities

Architecture:

- SQLOS layer for resource management
- Buffer pool for data caching
- Multiple filegroups for data organization
- Transaction log for recovery
- Service Broker for asynchronous messaging
- Integration with Microsoft ecosystem

Strengths:

- Strong integration with Microsoft products
- Excellent developer tools (SSMS, Visual Studio)
- Built-in business intelligence features
- Good performance for mixed workloads
- Comprehensive administration tools
- Simple licensing for small deployments

Limitations:

- Primarily Windows-based (though Linux support is growing)
- Higher cost for enterprise features
- Resource-intensive for large deployments
- Complex licensing model

Best Use Cases:

- Enterprise applications in Microsoft environments

- Business intelligence and reporting
- Data warehousing
- .NET application backends
- Mixed OLTP and OLAP workloads

Example Configuration:

```
-- Memory Settings
EXEC sp_configure 'show advanced options', 1;
RECONFIGURE;
EXEC sp_configure 'max server memory (MB)', 8192; -- 8GB max memory
EXEC sp_configure 'min server memory (MB)', 2048; -- 2GB min memory
RECONFIGURE;

-- MAXDOP Setting for Parallel Queries
EXEC sp_configure 'max degree of parallelism', 4;
RECONFIGURE;

-- Cost Threshold for Parallelism
EXEC sp_configure 'cost threshold for parallelism', 50;
RECONFIGURE;

-- Database Settings
ALTER DATABASE YourDatabase
SET RECOVERY FULL;

ALTER DATABASE YourDatabase
SET AUTO_SHRINK OFF;

ALTER DATABASE YourDatabase
SET AUTO_CREATE_STATISTICS ON;

ALTER DATABASE YourDatabase
SET AUTO_UPDATE_STATISTICS ON;

-- Tempdb Configuration
ALTER DATABASE tempdb
MODIFY FILE (NAME = 'tempdev', SIZE = 1GB, FILEGROWTH = 256MB);
```

Example SQL Features:

```
-- Temporal Tables
CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    name VARCHAR(100) NOT NULL,
    position VARCHAR(100) NOT NULL,
    department VARCHAR(100) NOT NULL,
    salary DECIMAL(12, 2) NOT NULL,
    valid_from DATETIME2 GENERATED ALWAYS AS ROW START NOT NULL,
    valid_to DATETIME2 GENERATED ALWAYS AS ROW END NOT NULL,
    PERIOD FOR SYSTEM_TIME (valid_from, valid_to)
)
```

```
WITH (SYSTEM_VERSIONING = ON);

-- Query history of a record
SELECT employee_id, name, position, department, salary, valid_from, valid_to
FROM employees
FOR SYSTEM_TIME ALL
WHERE employee_id = 123
ORDER BY valid_from;

-- Columnstore Indexes
CREATE CLUSTERED COLUMNSTORE INDEX CCI_FactSales
ON fact_sales;

-- Query Store Configuration
ALTER DATABASE YourDatabase
SET QUERY_STORE = ON (
    OPERATION_MODE = READ_WRITE,
    CLEANUP_POLICY = (STALE_QUERY_THRESHOLD_DAYS = 30),
    DATA_FLUSH_INTERVAL_SECONDS = 900,
    MAX_STORAGE_SIZE_MB = 1000,
    INTERVAL_LENGTH_MINUTES = 60
);

-- Memory-Optimized Tables
CREATE TABLE dbo.order_details (
    order_id INT NOT NULL,
    product_id INT NOT NULL,
    quantity INT NOT NULL,
    unit_price MONEY NOT NULL,
    PRIMARY KEY NONCLUSTERED (order_id, product_id)
)
WITH (
    MEMORY_OPTIMIZED = ON,
    DURABILITY = SCHEMA_AND_DATA
);

-- JSON Support
SELECT
    product_id,
    name,
    JSON_VALUE(properties, '$.color') AS color,
    JSON_VALUE(properties, '$.size') AS size,
    JSON_QUERY(properties, '$.features') AS features
FROM products
WHERE ISJSON(properties) = 1
AND JSON_VALUE(properties, '$.color') = 'Red';
```

IBM Db2

IBM Db2 is a family of data management products that includes relational database servers for enterprise workloads, focusing on performance, reliability, and integration with IBM's ecosystem.

Key Features:

- Hybrid Transactional/Analytical Processing (HTAP)
- pureScale for high availability and scalability
- BLU Acceleration for in-memory columnar processing
- Adaptive compression
- Workload management
- Advanced security features
- Native encryption
- Multi-temperature data management

Strengths:

- Excellent performance for enterprise workloads
- Strong workload management capabilities
- Scalability for large databases
- Robust security features
- Good integration with IBM products

Limitations:

- Complex administration
- Higher cost
- Steeper learning curve
- Less community support compared to open-source options

Best Use Cases:

- Large enterprise applications
- IBM ecosystem integration
- High-volume transaction processing
- Mixed workload environments
- Financial systems

MariaDB

MariaDB is an open-source fork of MySQL, offering enhanced features and performance improvements while maintaining compatibility with MySQL.

Key Features:

- Multiple storage engines (InnoDB, XtraDB, ColumnStore, etc.)
- Advanced replication features
- Thread pool for connection handling
- GIS functionality
- JSON support
- Temporal data tables
- Parallel query execution
- Sequence objects

Strengths:

- MySQL compatibility
- Better performance in some scenarios

- More open development model
- Additional storage engines
- Enhanced security features

Best Use Cases:

- MySQL migration
- Web applications
- OLTP workloads
- Applications requiring specific MariaDB features

SQLite

SQLite is a self-contained, serverless, zero-configuration database engine that is embedded into applications rather than running as a separate process.

Key Features:

- Serverless architecture
- Zero configuration
- Single file database
- Cross-platform
- ACID compliant
- Small footprint

Strengths:

- Extremely lightweight
- No setup required
- Excellent for embedded applications
- Single file makes backup simple
- No server process needed
- Public domain license

Limitations:

- Limited concurrency
- Not suitable for high-volume multi-user scenarios
- Limited features compared to client-server databases
- No user management

Best Use Cases:

- Mobile applications
- Desktop applications
- Development/testing environments
- File format for data exchange
- Small websites
- Embedded systems

Comparison Matrix

Feature	MySQL	PostgreSQL	Oracle	SQL Server	MariaDB	SQLite
License	Dual (GPL/Commercial)	PostgreSQL License (Open Source)	Commercial	Commercial	GPL	Public Domain
Architecture	Thread-based	Process-based	Process-based	Thread-based	Thread-based	Embedded
Storage Engines	Multiple	Single with extensions	Single	Single	Multiple	Single
Partitioning	Yes	Yes	Advanced	Advanced	Yes	No
Replication	Master-Slave, Group	Streaming, Logical	Advanced	AlwaysOn	Advanced	N/A
JSON Support	Yes	Advanced (JSONB)	Yes	Yes	Yes	Basic
Full-Text Search	Basic	Advanced	Advanced	Advanced	Basic	Basic
Window Functions	Yes (8.0+)	Advanced	Advanced	Advanced	Yes	Basic (3.25+)
Materialized Views	No	Yes	Advanced	Yes	No	No
In-Memory Processing	No	No	Yes	Yes	No	Yes (in-memory only mode)
Geospatial	Basic	Advanced (PostGIS)	Advanced	Advanced	Basic	Basic
Columnar Storage	No	Extension	Yes	Yes	ColumnStore	No
Graph Capabilities	No	Extension	Yes	Yes	No	No
Cost	Free/Commercial	Free	High	Medium/High	Free	Free
Typical Workloads	Web, OLTP	General purpose, OLTP, GIS	Enterprise, mixed	Enterprise, BI	Web, OLTP	Embedded, local storage

Interview Questions

Q: What factors would you consider when choosing between MySQL and PostgreSQL for a new project?

A: When selecting between MySQL and PostgreSQL for a new project, I would consider several factors across technical, operational, and organizational dimensions:

Technical Requirements

1. **Data Complexity and Types**:
    - PostgreSQL has superior support for complex data types (arrays, JSON/JSONB, hstore, range types, custom types)
    - PostgreSQL offers better support for complex data models and inheritance
    - MySQL has simpler data type handling but is sufficient for many applications
2. **Query Complexity**:
    - PostgreSQL has more advanced query capabilities including better subquery support, more window functions, and CTE support
    - PostgreSQL has more sophisticated query planner
    - MySQL has improved significantly in version 8.0 but still lags in complex query performance
3. **Concurrency Model**:
    - PostgreSQL uses Multi-Version Concurrency Control (MVCC) that handles concurrent transactions more elegantly
    - PostgreSQL doesn't use read locks for most operations, which improves concurrency
    - MySQL's InnoDB also uses MVCC but with some implementation differences
4. **ACID Compliance**:
    - Both offer ACID compliance with InnoDB in MySQL
    - PostgreSQL has a reputation for stricter data integrity enforcement
    - MySQL has multiple storage engines with varying ACID properties
5. **Scalability Requirements**:
    - MySQL traditionally scales read operations better with simpler replication setup
    - PostgreSQL has better write scalability in some scenarios
    - PostgreSQL handles larger single-table sizes more efficiently
6. **Special Requirements**:
    - Geospatial needs: PostgreSQL with PostGIS is superior
    - Full-text search: PostgreSQL has more powerful built-in capabilities
    - JSON operations: PostgreSQL's JSONB type offers better performance and functionality

Operational Considerations

1. **Performance Characteristics**:
    - MySQL generally performs better for read-heavy workloads with simple queries
    - PostgreSQL often performs better for complex queries, writes, and mixed workloads
    - MySQL has lower memory overhead per connection
    - PostgreSQL has better parallel query execution
2. **Administration Overhead**:

- MySQL is generally considered easier to set up and manage for simpler use cases
- PostgreSQL requires more initial configuration but offers more control
- MySQL's replication is simpler to set up initially
- PostgreSQL's tools for maintenance are more comprehensive

3. **Backup and Recovery**:

- Both have point-in-time recovery capabilities
- PostgreSQL's Write-Ahead Logging (WAL) provides more reliable recovery
- MySQL's binary logs are simpler but often sufficient

4. **High Availability Options**:

- MySQL has a mature replication ecosystem
- PostgreSQL has streaming replication and logical replication
- Third-party tools for both systems offer automated failover

Organizational Factors

1. **Team Expertise**:

- Consider the existing knowledge and experience of your team
- Training requirements and learning curve differences
- Availability of DBA talent in your market

2. **Ecosystem and Integration**:

- MySQL has historically had better integration with popular web frameworks
- PostgreSQL has better support for standards and integrations with analytical tools
- Consider compatibility with other systems in your stack

3. **Licensing and Cost**:

- MySQL has dual licensing (GPL/Commercial) which can affect how you use it
- PostgreSQL has a more permissive license (PostgreSQL License)
- Commercial support options and associated costs

4. **Future-Proofing**:

- PostgreSQL development has been more consistent and community-driven
- MySQL's development direction has been influenced by corporate ownership changes
- Consider growth trajectory and potential future requirements

Project-Specific Examples

1. **For a typical web application with mostly CRUD operations**:

- MySQL may be sufficient and offer simpler setup
- Decision factors: Development speed, framework integration, team familiarity

2. **For a financial system with complex transactions**:

- PostgreSQL might be better due to stricter ACID compliance and transaction handling
- Decision factors: Data integrity, complex query support, consistency guarantees

3. **For a location-based service with heavy geospatial requirements**:
    - PostgreSQL with PostGIS would be the clear choice
    - Decision factors: Advanced geospatial functions, spatial indexing
4. **For a high-throughput logging or IoT data collection system**:
    - MySQL might perform better for simple inserts and basic aggregation queries
    - Decision factors: Write performance, simpler schema, partitioning capabilities

Migration Considerations

If there's a potential future need to migrate:

- PostgreSQL to MySQL migrations are typically more challenging than the reverse
- MySQL to PostgreSQL migrations are relatively straightforward with tools like pgloader
- Consider starting with PostgreSQL if there's any chance of needing its advanced features later

The decision ultimately depends on weighing these factors against your specific project requirements, team capabilities, and organizational constraints. For many applications, both databases would work well, but the specific needs should guide the final choice.

Q: Describe the key architectural differences between Oracle Database and open-source alternatives like PostgreSQL or MySQL. When would you recommend investing in Oracle?

A: Oracle Database and open-source alternatives like PostgreSQL and MySQL have significant architectural differences that impact their performance, scalability, feature set, and cost of ownership. Understanding these differences is crucial for making informed decisions about database technology investments.

Key Architectural Differences

1. Memory Architecture

Oracle Database:

- Sophisticated System Global Area (SGA) with multiple components:
    - Database Buffer Cache for caching data blocks
    - Shared Pool for SQL parsing and execution plans
    - Large Pool for parallel operations and backup/restore
    - Java Pool for Java VM memory
    - Streams Pool for Oracle Streams
- Program Global Area (PGA) for session-specific memory
- Automatic Memory Management to dynamically allocate memory between components
- In-Memory Column Store for analytics (Enterprise Edition)

PostgreSQL:

- Simpler memory model with shared buffers for data caching
- Work memory allocated per operation (not per session)
- Maintenance work memory for maintenance operations
- No automatic memory balancing between components
- Process-based architecture with separate memory spaces

MySQL:

- InnoDB Buffer Pool as the main memory structure
- Key Buffer for MyISAM tables
- Thread-based architecture sharing memory

2. Process Architecture

Oracle Database:

- Background processes handle various system tasks:
  - Database Writer (DBWn) for writing dirty buffers
  - Log Writer (LGWR) for managing redo logs
  - Checkpoint (CKPT) for coordinating checkpoints
  - System Monitor (SMON) for instance recovery
  - Process Monitor (PMON) for process recovery
  - Archiver (ARCn) for archiving redo logs
- Dedicated or shared server processes for user connections

PostgreSQL:

- Postmaster process as the main controller
- Backend processes for user connections (one per connection)
- Background writer for writing dirty buffers
- WAL writer for writing to write-ahead log
- Autovacuum launcher and workers for maintenance
- Separate processes mean better isolation but higher memory overhead

MySQL:

- Single multi-threaded server process
- Thread-based architecture (one thread per connection by default)
- Various background threads for specific tasks
- Lower memory overhead but potential for thread contention

3. Storage Architecture

Oracle Database:

- Logical storage structures:
  - Tablespaces for organizing database objects
  - Segments for individual objects (tables, indexes)
  - Extents for contiguous blocks
  - Blocks (typically 8KB) as the smallest unit
- Automatic Storage Management (ASM) for volume management
- Sophisticated tablespace management with multiple attributes
- Temporary, undo, and system tablespaces with specific roles
- Fine-grained control over storage parameters

PostgreSQL:

- Simpler storage model with databases and tablespaces

- Relation files for tables and indexes
- 8KB pages as the basic unit
- Write-Ahead Logging (WAL) for recovery
- No built-in volume management
- Table-level storage parameters

MySQL:

- Storage engine architecture allowing different engines for different tables
- Each storage engine has its own storage format
- InnoDB tablespaces and data files
- System tablespace and file-per-table options
- Simpler configuration options

4. Scalability Architecture

Oracle Database:

- Real Application Clusters (RAC) for horizontal scaling
- Active Data Guard for read scaling and high availability
- Sophisticated global cache coherency
- Parallel execution for queries, DML, DDL, and utilities
- Partitioning with many strategies (range, list, hash, composite, etc.)
- Sharding for distributed data management

PostgreSQL:

- Table partitioning support
- Parallel query execution
- No built-in clustering solution (third-party solutions like Citus)
- Streaming replication for read scaling
- Foreign data wrappers for distributed queries

MySQL:

- NDB Cluster for shared-nothing clustering
- InnoDB Cluster for high availability
- Replication (async, semi-sync, group) for read scaling
- Limited parallel query capabilities
- Partitioning support with fewer options than Oracle

5. Concurrency and Transaction Management

Oracle Database:

- Multi-Version Concurrency Control (MVCC)
- Undo segments for transaction rollback and read consistency
- Advanced isolation levels with read consistency
- Distributed transaction support with two-phase commit
- Sophisticated lock manager with various lock types and modes

PostgreSQL:

- Also uses MVCC but with a different implementation
- Every transaction gets a snapshot of the database
- Old row versions stored in the same table files
- VACUUM process needed to reclaim space
- Serializable Snapshot Isolation (SSI) for serializable transactions

MySQL:

- InnoDB uses MVCC with a different approach
- REPEATABLE READ default isolation level
- Simpler locking mechanism
- Limited distributed transaction support

When to Recommend Oracle Database

Given these architectural differences and Oracle's significant licensing costs, here are specific scenarios where investing in Oracle Database makes sense:

1. Mission-Critical Enterprise Applications

When:

- The application is core to business operations
- Downtime has severe financial impact
- Contractual SLAs require 99.99%+ availability
- Legal or regulatory requirements necessitate maximum reliability

Why Oracle:

- RAC provides superior high availability with active-active clustering
- Data Guard offers robust disaster recovery
- Online maintenance reduces planned downtime
- End-to-end diagnostics for faster problem resolution

Example:

```
"For a banking system processing $5B in daily transactions, we implemented Oracle
RAC with Data Guard, achieving 99.999% availability over five years, with no data
loss events despite two major data center incidents."
```

2. Complex, High-Volume OLTP Systems

When:

- Processing thousands of transactions per second
- Complex transactions spanning multiple tables
- Mixed workload (OLTP with reporting queries)
- Need for predictable performance under load

Why Oracle:

- Result cache for frequent queries
- Better parallel execution for DML operations
- More sophisticated query optimizer for complex queries
- In-Memory option for analytics alongside transactions

Example:

```
-- Oracle parallel DML with high performance
ALTER SESSION ENABLE PARALLEL DML;

INSERT /*+ APPEND PARALLEL(o,8) */ INTO orders_history
SELECT /*+ PARALLEL(o,8) */ * FROM orders o
WHERE order_date < ADD_MONTHS(SYSDATE, -24);
```

3. Large Data Warehouses with Complex Analytics

When:

- Petabyte-scale data
- Complex analytical queries
- Mixed query workloads
- Need for advanced optimization techniques

Why Oracle:

- Partitioning options for very large tables
- Materialized views with query rewrite
- Result cache for repeated analytical queries
- In-Memory Column Store for analytics
- Parallel execution with better resource management

Example:

```
-- Oracle materialized view with query rewrite
CREATE MATERIALIZED VIEW sales_summary
ENABLE QUERY REWRITE
AS
SELECT
    product_category,
    region,
    SUM(amount) as total_sales,
    COUNT(DISTINCT customer_id) as customer_count
FROM sales
GROUP BY product_category, region;

-- Original query is automatically rewritten to use the materialized view
SELECT product_category, region, SUM(amount)
FROM sales
GROUP BY product_category, region;
```

4. Regulated Industries with Strict Security Requirements

When:

- Industry has stringent regulatory requirements (finance, healthcare)
- Sensitive data requiring advanced protection
- Complex security policies beyond basic row-level security
- Need for comprehensive audit capabilities

Why Oracle:

- Virtual Private Database for row and column-level security
- Oracle Label Security for multi-level security policies
- Transparent Data Encryption with key management
- Comprehensive auditing with minimal performance impact
- Database Vault for privileged user access control

Example:

```
-- Oracle Virtual Private Database policy
CREATE OR REPLACE FUNCTION emp_policy (
    p_schema VARCHAR2,
    p_object VARCHAR2
)
RETURN VARCHAR2
AS
BEGIN
    RETURN 'department_id = SYS_CONTEXT(''USERENV'', ''DEPARTMENT_ID'')';
END;
/

BEGIN
    DBMS_RLS.ADD_POLICY (
        object_schema    => 'HR',
        object_name      => 'EMPLOYEES',
        policy_name      => 'EMP_POLICY',
        function_schema  => 'HR',
        policy_function  => 'EMP_POLICY',
        statement_types  => 'SELECT, INSERT, UPDATE, DELETE'
    );
END;
/
```

5. Applications Requiring Zero-Downtime Updates

When:

- 24/7 operation with no maintenance windows
- Global user base across time zones
- Schema changes needed while system is live
- Need for rolling upgrades

Why Oracle:

- Online Table Redefinition for schema changes without downtime
- Rolling upgrades with RAC
- Online index builds and rebuilds
- Edition-Based Redefinition for application upgrades

Example:

```
-- Oracle online table redefinition
BEGIN
    DBMS_REDEFINITION.START_REDEF_TABLE(
        uname          => 'SCHEMA',
        orig_table     => 'ORDERS',
        int_table      => 'ORDERS_INTERIM',
        col_mapping    => 'order_id, customer_id, order_date,
                           amount, DECODE(status, 1, ''PENDING'', 2, ''PROCESSING'',
                           3, ''SHIPPED'', 4, ''DELIVERED'') AS status'
    );
END;
/

-- Finish redefinition after sync
BEGIN
    DBMS_REDEFINITION.FINISH_REDEF_TABLE(
        uname          => 'SCHEMA',
        orig_table     => 'ORDERS',
        int_table      => 'ORDERS_INTERIM'
    );
END;
/
```

6. Organizations with Existing Oracle Investments

When:

- Significant investment in Oracle technologies
- Staff with Oracle expertise
- Integration with other Oracle products
- Standardized on Oracle technology stack

Why Oracle:

- Lower total cost due to existing infrastructure and expertise
- Integration with Oracle Middleware, Applications, and Cloud
- Consistent management across database fleet
- Leverage existing license agreements

Cost-Benefit Analysis Framework

Before recommending Oracle, I would always perform a detailed cost-benefit analysis:

```
1. Quantify the total cost:
   - License costs (processor or named user plus)
   - Annual support costs (22% of license cost)
   - Hardware costs (typically higher than open-source)
   - Administration costs (specialized DBA skills)

2. Quantify the benefits:
   - Reduced downtime risk (calculate cost of downtime)
   - Performance improvements (hardware savings, user productivity)
   - Reduced development time for complex features
   - Compliance/regulatory benefits

3. Compare with open-source alternatives total cost:
   - Hardware costs (possibly higher to achieve similar performance)
   - Commercial support if needed
   - Additional development costs for missing features
   - Risk costs from potentially lower reliability
```

Hybrid Approach

In many cases, a hybrid approach offers the best value:

1. Use Oracle for truly mission-critical systems requiring its advanced features
2. Use PostgreSQL or MySQL for secondary systems, development/test environments
3. Consider PostgreSQL with enterprise support for systems that need reliability but not Oracle-specific features
4. Consider Oracle Standard Edition for smaller systems needing Oracle compatibility

In conclusion, while open-source databases have closed the gap significantly in recent years, Oracle Database still offers architectural advantages for specific enterprise use cases where performance, reliability, security, and advanced features justify its cost. The recommendation should always balance technical requirements, business needs, and total cost of ownership.

3.6 Transaction Isolation Levels

Transaction isolation levels determine how transaction integrity is visible to other users and systems. Different isolation levels provide various trade-offs between performance and data consistency.

Understanding ACID Properties

Transaction isolation is one of the four ACID properties:

1. **Atomicity**: A transaction is all or nothing
2. **Consistency**: A transaction brings the database from one valid state to another
3. **Isolation**: Concurrent transactions should not affect each other
4. **Durability**: Once a transaction is committed, it remains so

Concurrency Problems

Without proper isolation, several concurrency problems can occur:

1. **Dirty Read**: Reading uncommitted data from another transaction
2. **Non-repeatable Read**: Getting different results when reading the same data multiple times within a transaction
3. **Phantom Read**: When new rows appear in a repeated query due to another transaction's insert
4. **Lost Update**: When two transactions read and update the same data, with one overwriting the other's changes

Standard Isolation Levels

SQL standard defines four isolation levels, from least to most strict:

1. **READ UNCOMMITTED**
2. **READ COMMITTED**
3. **REPEATABLE READ**
4. **SERIALIZABLE**

The following table shows which concurrency problems are prevented at each isolation level:

Isolation Level	Dirty Read	Non-repeatable Read	Phantom Read	Lost Update
READ UNCOMMITTED	No	No	No	No
READ COMMITTED	Yes	No	No	No
REPEATABLE READ	Yes	Yes	No*	Yes
SERIALIZABLE	Yes	Yes	Yes	Yes

* Some implementations of REPEATABLE READ (like MySQL's InnoDB) prevent phantom reads

Implementation in Different Databases

Different database systems implement isolation levels in various ways:

PostgreSQL:

```
-- Set isolation level for the current transaction
BEGIN;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- or REPEATABLE READ, or SERIALIZABLE
-- Execute statements
COMMIT;

-- Set default isolation level for the session
SET SESSION CHARACTERISTICS AS TRANSACTION ISOLATION LEVEL SERIALIZABLE;
```

MySQL/MariaDB:

```
-- Set isolation level for the current transaction
START TRANSACTION;
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

```
-- Execute statements
COMMIT;

-- Set default isolation level for the session
SET SESSION TRANSACTION ISOLATION LEVEL READ COMMITTED;

-- Set global default isolation level
SET GLOBAL TRANSACTION ISOLATION LEVEL READ COMMITTED;
```

SQL Server:

```
-- Set isolation level for the session
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
-- Options: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ, SERIALIZABLE,
SNAPSHOT

-- Using a specific isolation level for a query
SELECT *
FROM customers WITH (READCOMMITTED)
WHERE customer_id = 123;
```

Oracle:

```
-- Oracle only supports READ COMMITTED and SERIALIZABLE
-- Set isolation level for the session
ALTER SESSION SET ISOLATION_LEVEL = SERIALIZABLE;

-- Oracle also supports READ ONLY mode
SET TRANSACTION READ ONLY;
```

Implementation Mechanisms

Different databases use different mechanisms to implement isolation:

1. **Locking**:
    - Shared locks (S-locks) for reading
    - Exclusive locks (X-locks) for writing
    - Various lock granularities (row, page, table)
2. **Multi-Version Concurrency Control (MVCC)**:
    - Keeps multiple versions of data
    - Readers don't block writers, writers don't block readers
    - Used by PostgreSQL, Oracle, MySQL's InnoDB, and others
    - Implementations vary significantly between systems

Practical Examples

Example 1: Demonstrating READ UNCOMMITTED vs. READ COMMITTED

Terminal 1:

```
-- Using READ UNCOMMITTED
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;
-- Check account balance (initial value: $1000)
SELECT balance FROM accounts WHERE account_id = 123;
-- Output: 1000

-- Wait for Terminal 2 to update the balance but not commit

-- Check balance again (dirty read)
SELECT balance FROM accounts WHERE account_id = 123;
-- Output: 900 (reading uncommitted data)

-- Wait for Terminal 2 to roll back

-- Check balance again
SELECT balance FROM accounts WHERE account_id = 123;
-- Output: 1000 (the update was rolled back)
COMMIT;
```

Terminal 2:

```
BEGIN;
-- Update balance
UPDATE accounts SET balance = 900 WHERE account_id = 123;

-- Wait for Terminal 1 to read the updated balance

-- Roll back the transaction
ROLLBACK;
```

If Terminal 1 had used READ COMMITTED, the second query would have returned 1000 (the committed value) instead of 900.

Example 2: Demonstrating NON-REPEATABLE READ

Terminal 1:

```
-- Using READ COMMITTED
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
-- Check account balance
SELECT balance FROM accounts WHERE account_id = 123;
-- Output: 1000

-- Wait for Terminal 2 to update and commit
```

```
-- Check balance again (non-repeatable read)
SELECT balance FROM accounts WHERE account_id = 123;
-- Output: 900 (different from the first read)
COMMIT;
```

Terminal 2:

```
BEGIN;
-- Update balance
UPDATE accounts SET balance = 900 WHERE account_id = 123;
COMMIT;
```

If Terminal 1 had used REPEATABLE READ, the second query would have returned 1000 (the value from the beginning of the transaction) instead of 900.

Example 3: Demonstrating PHANTOM READ

Terminal 1:

```
-- Using REPEATABLE READ
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
-- Count products in a category
SELECT COUNT(*) FROM products WHERE category = 'Electronics';
-- Output: 100

-- Wait for Terminal 2 to insert a new product and commit

-- Count products again (might see phantom rows depending on the database)
SELECT COUNT(*) FROM products WHERE category = 'Electronics';
-- Output: 100 in MySQL InnoDB (which prevents phantom reads in REPEATABLE READ)
-- Output: 101 in some other databases that follow the SQL standard more strictly
COMMIT;
```

Terminal 2:

```
BEGIN;
-- Insert new product
INSERT INTO products (product_id, name, category, price)
VALUES (1001, 'New Gadget', 'Electronics', 499.99);
COMMIT;
```

If Terminal 1 had used SERIALIZABLE, it would consistently get the same count (100) regardless of the database system.

Choosing the Right Isolation Level

The choice of isolation level depends on the specific requirements:

1. **READ UNCOMMITTED**:
    - Highest performance, lowest consistency
    - Useful for reporting queries that can tolerate dirty reads
    - Rarely used in practice due to data integrity concerns
2. **READ COMMITTED**:
    - Good balance of performance and consistency for most applications
    - Default in many databases (PostgreSQL, Oracle, SQL Server)
    - Suitable for most OLTP workloads
3. **REPEATABLE READ**:
    - Stronger consistency guarantees
    - Default in MySQL/MariaDB
    - Good for transactions that make decisions based on queried data
4. **SERIALIZABLE**:
    - Strongest isolation, highest consistency
    - Lowest concurrency, potential for blocking or serialization failures
    - Use for financial transactions or when absolute consistency is required

Special Isolation Levels

Some databases offer additional isolation levels:

Snapshot Isolation (SQL Server, Oracle):

- Readers don't block writers and vice versa
- Each transaction sees a consistent snapshot of the database
- Prevents dirty reads, non-repeatable reads, and phantom reads
- Might allow some serialization anomalies

SQL Server example:

```
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
-- operations see a consistent snapshot of the database
COMMIT;
```

Read-Only Transactions (Oracle, PostgreSQL):

- Optimized for read-only workloads
- No locks acquired, uses snapshot of data
- Usually combined with another isolation level

Oracle example:

```
SET TRANSACTION READ ONLY;
-- queries see a consistent snapshot and acquire no locks
```

Performance Implications

Higher isolation levels generally mean:

- More locks or version records
- Longer lock hold times
- Increased potential for blocking or conflicts
- More system resource usage
- Potential for deadlocks

Best Practices

1. **Use the minimum isolation level required**:
    - Start with the default (often READ COMMITTED)
    - Increase only when needed for specific transactions
2. **Keep transactions short**:
    - Minimize the time locks are held
    - Perform reads before acquiring write locks when possible
3. **Order operations consistently**:
    - Access tables in the same order in different transactions
    - Helps prevent deadlocks
4. **Consider application-level optimistic concurrency**:
    - Use version numbers or timestamps
    - Check for conflicts before updating
5. **Be aware of database-specific behaviors**:
    - Isolation level implementations vary
    - Read documentation for your specific database

Interview Questions

Q: Explain the differences between transaction isolation levels and when you would use each.

A: Transaction isolation levels control how changes made by one transaction are visible to other concurrent transactions. They represent different trade-offs between data consistency and concurrency. Here's a comprehensive explanation of each isolation level and their appropriate use cases:

READ UNCOMMITTED

How it works:

- Transactions can read data that has been modified but not yet committed by other transactions
- No shared locks are acquired when reading data
- Dirty reads, non-repeatable reads, and phantom reads can all occur

Use cases:

- Data analysis or reporting queries where approximate results are acceptable
- Scenarios where performance is critical and data inconsistency can be tolerated
- Checking system status or gathering metrics where absolute precision isn't required
- Creating rough estimates or trends where exact point-in-time accuracy isn't necessary

Example scenario:

```
-- Transaction A (READ UNCOMMITTED)
SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;
BEGIN;
SELECT COUNT(*) FROM orders; -- Might include uncommitted orders
SELECT AVG(amount) FROM orders; -- Might include uncommitted amounts
COMMIT;
```

When to avoid:

- Financial calculations
- Inventory management
- Any scenario where data integrity is critical
- When making decisions based on query results

READ COMMITTED

How it works:

- Only reads data that has been committed
- Prevents dirty reads but allows non-repeatable reads and phantom reads
- Reading operations acquire shared locks but release them immediately
- Each statement sees only committed data as of the start of that statement

Use cases:

- General-purpose OLTP (Online Transaction Processing) applications
- When data consistency within a single statement is important
- Default isolation level for most applications
- When you need a balance between performance and consistency

Example scenario:

```
-- Transaction A (READ COMMITTED)
SET TRANSACTION ISOLATION LEVEL READ COMMITTED;
BEGIN;
SELECT * FROM accounts WHERE user_id = 123; -- Sees committed data

-- Meanwhile, Transaction B updates and commits new balance for user 123
```

```
SELECT * FROM accounts WHERE user_id = 123; -- Sees the new committed data
-- This is a non-repeatable read, as the same query returns different results
COMMIT;
```

When to avoid:

- When a transaction needs a consistent view of data throughout its execution
- When making decisions based on multiple reads that should be consistent
- Complex reports that require a point-in-time snapshot

REPEATABLE READ

How it works:

- Guarantees that if a row is read twice within the same transaction, the values will be consistent
- Prevents dirty reads and non-repeatable reads
- May still allow phantom reads (though some implementations like MySQL's InnoDB prevent them)
- Typically implemented using lock-based or snapshot-based approaches depending on the database

Use cases:

- Transactions that read data and then make decisions based on that data
- Financial calculations where consistency is required
- Scenarios where a transaction reads the same data multiple times and expects consistent results
- When consistency within a transaction is important, but new rows appearing is acceptable

Example scenario:

```
-- Transaction A (REPEATABLE READ)
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
SELECT balance FROM accounts WHERE user_id = 123; -- Returns $1000

-- Meanwhile, Transaction B updates and commits new balance for user 123

SELECT balance FROM accounts WHERE user_id = 123; -- Still returns $1000
-- The repeatable read isolation ensures we see the same balance

-- However, if Transaction B inserts a new account for user 123:
SELECT * FROM accounts WHERE user_id = 123; -- Might show the new account (phantom
read)
-- in standard implementations, though MySQL's InnoDB would prevent this
COMMIT;
```

When to avoid:

- In high-concurrency environments where the additional locking might cause performance issues
- When transactions run for a long time, as it increases the chances of conflicts
- When the application can handle occasional non-repeatable reads

SERIALIZABLE

How it works:

- The highest isolation level, providing the strictest transaction isolation
- Prevents dirty reads, non-repeatable reads, and phantom reads
- Makes transactions appear as if they were executed serially (one after another)
- Typically implemented using stronger locks or sophisticated techniques like serialization graphs

Use cases:

- Financial transactions requiring the highest level of consistency
- Critical data operations where any inconsistency is unacceptable
- Transactions where business rules depend on the complete state of a set of records
- Regulatory compliance scenarios requiring maximum isolation

Example scenario:

```
-- Transaction A (SERIALIZABLE)
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
SELECT SUM(balance) FROM accounts WHERE type = 'savings'; -- $50,000

-- Meanwhile, Transaction B tries to insert a new savings account
-- In most implementations, Transaction B will either be blocked or will fail with a
serialization error

SELECT SUM(balance) FROM accounts WHERE type = 'savings'; -- Still $50,000
-- No phantom reads allowed
COMMIT;
```

When to avoid:

- High-concurrency environments where performance is critical
- Any scenario where the performance impact outweighs the consistency requirements
- Long-running transactions, as they're more likely to conflict with other transactions

Special Isolation Levels

Some databases offer additional isolation levels not in the SQL standard:

SNAPSHOT Isolation (SQL Server, Oracle as "Serializable")

How it works:

- Each transaction sees a consistent snapshot of the database as it was at the start of the transaction
- Prevents dirty reads, non-repeatable reads, and phantom reads
- Uses row versioning instead of locks for reads, improving concurrency
- May still allow some serialization anomalies (write skew)

Use cases:

- Reporting queries that need a consistent view but shouldn't block other transactions
- Applications requiring high concurrency while still maintaining a consistent view
- Long-running read transactions alongside concurrent write operations

```
-- SQL Server example
SET TRANSACTION ISOLATION LEVEL SNAPSHOT;
BEGIN TRANSACTION;
-- All queries see a consistent snapshot from the start of the transaction
COMMIT;
```

READ ONLY (Oracle, PostgreSQL)

How it works:

- Transaction cannot perform any updates, only reads
- Often combined with another isolation level
- Optimization hint for the database to avoid acquiring certain locks

Use cases:

- Reporting and analysis queries
- Data export operations
- Any scenario where the transaction only needs to read data

```
-- Oracle example
SET TRANSACTION READ ONLY ISOLATION LEVEL SERIALIZABLE;
-- All reads see a consistent snapshot and no locks are acquired for updates
```

Practical Decision Framework

When choosing an isolation level for a specific transaction, I consider:

1. **Consistency Requirements**:
    - Does the transaction need to see a consistent view throughout its execution?
    - Are decisions made based on multiple reads that should be consistent?
    - Would phantom rows cause business logic problems?
2. **Performance Requirements**:
    - How critical is the performance of this transaction?
    - What is the concurrency level of the application?
    - How long does the transaction run?
3. **Database-Specific Implementation**:
    - How does the specific database implement each isolation level?
    - Are there any optimizations or extensions available?
4. **Error Handling Capabilities**:

- Can the application handle serialization failures or deadlocks?
- Is retry logic in place for transactions that might fail due to concurrency issues?

Real-World Examples

1. **E-commerce Order Processing**:
   - Use READ COMMITTED for most operations
   - Use REPEATABLE READ or SERIALIZABLE for the final checkout process where inventory checks and financial transactions occur
2. **Financial Account Transfers**:
   - Use SERIALIZABLE for transferring money between accounts

```
SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;
BEGIN;
SELECT balance FROM accounts WHERE account_id = 123 FOR UPDATE;
SELECT balance FROM accounts WHERE account_id = 456 FOR UPDATE;
-- Check sufficient funds and then perform transfer
UPDATE accounts SET balance = balance - 1000 WHERE account_id = 123;
UPDATE accounts SET balance = balance + 1000 WHERE account_id = 456;
COMMIT;
```

3. **Real-time Dashboard**:
   - Use READ UNCOMMITTED or READ COMMITTED for dashboard queries that can tolerate slightly stale data
4. **Report Generation**:
   - Use SNAPSHOT isolation or SERIALIZABLE with READ ONLY for consistent point-in-time reports
5. **Inventory Management**:
   - Use REPEATABLE READ for checking and updating inventory levels

```
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
SELECT quantity FROM inventory WHERE product_id = 456;
-- Business logic to determine if there's enough inventory
UPDATE inventory SET quantity = quantity - 5 WHERE product_id = 456;
COMMIT;
```

By understanding these isolation levels and their trade-offs, you can choose the appropriate level for each transaction in your application, balancing data consistency needs with performance requirements.

Q: Describe how MVCC (Multi-Version Concurrency Control) works and how it differs between PostgreSQL and MySQL's InnoDB.

A: Multi-Version Concurrency Control (MVCC) is a concurrency control method used by database management systems to provide concurrent access to database data without traditional locking mechanisms. Both PostgreSQL and MySQL's InnoDB implement MVCC, but they do so in significantly different ways. Let me explain how MVCC works generally and then detail the specific implementations.

General MVCC Principles

At its core, MVCC works by:

1. **Maintaining multiple versions of data**:
    - When data is modified, the database creates a new version rather than overwriting the old one
    - Each transaction sees a consistent snapshot of data based on its start time
2. **Transaction visibility rules**:
    - Each transaction can only see changes committed before it started
    - A transaction can always see its own changes
    - Different transactions may see different versions of the same data
3. **Version cleanup (vacuum)**:
    - Old versions need to be removed when they're no longer visible to any transaction
    - This process prevents unbounded storage growth
4. **Key benefits**:
    - Readers don't block writers (and vice versa)
    - Improved concurrency without sacrificing consistency
    - Consistent point-in-time snapshots for transactions

PostgreSQL's MVCC Implementation

PostgreSQL uses a tuple-based MVCC system with in-place storage of all versions:

1. Version Storage

Tuple Structure:

- Each row (tuple) in PostgreSQL contains:
    - `xmin`: Transaction ID that created this version
    - `xmax`: Transaction ID that deleted this version (or `0` if still valid)
    - Actual data values
    - Additional metadata like visibility flags

Data Organization:

- All row versions are stored in the same table files
- New versions don't physically replace old ones but coexist
- This approach is known as "append-only" or "no-overwrite"
- Tables grow in size until vacuum reclaims space

```
Table structure (simplified):
-----------------------------------------------------------
| xmin | xmax | ctid      | data columns...            |
-----------------------------------------------------------
| 100  | 0    | (0,1)     | name: "John", age: 30      |
| 200  | 0    | (0,2)     | name: "Alice", age: 25     |
| 300  | 400  | (0,3)     | name: "Bob", age: 40       | <- Dead tuple
| 400  | 0    | (0,4)     | name: "Bob", age: 42       | <- Updated version
-----------------------------------------------------------
```

2. Transaction Visibility

Transaction Snapshot:

- Each transaction gets a snapshot containing:
  - `xmin`: Oldest transaction ID still running
  - `xmax`: Next transaction ID to be assigned
  - `xip_list`: List of all concurrent active transaction IDs

Visibility Rules:

- A row version is visible if:
  - Its `xmin` is valid (committed before snapshot taken)
  - Its `xmax` is invalid (not yet committed, aborted, or after snapshot)
  - The creating transaction is in the `xip_list` (concurrent) and it's the current transaction

Implementation:

```
function is_visible(tuple, snapshot):
    # Created by a transaction still running when snapshot taken?
    if tuple.xmin in snapshot.xip_list:
        # Only visible if it's the current transaction
        return tuple.xmin == current_transaction_id

    # Created after snapshot taken?
    if tuple.xmin >= snapshot.xmax:
        return false

    # Created by already aborted transaction?
    if tuple.xmin is aborted in clog:
        return false

    # Deleted by a transaction?
    if tuple.xmax != 0:
        # Deleted by a transaction still running when snapshot taken?
        if tuple.xmax in snapshot.xip_list:
            # Still visible if deleted by someone else
            return tuple.xmax != current_transaction_id

        # Deleted by a transaction that committed before snapshot?
        if tuple.xmax < snapshot.xmin and tuple.xmax is committed in clog:
```

```
            return false

        # Deleted by a transaction that's after snapshot or aborted?
        # Then the deletion is not visible to us

    # If we get here, the tuple is visible
    return true
```

3. VACUUM Process

VACUUM Operation:

- Regular VACUUM: Marks space as reusable but doesn't return it to OS
- VACUUM FULL: Rewrites the entire table to reclaim space (heavy operation)
- Autovacuum: Background process that runs VACUUM automatically

What VACUUM does:

- Removes dead tuples (those not visible to any transaction)
- Updates free space map
- Freezes old transaction IDs to prevent wraparound
- Updates visibility map for index-only scans

4. Transaction Isolation Levels

READ COMMITTED:

- Takes a new snapshot for each statement
- Sees the latest committed version as of the start of each statement

REPEATABLE READ:

- Takes one snapshot at the start of the transaction
- Uses this same snapshot for all statements in the transaction

SERIALIZABLE:

- Uses snapshot isolation plus additional checks for serialization anomalies
- Monitors read/write dependencies to detect conflicts
- Fails with a serialization error if potential anomalies are detected

MySQL's InnoDB MVCC Implementation

InnoDB uses a different approach, centered around undo logs and a system called "consistent read":

1. Version Storage

Undo Logs:

- Current data is always stored in the main tablespace
- Previous versions are stored in separate undo log records
- Updates modify the data in-place, with old versions moved to undo logs

- Each record contains a transaction ID and a rollback pointer to previous versions

```
Main Table:
-----------------
| ID | Data     |
-----------------
| 1  | Bob: 42  | --> Undo Log: [TRX_ID: 400, Data: "Bob: 40"] --> [TRX_ID: 200,
Data: "Robert: 40"]
| 2  | Alice: 25|
-----------------
```

2. Transaction Visibility

System Variables:

- `trx_id`: Unique transaction identifier
- `trx_list`: List of active transactions
- `view`: Read view that determines which versions are visible

Read View Components:

- `up_limit_id`: Minimum transaction ID that is still active
- `low_limit_id`: Next transaction ID to be assigned
- `trx_ids`: List of active transactions when view was created

Visibility Rules:

- A version with transaction ID `trx_id` is visible if:
  - `trx_id` < `up_limit_id` (transaction committed before view created)
  - `trx_id` = current transaction ID (version created by current transaction)
  - `trx_id` not in `trx_ids` and `trx_id` < `low_limit_id` (committed transaction)

Implementation:

```
function is_visible(record, read_view):
    # Created by current transaction?
    if record.trx_id == current_transaction_id:
        return true

    # Created by transaction committed before view?
    if record.trx_id < read_view.up_limit_id:
        return true

    # Created by transaction in progress at view creation?
    if record.trx_id in read_view.trx_ids:
        return false

    # Created by transaction after view?
    if record.trx_id >= read_view.low_limit_id:
        return false
```

```
    # Otherwise, it's a committed transaction not active at view creation
    return true
```

3. Purge Process

InnoDB Purge Thread:

- Background thread that removes old versions from undo logs
- Only removes versions that are no longer needed by any transaction
- Frees space in the undo tablespace

History Length:

- Controlled by `innodb_history_list_length`
- Long-running transactions prevent purging of old versions

4. Transaction Isolation Levels

READ UNCOMMITTED:

- Doesn't use MVCC at all
- Reads the latest version regardless of transaction status

READ COMMITTED:

- Creates a new read view for each statement
- Sees versions committed before each statement execution

REPEATABLE READ:

- Creates a read view at the first read operation
- Uses the same read view throughout the transaction
- Unlike PostgreSQL, also prevents phantom reads via next-key locking

SERIALIZABLE:

- Similar to REPEATABLE READ but adds shared locks for all reads
- Uses locking rather than MVCC features for serializability

Key Differences Between PostgreSQL and InnoDB MVCC

1. **Storage of Versions**:
    - PostgreSQL: All versions stored in-place in table files
    - InnoDB: Current version in table, old versions in undo logs
2. **Update Mechanism**:
    - PostgreSQL: Append new versions, leaving old versions untouched
    - InnoDB: Update in-place, moving old version to undo logs
3. **Space Management**:

    - PostgreSQL: Requires explicit VACUUM to reclaim space
    - InnoDB: Automatic purging of old versions by background threads

4. **Transaction ID Handling**:
    - PostgreSQL: Uses 32-bit transaction IDs with wraparound protection
    - InnoDB: Uses 48-bit transaction IDs

5. **Isolation Level Implementation**:
    - PostgreSQL: REPEATABLE READ uses pure snapshot isolation
    - InnoDB: REPEATABLE READ uses snapshot isolation plus next-key locking

6. **Phantom Read Handling**:
    - PostgreSQL: REPEATABLE READ allows phantom reads (per SQL standard)
    - InnoDB: REPEATABLE READ prevents phantom reads (exceeds SQL standard)

7. **Serializable Implementation**:
    - PostgreSQL: Uses Serializable Snapshot Isolation (SSI) with conflict detection
    - InnoDB: Uses locking-based approach (shared locks on all reads)

Practical Implications

These implementation differences lead to several practical consequences:

1. Storage and Performance

PostgreSQL:

- Tables grow larger with updates until VACUUM runs
- Can experience "bloat" if VACUUM doesn't keep up
- More disk space required for heavily updated tables
- Better for read-heavy workloads with complex queries

InnoDB:

- More efficient space utilization with in-place updates
- Automated cleanup of old versions
- Better for write-heavy workloads
- Can have issues with long-running transactions preventing purging

2. Concurrency Behavior

PostgreSQL:

```
-- Transaction A: REPEATABLE READ
BEGIN;
SELECT * FROM products WHERE category = 'Electronics';  -- Returns 100 rows

-- Transaction B: Inserts a new electronics product and commits
```

```
-- Transaction A continues
SELECT * FROM products WHERE category = 'Electronics';  -- Still returns 100 rows
SELECT COUNT(*) FROM products WHERE category = 'Electronics';  -- Returns 100

-- If another transaction requires exclusive lock on the table:
ALTER TABLE products ADD COLUMN feature VARCHAR(100);  -- May wait for Transaction A
```

InnoDB:

```
-- Transaction A: REPEATABLE READ
BEGIN;
SELECT * FROM products WHERE category = 'Electronics';  -- Returns 100 rows

-- Transaction B: Inserts a new electronics product and commits

-- Transaction A continues
SELECT * FROM products WHERE category = 'Electronics';  -- Still returns 100 rows
SELECT COUNT(*) FROM products WHERE category = 'Electronics';  -- Returns 100 (no
phantoms)

-- If another transaction requires exclusive lock on a row that Transaction A read:
UPDATE products SET price = 99.99 WHERE product_id = 123;  -- Can proceed (different
from PostgreSQL)
```

3. Application Considerations

PostgreSQL:

- Regular VACUUM maintenance is critical
- Long-running transactions can significantly impact VACUUM effectiveness
- SERIALIZABLE transactions may fail with serialization failures (need retry logic)
- Better for applications with complex analytical queries

InnoDB:

- Long-running transactions prevent purging of undo logs
- May experience more lock contention at higher isolation levels
- SERIALIZABLE may cause more blocking due to locking approach
- Better for applications with high update rates

Code Examples Demonstrating Differences

Example 1: REPEATABLE READ and Phantom Reads

PostgreSQL:

```
-- Session 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
SELECT COUNT(*) FROM users WHERE age > 30;  -- Returns 10
```

```
-- Session 2
INSERT INTO users (name, age) VALUES ('Alice', 35);
COMMIT;

-- Session 1 continues
SELECT COUNT(*) FROM users WHERE age > 30;  -- Returns 10 (consistent snapshot)
-- But if we use a new predicate:
SELECT COUNT(*) FROM users WHERE age > 20;  -- May include Alice (phantom read
possible)
COMMIT;
```

InnoDB:

```
-- Session 1
SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;
BEGIN;
SELECT COUNT(*) FROM users WHERE age > 30;  -- Returns 10

-- Session 2
INSERT INTO users (name, age) VALUES ('Alice', 35);
COMMIT;

-- Session 1 continues
SELECT COUNT(*) FROM users WHERE age > 30;  -- Returns 10 (no phantom)
-- Even with a new predicate:
SELECT COUNT(*) FROM users WHERE age > 20;  -- Will not include Alice (no phantom)
COMMIT;
```

Example 2: VACUUM vs. Purge

PostgreSQL:

```
-- Create a test table
CREATE TABLE test_mvcc AS SELECT generate_series(1, 1000000) AS id;

-- Update all rows
BEGIN;
UPDATE test_mvcc SET id = id + 1;
COMMIT;

-- Check table size (will be approximately double until VACUUM)
SELECT pg_size_pretty(pg_relation_size('test_mvcc'));

-- Run VACUUM to reclaim space
VACUUM test_mvcc;

-- Check size again (should be smaller)
SELECT pg_size_pretty(pg_relation_size('test_mvcc'));
```

InnoDB:

```
-- Create a test table
CREATE TABLE test_mvcc (id INT) ENGINE=InnoDB;
INSERT INTO test_mvcc SELECT 1+seq FROM seq_1_to_1000000;

-- Update all rows
BEGIN;
UPDATE test_mvcc SET id = id + 1;
COMMIT;

-- Check table size (will not double as versions are in undo logs)
SELECT table_name, data_length, index_length
FROM information_schema.tables
WHERE table_name = 'test_mvcc';

-- Purge happens automatically, can be monitored with:
SHOW ENGINE INNODB STATUS;
```

Understanding these differences is crucial when designing applications that need to work efficiently with either database system, especially when dealing with transaction isolation requirements, concurrency, and performance optimization.

4. NoSQL Databases

NoSQL (Not Only SQL) databases are designed to handle different data structures and scaling requirements beyond what traditional relational databases can efficiently manage.

4.1 Document Databases

Document databases store data in flexible, JSON-like documents, making them ideal for applications with evolving schemas and complex hierarchical data.

Key Concepts

1. **Documents**: Self-contained, semi-structured data units (typically JSON or BSON)
2. **Collections**: Groups of related documents (similar to tables in RDBMS)
3. **Schema Flexibility**: Documents in the same collection can have different fields
4. **Nested Data**: Documents can contain embedded documents and arrays
5. **Query Language**: Database-specific languages to retrieve and manipulate documents

MongoDB

MongoDB is the most popular document database, known for its flexible schema, scalability, and rich query capabilities.

Basic CRUD Operations:

```
// Creating a document
db.customers.insertOne({
```

```
  name: 'John Smith',
  email: 'john@example.com',
  address: {
    street: '123 Main St',
    city: 'Boston',
    state: 'MA',
    zip: '02101',
  },
  phones: ['617-555-1234', '617-555-5678'],
  created_at: new Date(),
});

// Reading documents
// Find all customers in Boston
db.customers.find({ 'address.city': 'Boston' });

// Find a specific customer
db.customers.findOne({ email: 'john@example.com' });

// Updating documents
// Update a specific field
db.customers.updateOne(
  { email: 'john@example.com' },
  { $set: { 'address.zip': '02102' } }
);

// Add a new field to all documents in Boston
db.customers.updateMany(
  { 'address.city': 'Boston' },
  { $set: { region: 'Northeast' } }
);

// Deleting documents
// Delete one document
db.customers.deleteOne({ email: 'john@example.com' });

// Delete all customers in Boston
db.customers.deleteMany({ 'address.city': 'Boston' });
```

Querying with Operators:

```
// Comparison operators
db.products.find({ price: { $gt: 100, $lt: 200 } }); // Products between $100 and
$200
db.products.find({ category: { $in: ['Electronics', 'Computers'] } }); // In
specific categories

// Logical operators
db.products.find({
  $and: [{ price: { $lt: 500 } }, { rating: { $gte: 4.5 } }],
}); // Products under $500 with rating >= 4.5

// Element operators
```

```
db.products.find({ description: { $exists: true } }); // Products with a description
field
db.products.find({ tags: { $type: 'array' } }); // Products where tags is an array

// Array operators
db.products.find({ tags: 'wireless' }); // Products with "wireless" tag
db.products.find({ tags: { $all: ['wireless', 'bluetooth'] } }); // Products with
both tags
db.products.find({ 'reviews.rating': { $gt: 4 } }); // Products with at least one
review > 4
```

Aggregation Framework:

MongoDB's aggregation framework allows for complex data processing and analysis:

```
// Calculate average order amount by customer
db.orders.aggregate([
  {
    $group: {
      _id: '$customer_id',
      average_order: { $avg: '$total' },
      order_count: { $sum: 1 },
    },
  },
  { $sort: { average_order: -1 } },
  { $limit: 10 },
]);

// Complex aggregation pipeline
db.orders.aggregate([
  // Match orders from the past month
  {
    $match: {
      order_date: {
        $gte: new Date(new Date().setMonth(new Date().getMonth() - 1)),
      },
    },
  },
  // Unwind order items
  { $unwind: '$items' },
  // Group by product category
  {
    $group: {
      _id: '$items.category',
      total_revenue: {
        $sum: { $multiply: ['$items.price', '$items.quantity'] },
      },
      average_quantity: { $avg: '$items.quantity' },
      order_count: { $sum: 1 },
    },
  },
  // Only include categories with significant revenue
  { $match: { total_revenue: { $gt: 10000 } } },
```

```
  // Sort by revenue
  { $sort: { total_revenue: -1 } },
]);
```

Indexing in MongoDB:

```
// Create a single field index
db.customers.createIndex({ email: 1 }); // 1 for ascending order

// Create a unique index
db.customers.createIndex({ email: 1 }, { unique: true });

// Create a compound index
db.products.createIndex({ category: 1, price: -1 }); // -1 for descending order

// Create a text index for full-text search
db.products.createIndex({ description: 'text', name: 'text' });

// Create a geospatial index
db.stores.createIndex({ location: '2dsphere' });

// TTL index (documents expire after 30 days)
db.sessions.createIndex({ createdAt: 1 }, { expireAfterSeconds: 2592000 });
```

Transactions in MongoDB:

```
// Start a session
const session = db.getMongo().startSession();

// Start a transaction
session.startTransaction();

try {
  // Perform operations within the transaction
  const ordersCollection = session.getDatabase('mydb').getCollection('orders');
  const inventoryCollection = session
    .getDatabase('mydb')
    .getCollection('inventory');

  // Insert order
  ordersCollection.insertOne({
    customer_id: '12345',
    items: [{ product_id: 'ABC123', quantity: 2, price: 29.99 }],
    total: 59.98,
    status: 'pending',
  });

  // Update inventory
  const result = inventoryCollection.updateOne(
    { product_id: 'ABC123', quantity: { $gte: 2 } },
    { $inc: { quantity: -2 } }
```

```
    );

    // Check if inventory was updated successfully
    if (result.modifiedCount !== 1) {
      throw new Error('Insufficient inventory');
    }

    // Commit the transaction
    session.commitTransaction();
  } catch (error) {
    // Abort transaction on error
    session.abortTransaction();
    console.error('Transaction aborted:', error);
  } finally {
    // End session
    session.endSession();
  }
```

Couchbase

Couchbase is a distributed NoSQL document database with a strong focus on high performance, scalability, and flexible data model.

Key Features:

- SQL-like query language (N1QL)
- Memory-first architecture with disk persistence
- Built-in caching layer
- Distributed architecture with easy scaling
- Full-text search capabilities
- Real-time analytics

N1QL Query Examples:

```
-- Basic SELECT query
SELECT name, address, phone
FROM `users`
WHERE type = "customer" AND status = "active";

-- JOIN between documents
SELECT o.order_id, o.order_date, c.name, c.email
FROM `orders` o
JOIN `customers` c ON KEYS o.customer_id
WHERE o.status = "shipped"
  AND o.order_date BETWEEN "2023-01-01" AND "2023-01-31";

-- Nested data and arrays
SELECT name, phone, address.city
FROM `customers`
WHERE ANY hobby IN hobbies SATISFIES hobby = "photography" END
  AND address.country = "Canada";
```

```
-- Aggregation
SELECT product_category,
       COUNT(*) AS product_count,
       AVG(price) AS avg_price,
       MIN(price) AS min_price,
       MAX(price) AS max_price
FROM `products`
WHERE is_active = true
GROUP BY product_category
HAVING COUNT(*) > 5
ORDER BY avg_price DESC;
```

Document Database Schema Design Patterns

Embedding vs. Referencing:

1. **Embedding (Denormalization):**
   - Store related data in a single document
   - Good for data that's frequently accessed together
   - Simplifies queries and improves read performance
   - Can lead to data duplication

```
// Embedding example: Customer with orders
{
  "_id": "customer123",
  "name": "John Smith",
  "email": "john@example.com",
  "orders": [
    {
      "order_id": "order1",
      "date": "2023-05-15",
      "items": [
        { "product": "Laptop", "price": 1299.99, "quantity": 1 },
        { "product": "Mouse", "price": 24.99, "quantity": 1 }
      ],
      "total": 1324.98
    },
    {
      "order_id": "order2",
      "date": "2023-06-02",
      "items": [
        { "product": "Headphones", "price": 149.99, "quantity": 1 }
      ],
      "total": 149.99
    }
  ]
}
```

2. **Referencing (Normalization):**

- Store references to related documents
- Good for data that's shared across multiple documents
- Reduces duplication and update anomalies
- Requires multiple queries or joins

```
// Customer document
{
  "_id": "customer123",
  "name": "John Smith",
  "email": "john@example.com"
}

// Order documents
{
  "_id": "order1",
  "customer_id": "customer123",
  "date": "2023-05-15",
  "items": [
    { "product_id": "prod1", "price": 1299.99, "quantity": 1 },
    { "product_id": "prod2", "price": 24.99, "quantity": 1 }
  ],
  "total": 1324.98
}

{
  "_id": "order2",
  "customer_id": "customer123",
  "date": "2023-06-02",
  "items": [
    { "product_id": "prod3", "price": 149.99, "quantity": 1 }
  ],
  "total": 149.99
}
```

Common Schema Design Patterns:

1. **Subset Pattern:**
   - Store a subset of fields from a related document
   - Useful for reducing the need for additional queries

```
// Product document with subset of category information
{
  "_id": "prod123",
  "name": "Ultra HD Monitor",
  "price": 499.99,
  "category": {
    "_id": "cat456",
    "name": "Monitors", // Subset of category data
    "department": "Electronics" // Subset of category data
  }
}
```

2. **Extended Reference Pattern:**
   - Store frequently accessed data from related documents
   - Update this data when the source document changes

```
// Order document with extended customer reference
{
  "_id": "order123",
  "customer": {
    "_id": "cust456",
    "name": "Alice Johnson", // Extended reference
    "email": "alice@example.com", // Extended reference
    "tier": "Gold" // Extended reference
  },
  "items": [...],
  "total": 256.78
}
```

3. **Computed Pattern:**
   - Store computed values to avoid recalculating them

```
// Product with pre-computed values
{
  "_id": "prod789",
  "name": "Smartphone",
  "base_price": 699.99,
  "discount": 100.00,
  "final_price": 599.99, // Pre-computed
  "tax": 48.00, // Pre-computed
  "total_price": 647.99 // Pre-computed
}
```

4. **Schema Versioning Pattern:**
   - Include a version field to track schema changes
   - Helps with backward compatibility and migrations

```
{
  "_id": "user123",
  "schema_version": 2, // Track schema version
  "name": "Bob Smith",
  "email": "bob@example.com",
  "preferences": {
    "theme": "dark",
    "notifications": true
  }
}
```

5. **Polymorphic Pattern:**
    - Store different types of entities in the same collection
    - Use a type field to distinguish between them

```
// Person entity
{
  "_id": "entity1",
  "type": "person",
  "name": "David Wilson",
  "birth_date": "1982-08-15"
}

// Company entity in the same collection
{
  "_id": "entity2",
  "type": "company",
  "name": "Acme Corp",
  "founded": "1995-03-20",
  "employees": 250
}
```

Interview Questions

Q: Compare the data modeling approaches in MongoDB with traditional relational database design. When would you choose embedded documents versus using references?

A: Data modeling in MongoDB differs fundamentally from relational database design, requiring a shift in thinking from normalized tables to document-oriented structures. Let me compare these approaches and discuss when to use embedded documents versus references in MongoDB.

Relational vs. Document Data Modeling

Relational Database Modeling

In relational databases, we typically:

1. **Normalize data**: Organize data into separate tables to minimize redundancy
2. **Define relationships**: Use foreign keys to establish connections between tables
3. **Enforce integrity**: Apply constraints (primary keys, foreign keys, unique constraints)
4. **Use joins**: Combine data from multiple tables when retrieving information

For example, a simple e-commerce schema might look like:

```
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE,
    address_id INT REFERENCES addresses(address_id)
```

```
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT REFERENCES customers(customer_id),
    order_date DATE,
    status VARCHAR(20)
);

CREATE TABLE order_items (
    order_id INT REFERENCES orders(order_id),
    product_id INT REFERENCES products(product_id),
    quantity INT,
    price DECIMAL(10,2),
    PRIMARY KEY (order_id, product_id)
);
```

MongoDB Document Modeling

In MongoDB, we focus on:

1. **Document structure**: Organizing data in JSON-like BSON documents
2. **Schema flexibility**: Documents in the same collection can have different fields
3. **Embedding vs. referencing**: Deciding when to nest related data or use references
4. **Access patterns**: Designing documents based on how the data will be accessed
5. **Atomic operations**: MongoDB guarantees atomicity at the document level

The same e-commerce data could be modeled in MongoDB using embedded documents:

```
// Embedded approach
{
  "_id": ObjectId("5f8a716b0c4f4913f9c3d0e1"),
  "name": "Jane Smith",
  "email": "jane@example.com",
  "address": {
    "street": "123 Main St",
    "city": "Boston",
    "state": "MA",
    "zip": "02101"
  },
  "orders": [
    {
      "order_id": "ORD12345",
      "order_date": ISODate("2023-04-15"),
      "status": "shipped",
      "items": [
        {
          "product_id": "PROD789",
          "name": "Wireless Headphones",
          "quantity": 1,
          "price": 79.99
        },
```

```
        {
          "product_id": "PROD456",
          "name": "Phone Case",
          "quantity": 2,
          "price": 19.99
        }
      ],
      "total": 119.97
    }
  ]
}
```

Or using references:

```
// Customer document
{
  "_id": ObjectId("5f8a716b0c4f4913f9c3d0e1"),
  "name": "Jane Smith",
  "email": "jane@example.com",
  "address": {
    "street": "123 Main St",
    "city": "Boston",
    "state": "MA",
    "zip": "02101"
  }
}

// Order document
{
  "_id": ObjectId("6a1b83cd1e5f5b24g0d4e1f2"),
  "order_id": "ORD12345",
  "customer_id": ObjectId("5f8a716b0c4f4913f9c3d0e1"),
  "order_date": ISODate("2023-04-15"),
  "status": "shipped",
  "items": [
    {
      "product_id": ObjectId("7c3e94df2g6h7i5j8k9l0m1"),
      "quantity": 1,
      "price": 79.99
    },
    {
      "product_id": ObjectId("8d4f05eg3h7i6j9k0l1m2n3"),
      "quantity": 2,
      "price": 19.99
    }
  ],
  "total": 119.97
}
```

Key Differences in Approach

1. **Schema Rigidity**:

- Relational: Schema must be defined upfront, changes require ALTER TABLE
- MongoDB: Schema is flexible, documents in the same collection can have different structures

2. **Relationships**:
   - Relational: Enforced through foreign keys and joins
   - MongoDB: Implemented through embedding or references, not enforced by the database

3. **Normalization vs. Denormalization**:
   - Relational: Emphasizes normalization to reduce redundancy
   - MongoDB: Often embraces strategic denormalization for performance

4. **Transaction Scope**:
   - Relational: Transactions across multiple tables
   - MongoDB: Prior to v4.0, atomic operations only at document level; now supports multi-document transactions but with performance implications

5. **Query Patterns**:
   - Relational: Optimized for complex joins and normalized data
   - MongoDB: Optimized for retrieving complete documents with minimal joins

When to Use Embedded Documents vs. References

The decision between embedding related data or using references depends on several factors:

Use Embedded Documents When:

1. **One-to-Few Relationships**:
   - When a parent has a small, fixed number of children
   - Example: A user with a few addresses (home, work, shipping)

```
{
  "_id": ObjectId("..."),
  "name": "John Doe",
  "email": "john@example.com",
  "addresses": [
    { "type": "home", "street": "123 Main St", "city": "Boston" },
    { "type": "work", "street": "456 Market St", "city": "Boston" }
  ]
}
```

2. **Data is Usually Accessed Together**:
   - When the related data is almost always retrieved together
   - Example: Order with its line items

```
{
  "_id": ObjectId("..."),
  "order_id": "ORD12345",
  "customer_id": ObjectId("..."),
  "order_date": ISODate("2023-05-15"),
  "items": [
    { "product_id": ObjectId("..."), "quantity": 2, "price": 29.99 },
    { "product_id": ObjectId("..."), "quantity": 1, "price": 49.99 }
  ],
  "total": 109.97
}
```

3. **Data Updates Atomically**:
   - When the embedded data needs to be updated together with the parent
   - Example: Product with inventory information

```
{
  "_id": ObjectId("..."),
  "name": "Wireless Headphones",
  "price": 79.99,
  "inventory": {
    "in_stock": 42,
    "reserved": 5,
    "available": 37
  }
}
```

4. **Read Performance is Critical**:
   - When you need to retrieve the complete object in a single query
   - Example: Blog post with comments for display

```
{
  "_id": ObjectId("..."),
  "title": "MongoDB Data Modeling",
  "content": "...",
  "author": "Jane Smith",
  "date": ISODate("2023-01-15"),
  "comments": [
    {
      "user": "Bob Jones",
      "text": "Great article!",
      "date": ISODate("2023-01-16T14:25:00Z")
    },
    {
      "user": "Alice Wong",
      "text": "This was very helpful.",
      "date": ISODate("2023-01-16T18:10:00Z")
    }
```

```
  ]
}
```

5. **Document Size Remains Manageable**:
   - When embedding won't cause the document to approach the 16MB size limit
   - When the number of embedded elements won't grow unbounded

Use References When:

1. **One-to-Many or Many-to-Many Relationships**:
   - When a parent can have many children
   - Example: A customer with many orders over time

```
// Customer document
{
  "_id": ObjectId("..."),
  "name": "Emily Johnson",
  "email": "emily@example.com"
}

// Order documents (stored in orders collection)
{
  "_id": ObjectId("..."),
  "order_id": "ORD67890",
  "customer_id": ObjectId("..."),  // Reference to customer
  "order_date": ISODate("2023-06-10"),
  "items": [...]
}
```

2. **Data Can Be Accessed Independently**:
   - When the related data is often accessed separately
   - Example: Products referenced in orders

```
// Product document (in products collection)
{
  "_id": ObjectId("..."),
  "name": "Ultra HD Monitor",
  "price": 499.99,
  "description": "...",
  "specifications": { ... }
}

// Order document with product references
{
  "_id": ObjectId("..."),
  "customer_id": ObjectId("..."),
  "items": [
    { "product_id": ObjectId("..."), "quantity": 1, "price": 499.99 }
```

```
  ]
}
```

3. **Data Changes Frequently**:
   - When the related data is updated often and independently
   - Example: Product inventory levels

```
// Product document
{
  "_id": ObjectId("..."),
  "name": "Smartphone",
  "price": 799.99,
  "category": "Electronics"
}

// Separate inventory document
{
  "_id": ObjectId("..."),
  "product_id": ObjectId("..."),
  "warehouse_id": "WH-005",
  "quantity": 157,
  "last_updated": ISODate("2023-06-15T14:30:00Z")
}
```

4. **Many Entities Reference the Same Data**:
   - When the data is shared across many parents
   - Example: Categories for products

```
// Category document
{
  "_id": ObjectId("..."),
  "name": "Smartphones",
  "parent_category": ObjectId("...")  // Electronics
}

// Product documents reference the category
{
  "_id": ObjectId("..."),
  "name": "iPhone 14",
  "category_id": ObjectId("...")  // Reference to Smartphones
}
```

5. **Document Would Exceed Size Limits**:
   - When embedding would cause the document to approach the 16MB size limit
   - Example: An article with thousands of comments

```
// Article document
{
  "_id": ObjectId("..."),
  "title": "Popular Article",
  "content": "..."
}

// Comments in separate collection
{
  "_id": ObjectId("..."),
  "article_id": ObjectId("..."),
  "user": "Bob Smith",
  "text": "Great read!",
  "date": ISODate("2023-06-01T10:15:00Z")
}
```

Hybrid Approaches

Often, the best solution is a hybrid approach:

1. **Extended References**:
   - Store a subset of frequently accessed fields from the referenced document
   - Example: Store basic product info in order items, but keep full product details separate

```
// Order with extended references
{
  "_id": ObjectId("..."),
  "customer_id": ObjectId("..."),
  "items": [
    {
      "product_id": ObjectId("..."),
      "product_name": "Ultra HD Monitor",  // Extended reference
      "product_category": "Electronics",   // Extended reference
      "quantity": 1,
      "price": 499.99
    }
  ]
}
```

2. **Subset Embedding**:
   - Embed only the most recent or important items
   - Example: Blog post with the 5 most recent comments embedded, older ones referenced

```
{
  "_id": ObjectId("..."),
  "title": "MongoDB Data Modeling",
  "content": "...",
  "recent_comments": [  // Only most recent comments embedded
```

```
      {
        "id": ObjectId("..."),
        "user": "Alice Wong",
        "text": "This was very helpful.",
        "date": ISODate("2023-01-16T18:10:00Z")
      }
    ],
    "comment_count": 42  // Total comments (most stored separately)
  }
```

3. **Bucketing**:
   - Group related data into fixed-size "buckets"
   - Example: Sensor readings grouped by day

```
// Daily buckets of sensor readings
{
  "_id": ObjectId("..."),
  "sensor_id": "TEMP-001",
  "date": ISODate("2023-06-15"),
  "readings": [
    { "time": ISODate("2023-06-15T00:15:00Z"), "value": 22.5 },
    { "time": ISODate("2023-06-15T00:30:00Z"), "value": 22.7 },
    // ... more readings for the day
  ]
}
```

Real-World Example: E-commerce Application

For an e-commerce application, I might design the schema as follows:

```
// User collection
{
  "_id": ObjectId("..."),
  "name": "Sarah Johnson",
  "email": "sarah@example.com",
  "addresses": [  // Embedded one-to-few relationship
    {
      "type": "shipping",
      "street": "123 Main St",
      "city": "Portland",
      "state": "OR",
      "zip": "97201",
      "default": true
    },
    {
      "type": "billing",
      "street": "123 Main St",
      "city": "Portland",
      "state": "OR",
      "zip": "97201"
```

```
    }
  ],
  "payment_methods": [  // Embedded one-to-few relationship
    {
      "type": "credit_card",
      "last4": "4242",
      "exp_month": 12,
      "exp_year": 2025,
      "default": true
    }
  ]
}

// Product collection
{
  "_id": ObjectId("..."),
  "name": "Ultra HD Monitor",
  "description": "...",
  "price": 499.99,
  "category_id": ObjectId("..."),  // Reference to category
  "category_name": "Monitors",     // Extended reference
  "specifications": { ... },
  "tags": ["electronics", "computer accessories", "displays"]
}

// Category collection (referenced by products)
{
  "_id": ObjectId("..."),
  "name": "Monitors",
  "parent_id": ObjectId("..."),  // Reference to Electronics category
  "path": "/Electronics/Monitors"
}

// Order collection
{
  "_id": ObjectId("..."),
  "order_number": "ORD-12345",
  "user_id": ObjectId("..."),  // Reference to user
  "user_info": {  // Extended reference
    "name": "Sarah Johnson",
    "email": "sarah@example.com"
  },
  "shipping_address": {  // Embedded copy at time of order
    "street": "123 Main St",
    "city": "Portland",
    "state": "OR",
    "zip": "97201"
  },
  "order_date": ISODate("2023-06-15T10:30:00Z"),
  "status": "shipped",
  "items": [  // Embedded one-to-many relationship
    {
      "product_id": ObjectId("..."),  // Reference to product
      "product_name": "Ultra HD Monitor",  // Extended reference
      "quantity": 1,
```

```
      "price": 499.99
    },
    {
      "product_id": ObjectId("..."),
      "product_name": "HDMI Cable",
      "quantity": 2,
      "price": 12.99
    }
  ],
  "shipping": 15.99,
  "tax": 42.08,
  "total": 570.05,
  "payment": {
    "method": "credit_card",
    "last4": "4242",
    "amount": 570.05,
    "status": "completed",
    "transaction_id": "txn_123456"
  },
  "shipments": [  // Embedded array for order fulfillment
    {
      "carrier": "UPS",
      "tracking_number": "1Z999AA10123456784",
      "status": "delivered",
      "shipped_date": ISODate("2023-06-16T14:20:00Z"),
      "delivered_date": ISODate("2023-06-18T13:45:00Z")
    }
  ]
}

// Review collection (could be embedded in product, but separate for scalability)
{
  "_id": ObjectId("..."),
  "product_id": ObjectId("..."),  // Reference to product
  "user_id": ObjectId("..."),     // Reference to user
  "user_name": "Sarah J.",        // Extended reference
  "rating": 4.5,
  "title": "Great monitor, minor issues",
  "review": "...",
  "date": ISODate("2023-06-20T09:15:00Z"),
  "helpful_votes": 12
}

// Inventory collection (separate because it changes frequently)
{
  "_id": ObjectId("..."),
  "product_id": ObjectId("..."),  // Reference to product
  "warehouse_id": "WH-005",
  "quantity": 157,
  "reserved": 10,
  "available": 147,
  "last_updated": ISODate("2023-06-15T14:30:00Z")
}
```

In this design:

- User addresses and payment methods are embedded (one-to-few)
- Product references its category (one-to-one)
- Order embeds its line items (one-to-many but bounded)
- Reviews are kept separate from products (potentially many-to-one)
- Inventory is separate from products (frequently changing data)

This balances:

- Read performance (embedded documents for commonly accessed data)
- Write performance (separate collections for frequently changing data)
- Data integrity (extended references for important lookup data)
- Scalability (avoiding unbounded array growth)

The key to effective MongoDB data modeling is understanding your application's query patterns and prioritizing the operations that need to be most efficient, then designing your document structure accordingly.

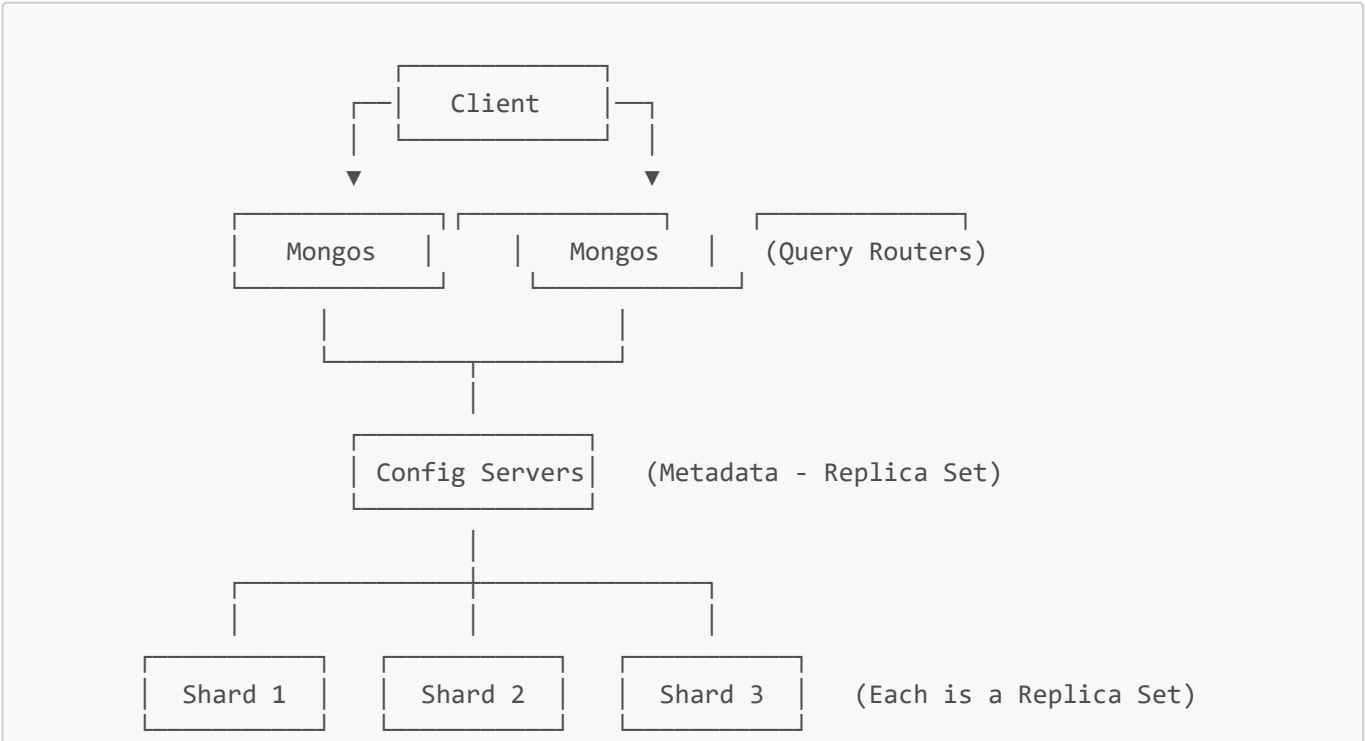
Q: Explain how you would design a sharded MongoDB deployment for a high-volume application. What considerations would you take into account for shard key selection?

A: Designing a sharded MongoDB deployment for a high-volume application requires careful planning across multiple dimensions including hardware, configuration, data distribution, and operational practices. The shard key selection is particularly critical as it determines how data is distributed across the cluster and directly impacts performance, scalability, and maintenance.

Sharded MongoDB Architecture Overview

A sharded MongoDB deployment consists of three main components:

1. **Shard Servers**: Each shard is a replica set containing a subset of the data
2. **Config Servers**: Store metadata about the cluster, including chunk distribution
3. **Mongos Routers**: Route queries to appropriate shards and aggregate results



Designing a Sharded MongoDB Deployment

1. Hardware Planning

Shard Servers:

- High disk I/O capacity (SSDs preferred)
- Sufficient RAM for working set (typically 50% of data size)
- Multiple CPU cores for query processing
- Fast network connections between nodes

Config Servers:

- Less resource-intensive than shards
- 3-node replica set configuration (required)
- SSDs for metadata storage
- Redundant power and network connections

Mongos Routers:

- CPU-intensive rather than I/O-intensive
- Multiple instances for high availability
- Can be colocated with application servers
- 1 or more instances per application server

2. Initial Configuration

```
// 1. Set up config server replica set
rs.initiate({
  _id: 'configRS',
  configsvr: true,
  members: [
    { _id: 0, host: 'config1.example.com:27019' },
    { _id: 1, host: 'config2.example.com:27019' },
    { _id: 2, host: 'config3.example.com:27019' },
  ],
});

// 2. Set up shard replica sets (repeat for each shard)
rs.initiate({
  _id: 'shard1RS',
  members: [
    { _id: 0, host: 'shard1-a.example.com:27018', priority: 2 },
    { _id: 1, host: 'shard1-b.example.com:27018' },
    { _id: 2, host: 'shard1-c.example.com:27018', arbiterOnly: true },
  ],
});

// 3. Start mongos instances
// mongos --configdb
configRS/config1.example.com:27019,config2.example.com:27019,config3.example.com:270
```

```
19

// 4. Add shards to the cluster (from mongos)
sh.addShard('shard1RS/shard1-a.example.com:27018,shard1-b.example.com:27018');
sh.addShard('shard2RS/shard2-a.example.com:27018,shard2-b.example.com:27018');
sh.addShard('shard3RS/shard3-a.example.com:27018,shard3-b.example.com:27018');

// 5. Enable sharding for a database
sh.enableSharding('myDatabase');

// 6. Create indexes for shard key fields before sharding the collection
db.products.createIndex({ category: 1, product_id: 1 });

// 7. Shard the collection
sh.shardCollection(
  'myDatabase.products',
  { category: 1, product_id: 1 } // Shard key
);
```

3. Data Distribution

MongoDB distributes data in chunks (default 64MB). Chunks are defined by ranges of the shard key and are automatically balanced across shards.

Chunk Distribution Example:

```
Shard 1: { category: "electronics", product_id: MinKey } → { category:
"electronics", product_id: 50000 }
Shard 2: { category: "electronics", product_id: 50001 } → { category: "home",
product_id: 10000 }
Shard 3: { category: "home", product_id: 10001 } → { category: MaxKey, product_id:
MaxKey }
```

MongoDB automatically balances chunks between shards to ensure even data distribution.

4. Shard Key Selection

The shard key is the most critical decision in a sharded deployment as it determines:

- How data is distributed across shards
- Which queries can be targeted to specific shards vs. broadcast to all shards
- The cluster's ability to scale writes evenly

Key Considerations for Shard Key Selection:

1. **High Cardinality**
    - Many possible unique values
    - Allows for better distribution of data
    - Example: User IDs are better than country codes

```
Good: { user_id: 1 }         // Millions of possible values
Poor: { country_code: 1 }    // Only ~200 possible values
```

2. **Low Frequency (Even Distribution)**
    - No single value should appear much more frequently than others
    - Prevents "hot spots" on specific shards

```
Good: { order_id: 1 }        // Evenly distributed
Poor: { status: 1 }          // "active" might be 90% of documents
```

3. **Non-Monotonically Increasing**
    - Avoid keys that increase regularly over time
    - Prevents new writes from all going to the same shard

```
Bad: { timestamp: 1 }         // Always increasing, creating hot spots
Bad: { order_id: 1 }          // If auto-incrementing

Better: { timestamp % 100: 1, timestamp: 1 }  // Modulo to distribute
Better: { UUID: 1 }           // Random distribution
```

4. **Query Patterns**
    - Include fields frequently used in queries
    - Enables targeted queries (sent to specific shards)
    - Consider common query patterns

```
// If many queries filter by user_id:
Good: { user_id: 1, timestamp: 1 }

// Queries like: db.orders.find({ user_id: 123 })
// Can be routed to specific shards
```

5. **Consider Compound Shard Keys**
    - Combine high-cardinality with fields used in queries
    - Provide better control over data distribution

```
// Compound key examples:
{ region: 1, user_id: 1 }
{ app_id: 1, timestamp: 1 }
{ tenant_id: 1, entity_id: 1 }
```

Real-World Shard Key Examples:

1. **E-commerce Orders Collection:**

```
// Good shard key for orders
{ customer_id: 1, order_date: 1 }

// Rationale:
// - customer_id provides good cardinality
// - Many queries are by customer_id
// - order_date prevents concentration of a single customer's recent orders
```

2. **IoT Sensor Readings:**

```
// Good shard key for sensor readings
{ sensor_id: 1, timestamp: 1 }

// Rationale:
// - Evenly distributes data across many sensors
// - Most queries filter by both sensor_id and time range
// - Natural partitioning for time-series data
```

3. **Multi-tenant Application:**

```
// Good shard key for multi-tenant data
{ tenant_id: 1, entity_id: 1 }

// Rationale:
// - tenant_id provides isolation between tenants
// - entity_id distributes data within each tenant
// - Most queries are scoped to a specific tenant
```

5. Operational Considerations

Monitoring:

- Track chunk distribution
- Watch for jumbo chunks (too large to migrate)
- Monitor migration processes
- Track query patterns for targeted vs. scatter-gather operations

Commands for monitoring:

```
// Check sharding status
db.adminCommand({ listShards: 1 });

// Get detailed sharding information
```

```
sh.status();

// Check chunk distribution
db.chunks.aggregate([
  { $group: { _id: '$shard', count: { $sum: 1 } } },
  { $sort: { count: -1 } },
]);

// Analyze query performance
db.products
  .explain('executionStats')
  .find({ category: 'electronics', price: { $gt: 100 } });
```

Balancing:

- MongoDB automatically balances chunks between shards
- Can be disabled during peak hours: `sh.setBalancerState(false)`
- Can be configured to run during specific timeframes

Scaling:

- Add new shards: `sh.addShard("newShardRS/host1:27018,host2:27018")`
- Adding shards redistributes chunks automatically
- Pre-splitting for known distribution:

```
// Pre-split chunks for a new collection
for (let i = 1; i <= 10; i++) {
  sh.splitAt('myDatabase.products', {
    category: `category${i}`,
    product_id: MinKey,
  });
}
```

Backup and Recovery:

- Back up each shard separately
- Config server backup is critical
- Consider tools like MongoDB Atlas for managed backups

6. Zone-Based Sharding

For geographic distribution or hardware tiering:

```
// Define zones
sh.addShardToZone('shard1', 'us-east');
sh.addShardToZone('shard2', 'us-west');
sh.addShardToZone('shard3', 'eu-central');

// Define zone ranges
sh.updateZoneKeyRange(
```

```
  'myDatabase.customers',
  { region: 'US-East', _id: MinKey },
  { region: 'US-East', _id: MaxKey },
  'us-east'
);

sh.updateZoneKeyRange(
  'myDatabase.customers',
  { region: 'US-West', _id: MinKey },
  { region: 'US-West', _id: MaxKey },
  'us-west'
);

sh.updateZoneKeyRange(
  'myDatabase.customers',
  { region: 'Europe', _id: MinKey },
  { region: 'Europe', _id: MaxKey },
  'eu-central'
);
```

7. Common Pitfalls and Solutions

1. Poor Shard Key Choice:

- Symptom: Uneven data distribution, "hot" shards
- Solution: For new collections, choose better shard keys; for existing collections, consider using a different collection with a better key

2. Jumbo Chunks:

- Symptom: Chunks that can't be migrated due to size
- Solution: Pre-split chunks, refine shard key choice, handle high-cardinality fields

3. Inefficient Queries:

- Symptom: Scatter-gather queries across all shards
- Solution: Include shard key fields in queries, create appropriate indexes, redesign query patterns

4. Slow Chunk Migrations:

- Symptom: Rebalancing takes too long
- Solution: Schedule migrations during off-peak hours, adjust migration settings, ensure network bandwidth between shards

5. Config Server Bottlenecks:

- Symptom: Slow metadata operations
- Solution: Ensure config servers have adequate resources, especially SSD storage and sufficient RAM

Summary of Sharded MongoDB Deployment Best Practices

1. Choose shard keys with high cardinality, low frequency, and aligned with query patterns
2. Use compound shard keys to combine distribution quality with query targeting

3. Pre-split chunks for known data distributions
4. Monitor chunk distribution and migration processes
5. Implement appropriate indexes beyond the shard key
6. Consider zone sharding for geo-distribution or tiered storage
7. Regularly back up config servers and each shard
8. Scale by adding shards rather than increasing individual shard size

By carefully designing your sharded MongoDB deployment with these considerations in mind, you can build a highly scalable system capable of handling high-volume applications with predictable performance.

4.2 Key-Value Stores

Key-value stores are the simplest form of NoSQL databases, offering high performance and scalability for applications that access data primarily through a single key.

Fundamental Concepts

1. **Key-Value Pairs**: Data is stored as a collection of key-value pairs, similar to a hash table or dictionary
2. **Simple API**: Basic operations include GET, PUT, DELETE by key
3. **High Performance**: Optimized for simple lookups and writes
4. **Scalability**: Easy to distribute across multiple nodes
5. **Schema-less**: Values can be of any format (strings, numbers, JSON, binary data)

Redis

Redis is an open-source, in-memory key-value data store known for its exceptional performance, rich data structures, and versatility.

Key Features:

- In-memory storage with persistence options
- Rich data structures (strings, lists, sets, sorted sets, hashes, streams, etc.)
- Pub/sub messaging capabilities
- Scripting with Lua
- Atomic operations
- Transactions
- Support for cluster mode and replication

Basic Operations:

```
# String operations
SET user:1001 "John Smith"
GET user:1001
# Returns: "John Smith"

SET counter 100
INCR counter
# Returns: 101

# Key expiration
SET session:abc123 "user_data" EX 3600  # Expires in 1 hour
```

```
TTL session:abc123
# Returns: 3600 (seconds remaining)

# Hash operations
HSET user:1001 name "John Smith" email "john@example.com" age 35
HGET user:1001 email
# Returns: "john@example.com"
HGETALL user:1001
# Returns: name "John Smith" email "john@example.com" age "35"

# List operations
LPUSH notifications:1001 "New message from Jane"
LPUSH notifications:1001 "Payment received"
LRANGE notifications:1001 0 -1
# Returns: 1) "Payment received" 2) "New message from Jane"

# Set operations
SADD tags:post:1001 "redis" "database" "nosql"
SADD tags:post:1002 "redis" "cache" "performance"
SINTER tags:post:1001 tags:post:1002
# Returns: "redis"

# Sorted set operations
ZADD leaderboard 1000 "player1"
ZADD leaderboard 2500 "player2"
ZADD leaderboard 1800 "player3"
ZREVRANGE leaderboard 0 2 WITHSCORES
# Returns: 1) "player2" 2) "2500" 3) "player3" 4) "1800" 5) "player1" 6) "1000"
```

Data Persistence in Redis:

Redis offers several options for persistence:

1. **RDB (Redis Database)**: Point-in-time snapshots

```
# In redis.conf
save 900 1        # Snapshot after 900 seconds if at least 1 key changed
save 300 10       # Snapshot after 300 seconds if at least 10 keys changed
save 60 10000     # Snapshot after 60 seconds if at least 10000 keys changed
```

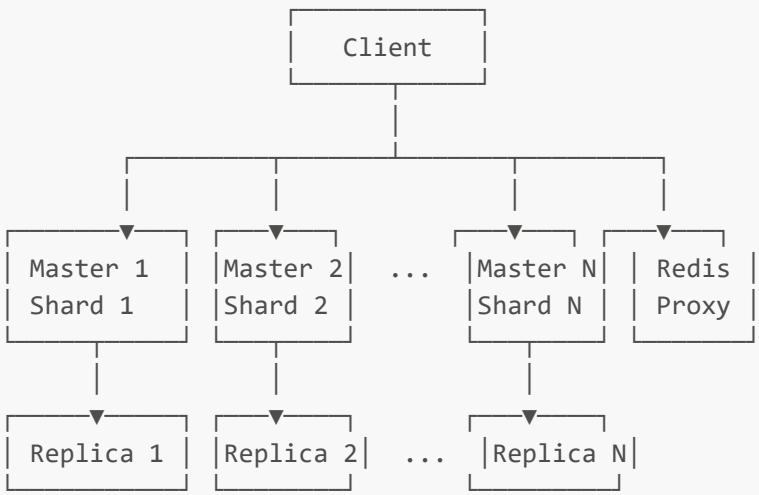
2. **AOF (Append Only File)**: Logs every write operation

```
# In redis.conf
appendonly yes
appendfsync everysec  # Sync once per second
```

3. **RDB + AOF**: Combined approach for better durability and performance

```
# In redis.conf
appendonly yes
appendfsync everysec
save 900 1
```

Redis Cluster Architecture:



Common Redis Use Cases:

1. **Caching:**

```
import redis
import json

r = redis.Redis(host='localhost', port=6379, db=0)

def get_user(user_id):
    # Try to get from cache
    cached_user = r.get(f"user:{user_id}")

    if cached_user:
        return json.loads(cached_user)

    # If not in cache, get from database
    user = database.query(f"SELECT * FROM users WHERE id = {user_id}")

    # Cache for 30 minutes
    r.setex(f"user:{user_id}", 1800, json.dumps(user))

    return user
```

2. **Session Store:**

```
def save_session(session_id, user_data, ttl=3600):
    r.hmset(f"session:{session_id}", user_data)
    r.expire(f"session:{session_id}", ttl)

def get_session(session_id):
    session_data = r.hgetall(f"session:{session_id}")
    if not session_data:
        return None
    return session_data

def extend_session(session_id, ttl=3600):
    r.expire(f"session:{session_id}", ttl)
```

3. **Rate Limiting:**

```
def is_rate_limited(user_id, limit=100, window=3600):
    key = f"ratelimit:{user_id}"
    current = r.get(key)

    if not current:
        r.setex(key, window, 1)
        return False

    if int(current) >= limit:
        return True

    r.incr(key)
    return False
```

4. **Leaderboards:**

```
def update_score(user_id, score):
    r.zadd("leaderboard", {user_id: score})

def get_top_players(count=10):
    return r.zrevrange("leaderboard", 0, count-1, withscores=True)

def get_rank(user_id):
    return r.zrevrank("leaderboard", user_id)
```

5. **Pub/Sub Messaging:**

```
# Publisher
def send_notification(channel, message):
    r.publish(channel, message)

# Subscriber
def listen_for_notifications(channel):
```

```
        pubsub = r.pubsub()
        pubsub.subscribe(channel)

        for message in pubsub.listen():
            if message['type'] == 'message':
                process_message(message['data'])
```

Amazon DynamoDB

DynamoDB is a fully managed, serverless NoSQL key-value and document database provided by AWS, designed for high performance at any scale.

Key Features:

- Automatic scaling
- Single-digit millisecond performance
- Built-in security and backup
- On-demand and provisioned capacity modes
- Global tables for multi-region deployment
- Point-in-time recovery
- Auto-expiring items

Data Model:

DynamoDB organizes data in tables with items (rows) and attributes (columns):

- **Partition Key (Required)**: Distributes data across partitions
- **Sort Key (Optional)**: Sorts items within a partition
- **Attributes**: Any number of attributes per item

Basic Operations (AWS SDK for JavaScript):

```
// Import AWS SDK
const AWS = require('aws-sdk');
const dynamoDB = new AWS.DynamoDB.DocumentClient();

// Creating an item
const createUser = async (user) => {
  const params = {
    TableName: 'Users',
    Item: {
      userId: user.id, // Partition key
      email: user.email,
      name: user.name,
      createdAt: Date.now(),
    },
  };

  await dynamoDB.put(params).promise();
};

// Reading an item
```

```
const getUser = async (userId) => {
  const params = {
    TableName: 'Users',
    Key: {
      userId: userId,
    },
  };

  const result = await dynamoDB.get(params).promise();
  return result.Item;
};

// Updating an item
const updateUser = async (userId, updates) => {
  // Build update expression and attribute values
  let updateExpression = 'SET ';
  let expressionAttributeValues = {};

  Object.keys(updates).forEach((key, index) => {
    const valueKey = `:val${index}`;
    updateExpression += `${key} = ${valueKey},`;
    expressionAttributeValues[valueKey] = updates[key];
  });

  // Remove the trailing comma
  updateExpression = updateExpression.slice(0, -1);

  const params = {
    TableName: 'Users',
    Key: {
      userId: userId,
    },
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ReturnValues: 'ALL_NEW',
  };

  const result = await dynamoDB.update(params).promise();
  return result.Attributes;
};

// Deleting an item
const deleteUser = async (userId) => {
  const params = {
    TableName: 'Users',
    Key: {
      userId: userId,
    },
  };

  await dynamoDB.delete(params).promise();
};
```

Advanced Query Patterns:

```
// Query items by partition key and sort key condition
const getUserOrders = async (userId, startDate, endDate) => {
  const params = {
    TableName: 'Orders',
    KeyConditionExpression:
      'userId = :uid AND orderDate BETWEEN :start AND :end',
    ExpressionAttributeValues: {
      ':uid': userId,
      ':start': startDate,
      ':end': endDate,
    },
  };

  const result = await dynamoDB.query(params).promise();
  return result.Items;
};

// Scan with filter (more expensive, avoid if possible)
const findInactiveUsers = async () => {
  const params = {
    TableName: 'Users',
    FilterExpression: 'lastLoginDate < :threshold',
    ExpressionAttributeValues: {
      ':threshold': Date.now() - 90 * 24 * 60 * 60 * 1000, // 90 days ago
    },
  };

  const result = await dynamoDB.scan(params).promise();
  return result.Items;
};
```

Secondary Indexes:

```
// Global Secondary Index (GSI)
// Define during table creation:
const createTableParams = {
  TableName: 'Users',
  KeySchema: [
    { AttributeName: 'userId', KeyType: 'HASH' }, // Partition key
  ],
  AttributeDefinitions: [
    { AttributeName: 'userId', AttributeType: 'S' },
    { AttributeName: 'email', AttributeType: 'S' },
  ],
  GlobalSecondaryIndexes: [
    {
      IndexName: 'EmailIndex',
      KeySchema: [{ AttributeName: 'email', KeyType: 'HASH' }],
      Projection: {
        ProjectionType: 'ALL',
      },
      ProvisionedThroughput: {
```

```
        ReadCapacityUnits: 5,
        WriteCapacityUnits: 5,
      },
    },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 5,
    WriteCapacityUnits: 5,
  },
};

// Query using a GSI
const getUserByEmail = async (email) => {
  const params = {
    TableName: 'Users',
    IndexName: 'EmailIndex',
    KeyConditionExpression: 'email = :email',
    ExpressionAttributeValues: {
      ':email': email,
    },
  };

  const result = await dynamoDB.query(params).promise();
  return result.Items[0]; // Assuming email is unique
};
```

DynamoDB Streams and Triggers:

DynamoDB Streams capture item-level changes in a table, which can trigger Lambda functions:

```
// Lambda function triggered by DynamoDB Stream
exports.handler = async (event) => {
  for (const record of event.Records) {
    if (record.eventName === 'INSERT') {
      const newItem = AWS.DynamoDB.Converter.unmarshall(
        record.dynamodb.NewImage
      );

      // Example: Send welcome email when new user is created
      if (record.eventSourceARN.includes('Users')) {
        await sendWelcomeEmail(newItem.email, newItem.name);
      }
    }

    if (record.eventName === 'MODIFY') {
      const oldItem = AWS.DynamoDB.Converter.unmarshall(
        record.dynamodb.OldImage
      );
      const newItem = AWS.DynamoDB.Converter.unmarshall(
        record.dynamodb.NewImage
      );

      // Example: Track inventory changes
```

```
      if (
        record.eventSourceARN.includes('Products') &&
        oldItem.stockCount !== newItem.stockCount
      ) {
        await logInventoryChange(
          newItem.productId,
          oldItem.stockCount,
          newItem.stockCount
        );
      }
    }
  }

  return { status: 'success' };
};
```

Key-Value Store Design Patterns

1. **Composite Keys:**
   - Use delimiters to create hierarchical keys
   - Example: `user:1001:profile`, `user:1001:orders`, `user:1001:orders:2023-06`

```
// Redis example
HSET user:1001:profile name "John Smith" email "john@example.com"
LPUSH user:1001:orders "order:12345" "order:12346" "order:12347"
```

2. **Time-Based Keys:**
   - Include timestamps in keys for natural expiration ordering
   - Example: `session:1623876542:abc123`

```
// DynamoDB example
const createSession = async (userId, sessionData) => {
  const timestamp = Date.now();
  const sessionId = generateSessionId();

  await dynamoDB
    .put({
      TableName: 'Sessions',
      Item: {
        userId: userId,
        sessionKey: `session:${timestamp}:${sessionId}`,
        data: sessionData,
        expiresAt: timestamp + 24 * 60 * 60 * 1000, // 24 hours
      },
    })
    .promise();
};
```

3. **Denormalization and Aggregation:**
   - Store precomputed aggregates for quick access
   - Example: Store daily, weekly, monthly counts separately

```
// Redis example
INCR pageviews:daily:2023-06-15
INCR pageviews:monthly:2023-06
EXPIRE pageviews:daily:2023-06-15 86400  // Expire after 1 day
```

4. **Caching Patterns:**
   - Cache-Aside: Application checks cache first, then database
   - Write-Through: Update cache and database simultaneously
   - Write-Behind: Update cache immediately, database asynchronously

```
// Cache-Aside pattern with Redis
async function getProduct(productId) {
  const cacheKey = `product:${productId}`;

  // Try cache first
  const cachedProduct = await redisClient.get(cacheKey);
  if (cachedProduct) {
    return JSON.parse(cachedProduct);
  }

  // Cache miss - get from database
  const product = await database.getProduct(productId);

  // Update cache with TTL
  await redisClient.setex(cacheKey, 3600, JSON.stringify(product));

  return product;
}
```

5. **Distributed Locking:**
   - Use atomic operations to implement locks
   - Set expiration to prevent deadlocks

```
// Redis distributed lock implementation
async function acquireLock(resource, ttlMs = 30000) {
  const token = generateUniqueToken();
  const acquired = await redisClient.set(
    `lock:${resource}`,
    token,
    'PX',
    ttlMs,
    'NX' // Only set if key doesn't exist
  );
```

```
  return acquired ? token : null;
}

async function releaseLock(resource, token) {
  // Only release if we own the lock
  const script = `
    if redis.call("get", KEYS[1]) == ARGV[1] then
      return redis.call("del", KEYS[1])
    else
      return 0
    end
  `;

  await redisClient.eval(script, 1, `lock:${resource}`, token);
}
```

Interview Questions

Q: Compare and contrast Redis and DynamoDB, discussing scenarios where you would choose one over the other.

A: Redis and DynamoDB are both powerful NoSQL databases in the key-value store category, but they have fundamentally different architectures, capabilities, and use cases. Let me break down their key differences and explain when you might choose one over the other.

Architectural Differences

Redis

- **In-memory with persistence options**: Primary storage is RAM with optional disk persistence
- **Single-instance or cluster deployment**: Can be deployed as a single instance or clustered
- **Rich data structures**: Supports strings, lists, sets, sorted sets, hashes, streams, bitmaps, and more
- **Self-managed or managed service**: Available as open-source software or as managed services (Redis Cloud, Amazon ElastiCache, etc.)
- **Server scaling**: Requires manual scaling or configuration of clusters

DynamoDB

- **Fully managed, serverless database**: No servers to provision or manage
- **Disk-based with caching**: Data stored on SSDs with optional caching via DAX
- **Automatic scaling**: Automatically scales throughput capacity up or down
- **AWS-native service**: Deeply integrated with AWS ecosystem
- **Global tables**: Built-in multi-region replication
- **Simple data model**: Key-value pairs with optional document structure (attributes)

Performance Characteristics

Redis

- **Sub-millisecond latency**: Extremely low latency due to in-memory operation

- **High throughput for simple operations**: Excels at simple key-value operations
- **Memory-bound scalability**: Limited by available memory
- **Single-threaded core**: Processes commands one at a time (though I/O is multithreaded in newer versions)
- **Potential for data loss**: Depending on persistence configuration

DynamoDB

- **Single-digit millisecond latency**: Fast but not as fast as Redis
- **Virtually unlimited throughput**: Can scale to millions of requests per second
- **Virtually unlimited storage**: No practical storage limits
- **Consistent performance at scale**: Performance remains predictable regardless of size
- **Strong durability guarantees**: Multiple copies of data across different availability zones

Feature Comparison

Feature	Redis	DynamoDB
Data Types	Rich (strings, lists, sets, sorted sets, hashes, streams, etc.)	Simple (scalar values, sets, lists, maps)
Query Capabilities	Limited to key-based access; secondary indexes via sorted sets	Primary key access plus secondary indexes
Transactions	Supports ACID transactions with optimistic locking	Supports ACID transactions with limitations
TTL (Time to Live)	Key-level expiration with precise timing	Item-level expiration with 48-hour precision
Pub/Sub Messaging	Robust publish/subscribe mechanism	None (use SNS/SQS instead)
Lua Scripting	Supports server-side Lua scripts	No server-side scripting
Backup & Recovery	Point-in-time recovery depends on configuration	Built-in point-in-time recovery
Security Model	Basic authentication; network security	IAM integration, VPC endpoints, encryption
Cost Model	Based on instance size	Based on provisioned capacity and storage

When to Choose Redis

1. **Caching Layer**

   Redis excels as a caching solution due to its extremely low latency and in-memory nature.

```
# Redis caching example
def get_user_profile(user_id):
    cache_key = f"user:{user_id}"
```

```
    # Try cache first
    cached_data = redis_client.get(cache_key)
    if cached_data:
        return json.loads(cached_data)

    # Cache miss - query database
    user_data = database.query(f"SELECT * FROM users WHERE id = {user_id}")

    # Store in cache for 15 minutes
    redis_client.setex(cache_key, 900, json.dumps(user_data))

    return user_data
```

This pattern provides sub-millisecond response times for frequently accessed data.

2. **Real-time Analytics**

   Redis's rich data structures make it ideal for real-time counters, leaderboards, and analytics.

```
# Increment page views counter
redis_client.incr(f"pageviews:{page_id}")

# Get top 10 pages by views
redis_client.zincrby("page_leaderboard", 1, page_id)
top_pages = redis_client.zrevrange("page_leaderboard", 0, 9, withscores=True)
```

   The sorted set data structure makes leaderboards trivial to implement with Redis.

3. **Message Broker/Pub-Sub**

   Redis provides a lightweight pub-sub messaging system for real-time communication.

```
# Publisher
redis_client.publish("notifications", json.dumps({
    "user_id": 1001,
    "message": "New message received",
    "timestamp": time.time()
}))

# Subscriber
def handle_message(message):
    data = json.loads(message["data"])
    notify_user(data["user_id"], data["message"])

pubsub = redis_client.pubsub()
pubsub.subscribe(**{"notifications": handle_message})
pubsub.run_in_thread()
```

   This enables real-time notifications, chat systems, and event propagation.

4. **Rate Limiting and Throttling**

   Redis's atomic operations make it perfect for implementing rate limiters.

```
def is_rate_limited(user_id, max_requests=100, window_seconds=3600):
    key = f"ratelimit:{user_id}"
    current = redis_client.get(key)

    if not current:
        redis_client.setex(key, window_seconds, 1)
        return False

    if int(current) >= max_requests:
        return True

    redis_client.incr(key)
    return False
```

   This pattern protects APIs and services from abuse with minimal overhead.

5. **Session Storage**

   Redis's speed and TTL features make it ideal for managing user sessions.

```
def create_session(user_id, session_data):
    session_id = str(uuid.uuid4())
    key = f"session:{session_id}"

    # Store session with 1-hour expiration
    redis_client.hmset(key, session_data)
    redis_client.expire(key, 3600)

    return session_id
```

   The automatic expiration ensures sessions are cleaned up without manual intervention.

When to Choose DynamoDB

1. **Scalable Web Applications**

   DynamoDB's automatic scaling makes it ideal for applications with unpredictable or growing traffic.

```
// Creating a user profile that can scale to millions of users
const createUser = async (userData) => {
  const params = {
    TableName: 'Users',
    Item: {
      userId: userData.id,
      email: userData.email,
      name: userData.name,
```

```
      createdAt: Date.now(),
    },
  };

  await dynamoDB.put(params).promise();
};
```

The application can scale from 10 to 10 million users without database rearchitecture.

2. **High-Throughput OLTP Workloads**

   DynamoDB can handle massive write throughput for transactional systems.

```
// E-commerce order processing
const createOrder = async (orderData) => {
  const params = {
    TableName: 'Orders',
    Item: {
      orderId: generateOrderId(),
      userId: orderData.userId,
      orderDate: Date.now(),
      items: orderData.items,
      total: orderData.total,
      status: 'PENDING',
    },
  };

  await dynamoDB.put(params).promise();
};
```

   This can scale to handle holiday shopping peaks without performance degradation.

3. **Microservices Persistence Layer**

   DynamoDB's managed nature makes it ideal for microservices architectures.

```
// User service data store
const userService = {
  async getUser(userId) {
    const result = await dynamoDB
      .get({
        TableName: 'Users',
        Key: { userId },
      })
      .promise();

    return result.Item;
  },

  async updateUser(userId, updates) {
    // Build update expression
```

```
    // ...

    return dynamoDB
      .update({
        TableName: 'Users',
        Key: { userId },
        UpdateExpression: expression,
        ExpressionAttributeValues: values,
      })
      .promise();
  },
};
```

Each microservice can have its dedicated DynamoDB table without operational overhead.

4. **IoT Data Collection**

   DynamoDB's scalability makes it suitable for IoT applications generating massive data volumes.

```
// Storing IoT device readings
const storeDeviceReading = async (deviceId, reading) => {
  await dynamoDB
    .put({
      TableName: 'DeviceReadings',
      Item: {
        deviceId,
        timestamp: Date.now(),
        temperature: reading.temperature,
        humidity: reading.humidity,
        batteryLevel: reading.batteryLevel,
      },
    })
    .promise();
};
```

   This can handle millions of devices sending readings without performance issues.

5. **Event-Driven Architectures**

   DynamoDB Streams enable event-driven processing of data changes.

```
// Lambda triggered by DynamoDB Stream when order status changes
exports.handler = async (event) => {
  for (const record of event.Records) {
    if (record.eventName === 'MODIFY') {
      const newImage = AWS.DynamoDB.Converter.unmarshall(
        record.dynamodb.NewImage
      );
      const oldImage = AWS.DynamoDB.Converter.unmarshall(
        record.dynamodb.OldImage
      );
```

```
        if (
          oldImage.status !== newImage.status &&
          newImage.status === 'SHIPPED'
        ) {
          // Send notification to customer
          await sendShippingNotification(newImage.userId, newImage.orderId);
        }
      }
    }
  };
```

This creates a reactive system that responds automatically to data changes.

Hybrid Approaches

In many real-world systems, both Redis and DynamoDB are used together, leveraging their complementary strengths:

```
// Using Redis as a cache in front of DynamoDB
async function getProduct(productId) {
  // First, try Redis cache
  const cachedProduct = await redisClient.get(`product:${productId}`);
  if (cachedProduct) {
    return JSON.parse(cachedProduct);
  }

  // Cache miss - get from DynamoDB
  const result = await dynamoDB
    .get({
      TableName: 'Products',
      Key: { productId },
    })
    .promise();

  if (!result.Item) {
    return null;
  }

  // Update cache with 15-minute TTL
  await redisClient.setex(
    `product:${productId}`,
    900,
    JSON.stringify(result.Item)
  );

  return result.Item;
}
```

Cost Considerations

Redis Cost Factors:

- Instance size and number
- Memory usage
- Replication configuration
- Managed service premiums

DynamoDB Cost Factors:

- Read/write capacity units (provisioned or on-demand)
- Storage usage
- Data transfer
- Additional features (backup, global tables, streams)

Decision Framework

When choosing between Redis and DynamoDB, consider these factors:

1. **Performance Requirements**
    - Need sub-millisecond latency? → Redis
    - Need predictable performance at any scale? → DynamoDB
2. **Operational Considerations**
    - Want zero operational overhead? → DynamoDB
    - Need complete control over configuration? → Redis
    - Want automatic scaling? → DynamoDB
3. **Data Structure Requirements**
    - Need complex data structures (sorted sets, lists, etc.)? → Redis
    - Simple key-value or document access patterns? → Either works, but DynamoDB may be simpler
4. **Durability Requirements**
    - Cannot afford any data loss? → DynamoDB
    - Can tolerate some data loss for performance? → Redis
5. **Integration Requirements**
    - Deep AWS integration needed? → DynamoDB
    - Need pub/sub capabilities? → Redis
    - Need triggers on data changes? → Either (Redis Keyspace Notifications or DynamoDB Streams)
6. **Development Experience**
    - Familiar with AWS services? → DynamoDB
    - Experience with Redis commands? → Redis
    - Need simple API? → DynamoDB's SDK may be simpler for basic operations

Summary

Redis and DynamoDB represent two different philosophies in database design:

- **Redis** offers exceptional performance and flexibility through specialized in-memory data structures and operations, but requires more operational attention and has inherent capacity limitations.
- **DynamoDB** provides worry-free scalability and durability in a managed service with predictable performance, but with a more limited feature set and slightly higher latency.

The choice between them is not mutually exclusive. Many modern applications use both: Redis for caching, real-time features, and specialized data structures, and DynamoDB for durable storage of application data that needs to scale without operational overhead.

Q: Describe how you would implement a rate limiting system using Redis. What design considerations would you take into account?

A: A rate limiting system restricts the number of requests or actions that can be performed within a given time window. Implementing an effective rate limiter with Redis requires careful consideration of algorithms, performance, and failure scenarios. I'll explain how to build a robust rate limiting system using Redis and discuss important design considerations.

Rate Limiting Fundamentals

Before diving into implementation, let's understand the key components of a rate limiting system:

1. **Resource Identifier**: What is being rate limited (user ID, IP address, API key)
2. **Rate Limit Policy**: Maximum number of operations allowed in a time window
3. **Time Window**: Period over which the rate is measured (second, minute, hour)
4. **Counter**: Tracking mechanism for operations within the window
5. **Response Strategy**: What happens when a limit is reached (block, queue, throttle)

Rate Limiting Algorithms

1. Fixed Window Algorithm

The simplest approach, counting events in fixed time intervals.

```
Time: |--Window 1--|--Window 2--|--Window 3--|
      ↑            ↑            ↑
    Counter      Counter      Counter
    resets       resets       resets
```

Redis implementation:

```python
def fixed_window_rate_limit(user_id, action, limit=10, window_seconds=60):
    """
    Fixed window rate limiting.

    Args:
        user_id: The user to rate limit
        action: The action being limited (e.g. 'login', 'api_call')
        limit: Maximum number of actions allowed in the window
        window_seconds: The time window in seconds
```

```
    Returns:
        tuple: (allowed (bool), current_count (int), ttl (int))
    """
    # Create a key that includes the current time window
    current_window = int(time.time() / window_seconds)
    key = f"ratelimit:{user_id}:{action}:{current_window}"

    # Increment the counter for the current window
    current_count = redis_client.incr(key)

    # Set expiration if this is the first increment in this window
    if current_count == 1:
        redis_client.expire(key, window_seconds)

    # Get the remaining TTL
    ttl = redis_client.ttl(key)

    # Check if the limit has been exceeded
    allowed = current_count <= limit

    return (allowed, current_count, ttl)
```

Pros:

- Simple to understand and implement
- Low memory usage (one key per user/action/window)

Cons:

- Boundary problem: A user could make `limit` requests at the end of one window and `limit` more at the start of the next window, effectively doubling the rate

2. Sliding Window Algorithm

A more refined approach that smooths out the boundary problem by overlapping windows.

Redis implementation:

```
def sliding_window_rate_limit(user_id, action, limit=10, window_seconds=60):
    """
    Sliding window rate limiting using sorted sets.

    Args:
        user_id: The user to rate limit
        action: The action being limited
        limit: Maximum number of actions allowed in the window
        window_seconds: The time window in seconds

    Returns:
        tuple: (allowed (bool), current_count (int), retry_after (int))
    """
    key = f"ratelimit:sliding:{user_id}:{action}"
```

```
    now = time.time()
    window_start = now - window_seconds

    # Add the current timestamp to the sorted set with the score as the timestamp
    pipeline = redis_client.pipeline()
    pipeline.zadd(key, {str(now): now})

    # Remove elements outside the current window
    pipeline.zremrangebyscore(key, 0, window_start)

    # Count the number of elements in the current window
    pipeline.zcard(key)

    # Set the expiration on the key
    pipeline.expire(key, window_seconds)

    # Execute the commands
    _, _, current_count, _ = pipeline.execute()

    # Check if the limit has been exceeded
    allowed = current_count <= limit

    # Calculate when the user can retry
    retry_after = 0
    if not allowed:
        # Get the oldest timestamp in the set
        oldest = redis_client.zrange(key, 0, 0, withscores=True)
        if oldest:
            retry_after = int(oldest[0][1] + window_seconds - now)

    return (allowed, current_count, retry_after)
```

Pros:

- More accurate rate limiting, without the boundary problem
- Provides a precise time when rate limit will reset

Cons:

- Higher memory usage due to storing individual timestamps
- More complex Redis operations required

3. Token Bucket Algorithm

Models rate limiting as a bucket that continuously refills with tokens.

```
      ↓ Tokens added at a constant rate
      ↓
      ▼
+------------+
|   Bucket   |
+------------+
```

```
      ▲
      | Tokens removed when requests are made
```

Redis implementation using Lua script:

```
# Lua script for token bucket algorithm
token_bucket_script = """
local key = KEYS[1]
local max_tokens = tonumber(ARGV[1])
local refill_rate = tonumber(ARGV[2])  -- tokens per second
local now = tonumber(ARGV[3])
local requested_tokens = tonumber(ARGV[4])

-- Get the current token count and last refill time
local current = redis.call('hmget', key, 'tokens', 'last_refill')
local tokens = tonumber(current[1] or max_tokens)
local last_refill = tonumber(current[2] or 0)

-- Calculate the new token count based on the time passed
local time_passed = math.max(0, now - last_refill)
local new_tokens = math.min(max_tokens, tokens + (time_passed * refill_rate))

-- If we have enough tokens, consume them
local allowed = new_tokens >= requested_tokens
local remaining = new_tokens

if allowed then
    remaining = new_tokens - requested_tokens
    redis.call('hmset', key, 'tokens', remaining, 'last_refill', now)
else
    -- Update the token count without consuming
    redis.call('hmset', key, 'tokens', new_tokens, 'last_refill', now)
end

-- Set expiration for the key
redis.call('expire', key, 60)

-- Return the result
return {allowed and 1 or 0, remaining, max_tokens}
"""

# Register the script with Redis
token_bucket_sha = redis_client.script_load(token_bucket_script)

def token_bucket_rate_limit(user_id, action, tokens=1, max_tokens=10,
refill_rate=0.16):
    """
    Token bucket rate limiting.

    Args:
        user_id: The user to rate limit
        action: The action being limited
        tokens: Number of tokens to consume for this action
```

```
        max_tokens: Maximum bucket capacity
        refill_rate: Tokens added per second

    Returns:
        tuple: (allowed (bool), remaining_tokens (int), total_tokens (int))
    """
    key = f"ratelimit:bucket:{user_id}:{action}"
    now = time.time()

    # Execute the Lua script
    result = redis_client.evalsha(
        token_bucket_sha,
        1,  # Number of keys
        key,  # The key
        max_tokens,  # Maximum tokens
        refill_rate,  # Refill rate
        now,  # Current time
        tokens  # Tokens to consume
    )

    return (bool(result[0]), result[1], result[2])
```

Pros:

- Supports burst allowance (bucket can fill up during idle periods)
- Handles different costs for different actions (consume more tokens for expensive operations)
- Smooth rate limiting over time

Cons:

- More complex to implement and understand
- Slightly higher computational overhead

4. Sliding Window with Counter

A hybrid approach that approximates a sliding window using counters for fixed windows.

Redis implementation:

```
def sliding_window_counter(user_id, action, limit=10, window_seconds=60):
    """
    Sliding window counter implementation.

    Args:
        user_id: The user to rate limit
        action: The action being limited
        limit: Maximum number of actions allowed in the window
        window_seconds: The time window in seconds

    Returns:
        tuple: (allowed (bool), current_count (float), reset_after (int))
    """
    # Calculate the current and previous windows
```

```
    now = time.time()
    current_window_start = int(now / window_seconds) * window_seconds
    previous_window_start = current_window_start - window_seconds

    # Calculate how far into the current window we are (0 to 1)
    position_in_window = (now - current_window_start) / window_seconds

    # Keys for the current and previous windows
    current_key = f"ratelimit:{user_id}:{action}:{current_window_start}"
    previous_key = f"ratelimit:{user_id}:{action}:{previous_window_start}"

    # Get the counts for the current and previous windows
    pipeline = redis_client.pipeline()
    pipeline.get(current_key)
    pipeline.get(previous_key)
    pipeline.incr(current_key)
    pipeline.expire(current_key, window_seconds * 2)  # Keep a bit longer for
overlap
    current_count, previous_count, new_count, _ = pipeline.execute()

    # Convert to integers, default to 0 if None
    current_count = int(current_count or 0)
    previous_count = int(previous_count or 0)

    # Calculate the weighted count based on position in the window
    weighted_count = current_count + previous_count * (1 - position_in_window)

    # Check if the weighted count exceeds the limit
    allowed = weighted_count <= limit

    # Calculate when the rate limit will reset
    reset_after = int(window_seconds - (now - current_window_start))

    return (allowed, weighted_count, reset_after)
```

Pros:

- Provides sliding window behavior with less memory than storing all timestamps
- More accurate than fixed window, with less overhead than full sliding window

Cons:

- Approximation rather than exact count
- More complex logic

Design Considerations

1. Performance Optimization

Redis operations should be optimized to minimize latency:

1. **Pipelining**: Use Redis pipelines to reduce network round trips

```
pipeline = redis_client.pipeline()
pipeline.incr(key)
pipeline.expire(key, window_seconds)
results = pipeline.execute()
```

2. **Lua Scripting**: Use Lua scripts to make operations atomic and reduce round trips

```
increment_and_check_script = """
local key = KEYS[1]
local limit = tonumber(ARGV[1])
local expire_time = tonumber(ARGV[2])

local count = redis.call('incr', key)

if count == 1 then
  redis.call('expire', key, expire_time)
end

return {count, count <= limit}
"""

# Register the script
script_sha = redis_client.script_load(increment_and_check_script)

# Execute the script
count, allowed = redis_client.evalsha(script_sha, 1, key, limit,
window_seconds)
```

3. **Key Design**: Optimize key patterns for sharding and memory usage

```
# Good key design for sharding
key = f"ratelimit:{action}:{user_id % 1000}:{user_id}"
```

2. Distributed Rate Limiting

In a distributed system, consider:

1. **Centralized Redis**: Use a shared Redis instance or cluster for all application nodes

```
App Server 1 --┐
               \
App Server 2 ----► Redis Cluster
               /
App Server 3 --┘
```

2. **Consistent Hashing**: If using multiple Redis instances, use consistent hashing for key distribution

```
def get_redis_node(key):
    nodes = ['redis1:6379', 'redis2:6379', 'redis3:6379']
    hash_value = hash(key)
    node_index = hash_value % len(nodes)
    return redis.Redis.from_url(f"redis://{nodes[node_index]}")

def rate_limit(user_id, action):
    key = f"ratelimit:{user_id}:{action}"
    redis_node = get_redis_node(key)
    # Continue with rate limiting using the selected node
```

3. **Local Caching**: Reduce Redis load with local in-memory caches for rejected requests

```
# Simple in-memory cache for rejected requests
rejected_cache = {}

def check_rate_limit(user_id, action):
    # Check local cache first
    cache_key = f"{user_id}:{action}"
    if cache_key in rejected_cache and time.time() < rejected_cache[cache_key]:
        return False

    # Check Redis rate limit
    allowed, _, reset_after = sliding_window_rate_limit(user_id, action)

    # If rejected, cache locally for a short time
    if not allowed:
        rejected_cache[cache_key] = time.time() + min(reset_after, 5)  # Max 5
seconds

    return allowed
```

3. Fault Tolerance

Ensure the rate limiter degrades gracefully during Redis failures:

1. **Circuit Breaker Pattern**: Allow requests if Redis is unavailable

```
def rate_limit_with_circuit_breaker(user_id, action):
    try:
        return rate_limit(user_id, action)
    except redis.RedisError:
        # Log the error
        logger.error("Redis error in rate limiter", exc_info=True)

        # Circuit breaker - allow the request if Redis is down
        return True
```

2. **Redis Sentinel or Cluster**: Use Redis high-availability features

```
# Redis Sentinel configuration
sentinel = redis.Sentinel([
    ('sentinel1', 26379),
    ('sentinel2', 26379),
    ('sentinel3', 26379)
])

# Get the master for writing
master = sentinel.master_for('mymaster')

# Use the master for rate limiting
master.incr(key)
```

3. **Multiple Redis Instances**: Implement rate limiting with redundancy

```
def redundant_rate_limit(user_id, action):
    primary_allowed = rate_limit(user_id, action, redis_primary)

    # If primary rejects, check secondary with higher limit as backup
    if not primary_allowed:
        return rate_limit(user_id, action, redis_secondary, higher_limit=True)

    return primary_allowed
```

4. Multi-Level Rate Limiting

Implement different tiers of rate limiting:

```
def multi_level_rate_limit(user_id, action, ip_address):
    # Global rate limit
    global_allowed = rate_limit('global', action, limit=10000, window_seconds=60)
    if not global_allowed:
        return False, 'global_limit_exceeded'

    # IP-based rate limit
    ip_allowed = rate_limit(ip_address, action, limit=100, window_seconds=60)
    if not ip_allowed:
        return False, 'ip_limit_exceeded'

    # User-specific rate limit
    user_allowed = rate_limit(user_id, action, limit=10, window_seconds=60)
    if not user_allowed:
        return False, 'user_limit_exceeded'

    # Resource-specific limit (e.g., for expensive operations)
    if action == 'expensive_api':
        action_allowed = token_bucket_rate_limit(user_id, action, tokens=5)
        if not action_allowed:
            return False, 'expensive_action_limit_exceeded'
```

```
    return True, 'allowed'
```

5. Dynamic Rate Limiting

Adjust rate limits based on system load or user behavior:

```
def dynamic_rate_limit(user_id, action):
    # Get current system load
    system_load = redis_client.get('system:load') or 1.0
    system_load = float(system_load)

    # Get user tier/reputation
    user_tier = get_user_tier(user_id)  # e.g., 'free', 'premium', 'enterprise'

    # Base limits by tier
    tier_limits = {
        'free': 10,
        'premium': 50,
        'enterprise': 200
    }

    # Adjust limit based on system load
    adjusted_limit = int(tier_limits[user_tier] / system_load)

    # Apply the adjusted rate limit
    return rate_limit(user_id, action, limit=adjusted_limit)
```

6. Client Response Strategy

Provide helpful information in API responses when rate limited:

```
def handle_request(user_id, action):
    allowed, count, retry_after = sliding_window_rate_limit(user_id, action)

    if not allowed:
        response = {
            'status': 'error',
            'code': 429,  # Too Many Requests
            'message': 'Rate limit exceeded',
            'retry_after': retry_after,
            'limit': 10,
            'current': count,
            'reset': int(time.time()) + retry_after
        }
        # Add standard headers
        headers = {
            'Retry-After': str(retry_after),
            'X-RateLimit-Limit': '10',
            'X-RateLimit-Remaining': '0',
```

```
            'X-RateLimit-Reset': str(int(time.time()) + retry_after)
        }
        return response, headers, 429

    # Process the request normally...
```

Complete Implementation Example

Here's a complete rate limiter service using Redis with the sliding window algorithm:

```
import time
import redis
import uuid

class RedisRateLimiter:
    """Redis-based rate limiter using sliding window algorithm."""

    def __init__(self, redis_url='redis://localhost:6379/0'):
        self.redis = redis.from_url(redis_url)

        # Lua script for atomic rate limiting operation
        self.limit_script = """
        local key = KEYS[1]
        local now = tonumber(ARGV[1])
        local window = tonumber(ARGV[2])
        local limit = tonumber(ARGV[3])

        -- Add the current request timestamp to the sorted set
        redis.call('ZADD', key, now, now .. ':' .. math.random())

        -- Remove elements outside the current window
        redis.call('ZREMRANGEBYSCORE', key, 0, now - window)

        -- Count the number of elements in the current window
        local count = redis.call('ZCARD', key)

        -- Set the expiration on the key
        redis.call('EXPIRE', key, window)

        -- Return the count and whether it's allowed
        return {count, count <= limit}
        """

        # Load the script into Redis
        self.limit_script_sha = self.redis.script_load(self.limit_script)

    def is_rate_limited(self, resource_id, action, limit=10, window_seconds=60):
        """
        Check if a resource has exceeded its rate limit.

        Args:
            resource_id: Identifier for the resource (user_id, IP, etc.)
            action: The action being rate limited
```

```
            limit: Maximum number of actions in the time window
            window_seconds: Time window in seconds

        Returns:
            tuple: (allowed (bool), current_count (int), retry_after (int or None))
        """
        key = f"ratelimit:{resource_id}:{action}"
        now = time.time()

        try:
            # Execute the rate limiting script
            count, allowed = self.redis.evalsha(
                self.limit_script_sha,
                1,  # number of keys
                key,
                now,
                window_seconds,
                limit
            )

            # Convert to appropriate types
            count = int(count)
            allowed = bool(allowed)

            # Calculate retry after time if not allowed
            retry_after = None
            if not allowed:
                oldest_timestamp = now
                try:
                    # Get the oldest request timestamp in the window
                    oldest = self.redis.zrange(key, 0, 0, withscores=True)
                    if oldest:
                        oldest_timestamp = oldest[0][1]

                    # Calculate when a spot will open up
                    retry_after = max(1, int(oldest_timestamp + window_seconds -
now))
                except:
                    # Default fallback if we can't determine the exact time
                    retry_after = window_seconds

            return (allowed, count, retry_after)

        except redis.RedisError as e:
            # Log the error
            print(f"Redis error in rate limiter: {e}")

            # Fail open - allow the request if Redis is down
            return (True, 0, None)

    def reset_limits(self, resource_id, action=None):
        """
        Reset rate limits for a resource.

        Args:
```

```
            resource_id: Identifier for the resource
            action: Specific action to reset, or None for all actions
        """
        try:
            if action:
                key = f"ratelimit:{resource_id}:{action}"
                self.redis.delete(key)
            else:
                # Find and delete all keys for this resource
                pattern = f"ratelimit:{resource_id}:*"
                keys = self.redis.keys(pattern)
                if keys:
                    self.redis.delete(*keys)

            return True
        except redis.RedisError as e:
            print(f"Error resetting rate limits: {e}")
            return False
```

Usage Examples

```
# Initialize the rate limiter
limiter = RedisRateLimiter(redis_url="redis://localhost:6379/0")

# API endpoint with rate limiting
def api_request_handler(user_id, action):
    # Check rate limit
    allowed, count, retry_after = limiter.is_rate_limited(user_id, action)

    if not allowed:
        response = {
            "status": "error",
            "message": "Rate limit exceeded",
            "current_count": count,
            "limit": 10,
            "retry_after": retry_after
        }
        return response, 429  # HTTP 429 Too Many Requests

    # Process the request normally
    result = process_request()

    # Return success response with rate limit information
    response = {
        "status": "success",
        "data": result,
        "rate_limit": {
            "current_count": count,
            "limit": 10,
            "remaining": 10 - count
        }
    }
    return response, 200
```

```
# Simulate multiple requests
for i in range(15):
    user_id = "user123"
    action = "api_call"
    allowed, count, retry_after = limiter.is_rate_limited(user_id, action)

    print(f"Request {i+1}: {'Allowed' if allowed else 'Blocked'}, Count: {count}")

    if not allowed:
        print(f"Rate limited! Retry after {retry_after} seconds")
        # In a real system, you might wait or stop here

    # Small delay to simulate request timing
    time.sleep(0.1)

# Admin functionality - reset a user's limits
limiter.reset_limits("user123")
```

Conclusion

An effective Redis-based rate limiting system balances accuracy, performance, and fault tolerance. The sliding window algorithm provides the best balance of accuracy and efficiency for most use cases, while the token bucket algorithm offers more flexibility for varying request costs and burst handling.

Key implementation considerations include:

- Using Lua scripts for atomic operations
- Optimizing key design for Redis performance
- Implementing proper error handling for Redis failures
- Providing clear feedback to clients when rate limits are exceeded
- Adjusting limits dynamically based on system conditions and user tiers

With these considerations in mind, Redis provides an excellent foundation for building scalable, reliable rate limiting systems that can protect your applications from abuse while ensuring fair resource allocation.

4.3 Column-Family Stores

Column-family stores organize data in columns rather than rows, enabling high performance for specific access patterns, especially for large-scale data with sparse attributes.

Key Concepts

1. **Column Families**: Groups of related columns
2. **Rows**: Identified by a unique key
3. **Columns**: Name-value pairs within column families
4. **Super Columns**: Columns that contain sub-columns (in some implementations)
5. **Wide Row Design**: Rows can have many columns, even millions
6. **Sparse Data Storage**: Only stores columns that have data

Apache Cassandra

Cassandra is a highly scalable, distributed NoSQL database designed for handling large amounts of data across multiple commodity servers.

Key Features:

- Masterless architecture (no single point of failure)
- Linear scalability
- Tunable consistency
- Efficient writes with log-structured storage engine
- CQL (Cassandra Query Language) similar to SQL
- Multi-datacenter replication
- Secondary indexes and materialized views

Data Model:

```
Keyspace (similar to a database)
└── Table (similar to a table)
    └── Row (identified by partition key)
        ├── Column Family 1
        │   ├── Column 1: Value
        │   ├── Column 2: Value
        │   └── Column 3: Value
        └── Column Family 2
            ├── Column 1: Value
            └── Column 2: Value
```

CQL Examples:

```
-- Create a keyspace
CREATE KEYSPACE e_commerce
WITH REPLICATION = {
    'class': 'SimpleStrategy',
    'replication_factor': 3
};

-- Use the keyspace
USE e_commerce;

-- Create a table
CREATE TABLE products (
    product_id UUID PRIMARY KEY,
    name TEXT,
    description TEXT,
    price DECIMAL,
    category TEXT,
    tags SET<TEXT>,
    attributes MAP<TEXT, TEXT>,
    created_at TIMESTAMP,
    updated_at TIMESTAMP
);
```

```
-- Insert data
INSERT INTO products (
    product_id, name, price, category, tags, attributes, created_at, updated_at
)
VALUES (
    uuid(),
    'Smartphone XYZ',
    699.99,
    'Electronics',
    {'mobile', 'smartphone', '5G'},
    {'color': 'black', 'storage': '128GB', 'RAM': '8GB'},
    toTimestamp(now()),
    toTimestamp(now())
);

-- Query data
SELECT name, price, category FROM products WHERE category = 'Electronics';

-- Update data
UPDATE products
SET price = 649.99, updated_at = toTimestamp(now())
WHERE product_id = 550e8400-e29b-41d4-a716-446655440000;

-- Delete data
DELETE FROM products WHERE product_id = 550e8400-e29b-41d4-a716-446655440000;
```

Composite Primary Keys and Clustering:

Cassandra uses the concept of partition keys and clustering columns to organize data:

```
-- User activity table with composite primary key
CREATE TABLE user_activity (
    user_id UUID,
    activity_date DATE,
    activity_time TIMESTAMP,
    activity_type TEXT,
    details TEXT,
    PRIMARY KEY ((user_id, activity_date), activity_time)
) WITH CLUSTERING ORDER BY (activity_time DESC);

-- The primary key has:
-- Partition key: (user_id, activity_date)
-- Clustering column: activity_time

-- This structure allows efficient queries like:
SELECT * FROM user_activity
WHERE user_id = 123 AND activity_date = '2023-06-15'
ORDER BY activity_time DESC
LIMIT 10;
```

Time Series Data Example:

Cassandra is particularly well-suited for time series data:

```
-- Sensor readings table optimized for time series data
CREATE TABLE sensor_readings (
    sensor_id TEXT,
    date DATE,
    timestamp TIMESTAMP,
    temperature FLOAT,
    humidity FLOAT,
    pressure FLOAT,
    PRIMARY KEY ((sensor_id, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);

-- Insert sensor readings
INSERT INTO sensor_readings (
    sensor_id, date, timestamp, temperature, humidity, pressure
)
VALUES (
    'sensor-001',
    '2023-06-15',
    '2023-06-15 14:30:00',
    22.5,
    45.2,
    1013.2
);

-- Query by time range
SELECT timestamp, temperature, humidity, pressure
FROM sensor_readings
WHERE sensor_id = 'sensor-001'
AND date = '2023-06-15'
AND timestamp >= '2023-06-15 12:00:00'
AND timestamp <= '2023-06-15 18:00:00';
```

Consistency Levels:

Cassandra allows tunable consistency for each operation:

```
// Java example with different consistency levels
import com.datastax.driver.core.*;

public class CassandraConsistencyExample {
    public static void main(String[] args) {
        Cluster cluster = Cluster.builder()
            .addContactPoint("127.0.0.1")
            .build();
        Session session = cluster.connect("e_commerce");

        // Read query with QUORUM consistency
        PreparedStatement selectStmt = session.prepare(
            "SELECT * FROM products WHERE product_id = ?"
        );
```

```
        BoundStatement boundSelect = selectStmt.bind(productId)
            .setConsistencyLevel(ConsistencyLevel.QUORUM);

        ResultSet results = session.execute(boundSelect);

        // Write query with ALL consistency
        PreparedStatement updateStmt = session.prepare(
            "UPDATE products SET price = ? WHERE product_id = ?"
        );

        BoundStatement boundUpdate = updateStmt.bind(newPrice, productId)
            .setConsistencyLevel(ConsistencyLevel.ALL);

        session.execute(boundUpdate);

        cluster.close();
    }
}
```

Secondary Indexes and Materialized Views:

```
-- Create a secondary index
CREATE INDEX idx_products_category ON products(category);

-- Use the index in a query
SELECT * FROM products WHERE category = 'Electronics';

-- Create a materialized view
CREATE MATERIALIZED VIEW products_by_category AS
    SELECT * FROM products
    WHERE category IS NOT NULL
    PRIMARY KEY (category, product_id);

-- Query the materialized view
SELECT * FROM products_by_category WHERE category = 'Electronics';
```

HBase

HBase is a distributed, scalable, big data store built on top of Hadoop and modeled after Google's BigTable.

Key Features:

- Linear and modular scalability
- Strictly consistent reads and writes
- Automatic sharding
- RegionServer failover
- Bloom filters and block cache for performance
- Integration with Hadoop ecosystem
- Real-time queries

Data Model:

```
Table
└── Row Key (uniquely identifies a row)
    ├── Column Family 1
    │   ├── Column Qualifier 1: Value + Timestamp
    │   └── Column Qualifier 2: Value + Timestamp
    └── Column Family 2
        ├── Column Qualifier 1: Value + Timestamp
        └── Column Qualifier 2: Value + Timestamp
```

HBase Shell Examples:

```
# Create a table with two column families
create 'users', 'info', 'contacts'

# Insert data
put 'users', 'user123', 'info:first_name', 'John'
put 'users', 'user123', 'info:last_name', 'Smith'
put 'users', 'user123', 'info:age', '35'
put 'users', 'user123', 'contacts:email', 'john@example.com'
put 'users', 'user123', 'contacts:phone', '555-1234'

# Get a specific row
get 'users', 'user123'

# Scan the table
scan 'users'

# Get specific columns
get 'users', 'user123', {COLUMNS => ['info:first_name', 'contacts:email']}

# Delete a cell
delete 'users', 'user123', 'contacts:phone'

# Delete a row
deleteall 'users', 'user123'
```

Java API Example:

```
import org.apache.hadoop.conf.Configuration;
import org.apache.hadoop.hbase.*;
import org.apache.hadoop.hbase.client.*;
import org.apache.hadoop.hbase.util.Bytes;

public class HBaseExample {
    public static void main(String[] args) throws Exception {
        // Configure HBase
        Configuration config = HBaseConfiguration.create();
        config.set("hbase.zookeeper.quorum", "localhost");
```

```
        config.set("hbase.zookeeper.property.clientPort", "2181");

        // Create a connection
        Connection connection = ConnectionFactory.createConnection(config);

        try {
            // Get table
            Table table = connection.getTable(TableName.valueOf("users"));

            // Create a put operation
            Put put = new Put(Bytes.toBytes("user456"));

            // Add columns
            put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("first_name"),
Bytes.toBytes("Jane"));
            put.addColumn(Bytes.toBytes("info"), Bytes.toBytes("last_name"),
Bytes.toBytes("Doe"));
            put.addColumn(Bytes.toBytes("contacts"), Bytes.toBytes("email"),
Bytes.toBytes("jane@example.com"));

            // Execute put
            table.put(put);

            // Create a get operation
            Get get = new Get(Bytes.toBytes("user456"));

            // Execute get
            Result result = table.get(get);

            // Print results
            byte[] firstName = result.getValue(Bytes.toBytes("info"),
Bytes.toBytes("first_name"));
            System.out.println("First Name: " + Bytes.toString(firstName));

            // Scan example
            Scan scan = new Scan();
            scan.addFamily(Bytes.toBytes("info"));

            ResultScanner scanner = table.getScanner(scan);
            for (Result scanResult : scanner) {
                byte[] key = scanResult.getRow();
                byte[] name = scanResult.getValue(Bytes.toBytes("info"),
Bytes.toBytes("first_name"));
                System.out.println("Key: " + Bytes.toString(key) + ", Name: " +
Bytes.toString(name));
            }

            // Clean up
            scanner.close();
            table.close();
        } finally {
            connection.close();
        }
    }
}
```

Column-Family Schema Design Patterns

1. **Wide Row Pattern**:
   - Store related data in a single row with many columns
   - Good for time-series or event data
   - Enables efficient slicing queries

```
-- Cassandra time-series example
CREATE TABLE temperature_by_sensor (
    sensor_id TEXT,
    day DATE,
    hour INT,
    minute INT,
    temperature FLOAT,
    PRIMARY KEY ((sensor_id, day), hour, minute)
);
```

2. **Tall Table Pattern**:
   - Store each attribute as a separate row
   - Good for sparse data or data with unknown attributes
   - Enables dynamic schema

```
HBase tall table example:
Row key: user123#first_name
Values: 'John'

Row key: user123#last_name
Values: 'Smith'

Row key: user123#email
Values: 'john@example.com'
```

3. **Composite Row Key Pattern**:
   - Combine multiple identifiers in the row key
   - Enables efficient range queries
   - Creates natural partitioning

```
-- Cassandra composite key example
CREATE TABLE user_logins (
    tenant_id TEXT,
    user_id UUID,
    login_date DATE,
    login_time TIMESTAMP,
    ip_address TEXT,
    device TEXT,
```

```
    PRIMARY KEY ((tenant_id, login_date), user_id, login_time)
) WITH CLUSTERING ORDER BY (user_id ASC, login_time DESC);
```

4. **Row Key Hashing Pattern**:
    - Use hashing to distribute data evenly
    - Prevents hot spots for sequential access

```
// HBase key hashing example
String userId = "user123";
int bucket = Math.abs(userId.hashCode() % 10);
String rowKey = bucket + ":" + userId;
```

5. **Time-Based Partitioning**:
    - Include time components in partition keys
    - Natural way to manage data lifecycle
    - Enables efficient time-based queries

```
-- Cassandra time-partitioned example
CREATE TABLE events (
    application_id TEXT,
    year INT,
    month INT,
    event_id UUID,
    event_time TIMESTAMP,
    event_type TEXT,
    payload TEXT,
    PRIMARY KEY ((application_id, year, month), event_time, event_id)
) WITH CLUSTERING ORDER BY (event_time DESC, event_id ASC);
```

Interview Questions

Q: How would you design a Cassandra data model for a time-series application that needs to store sensor readings from millions of IoT devices?

A: Designing a Cassandra data model for a time-series application with millions of IoT devices requires careful consideration of data access patterns, write/read efficiency, and scalability. I'll outline a comprehensive approach that addresses these concerns while optimizing for Cassandra's strengths.

Understanding the Requirements

Before diving into the data model, let's consider the typical requirements for an IoT sensor data application:

1. **High write throughput**: Millions of devices sending readings frequently
2. **Time-based queries**: Retrieving data for specific time ranges
3. **Device-specific queries**: Getting all readings for a specific device
4. **Aggregations**: Computing statistics over time periods

5. **Data retention policies**: Managing data lifecycle as older data becomes less valuable
6. **Scalability**: Handling growing numbers of devices and increasing data volume

Cassandra Data Model Design Principles

When designing for Cassandra, I follow these key principles:

1. **Partition key design**: Choose partition keys that distribute data evenly and align with query patterns
2. **Clustering columns**: Order data within partitions to enable efficient range queries
3. **Denormalization**: Duplicate data to optimize for specific query patterns
4. **Time bucketing**: Group time-series data into manageable time buckets
5. **Query-first approach**: Design tables specifically for known query patterns

Core Data Model

I'll create multiple tables optimized for different query patterns. The primary table for storing raw sensor readings would be:

```
CREATE TABLE sensor_readings (
    sensor_id TEXT,
    reading_date DATE,
    reading_time TIMESTAMP,
    temperature FLOAT,
    humidity FLOAT,
    pressure FLOAT,
    battery_level FLOAT,
    alert_flag BOOLEAN,
    metadata MAP<TEXT, TEXT>,
    PRIMARY KEY ((sensor_id, reading_date), reading_time)
) WITH CLUSTERING ORDER BY (reading_time DESC);
```

This model uses a composite partition key of `sensor_id` and `reading_date`, with `reading_time` as a clustering column.

Design rationale:

1. **Partition key choice**:
   - `sensor_id` ensures data for each device is grouped together
   - `reading_date` creates time-based partitioning, preventing unbounded partition growth
   - Together they distribute data evenly across the cluster while enabling efficient queries
2. **Clustering column**:
   - `reading_time` orders readings within a partition chronologically
   - `DESC` order makes the newest readings appear first in queries
3. **Data fields**:
   - Core measurements like temperature, humidity, pressure
   - Operational data like battery level and alert flags
   - Flexible `metadata` map for device-specific attributes

Implementing Time Buckets

For high-frequency devices, even a single day might contain too many readings in one partition. We can use smaller time buckets:

```
CREATE TABLE sensor_readings_by_hour (
    sensor_id TEXT,
    reading_date DATE,
    hour INT,
    reading_time TIMESTAMP,
    temperature FLOAT,
    humidity FLOAT,
    pressure FLOAT,
    battery_level FLOAT,
    alert_flag BOOLEAN,
    metadata MAP<TEXT, TEXT>,
    PRIMARY KEY ((sensor_id, reading_date, hour), reading_time)
) WITH CLUSTERING ORDER BY (reading_time DESC);
```

This further subdivides the partitions, ensuring they remain manageable even with high-frequency readings.

Query-Specific Tables

To support various query patterns efficiently, I'll create additional tables:

1. Recent Readings Across All Sensors

```
CREATE TABLE recent_readings (
    bucket_id INT,
    reading_time TIMESTAMP,
    sensor_id TEXT,
    temperature FLOAT,
    humidity FLOAT,
    pressure FLOAT,
    alert_flag BOOLEAN,
    PRIMARY KEY ((bucket_id), reading_time, sensor_id)
) WITH CLUSTERING ORDER BY (reading_time DESC, sensor_id ASC);
```

The `bucket_id` is a small integer (0-9) derived from a simple hash or modulo of the timestamp, distributing recent readings across a fixed number of partitions. This enables queries like "show me the most recent readings across all sensors" without needing to query every sensor individually.

2. Alerts Table

```
CREATE TABLE sensor_alerts (
    alert_date DATE,
    alert_time TIMESTAMP,
    sensor_id TEXT,
    alert_type TEXT,
```

```
    reading_value FLOAT,
    threshold_value FLOAT,
    PRIMARY KEY ((alert_date), alert_time, sensor_id)
) WITH CLUSTERING ORDER BY (alert_time DESC, sensor_id ASC);
```

This table is specifically designed for monitoring systems to quickly identify all sensors that have triggered alerts.

3. Aggregated Statistics Table

For efficient retrieval of pre-computed statistics:

```
CREATE TABLE sensor_stats_hourly (
    sensor_id TEXT,
    stat_date DATE,
    hour INT,
    min_temperature FLOAT,
    max_temperature FLOAT,
    avg_temperature FLOAT,
    min_humidity FLOAT,
    max_humidity FLOAT,
    avg_humidity FLOAT,
    reading_count INT,
    PRIMARY KEY ((sensor_id, stat_date), hour)
) WITH CLUSTERING ORDER BY (hour ASC);

CREATE TABLE sensor_stats_daily (
    sensor_id TEXT,
    year INT,
    month INT,
    day INT,
    min_temperature FLOAT,
    max_temperature FLOAT,
    avg_temperature FLOAT,
    min_humidity FLOAT,
    max_humidity FLOAT,
    avg_humidity FLOAT,
    reading_count INT,
    PRIMARY KEY ((sensor_id, year, month), day)
) WITH CLUSTERING ORDER BY (day ASC);
```

These tables store pre-computed aggregates, enabling efficient retrieval of statistics without scanning raw data.

Data Insertion Strategies

To efficiently insert data into multiple tables, I would use batch statements with an asynchronous approach:

```
// Java example of inserting data into multiple tables
public void insertSensorReading(SensorReading reading) {
    // Prepare statements for each table (typically cached/prepared in advance)
    PreparedStatement insertRawStatement = session.prepare(
        "INSERT INTO sensor_readings (sensor_id, reading_date, reading_time,
```

```
temperature, " +
        "humidity, pressure, battery_level, alert_flag, metadata) " +
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?)"
    );

    PreparedStatement insertHourlyStatement = session.prepare(
        "INSERT INTO sensor_readings_by_hour (sensor_id, reading_date, hour,
reading_time, " +
        "temperature, humidity, pressure, battery_level, alert_flag, metadata) " +
        "VALUES (?, ?, ?, ?, ?, ?, ?, ?, ?, ?)"
    );

    // Extract date components
    LocalDateTime dateTime = reading.getTimestamp().toLocalDateTime();
    LocalDate date = dateTime.toLocalDate();
    int hour = dateTime.getHour();

    // Create batch for atomic insertion
    BatchStatement batch = new BatchStatement(BatchStatement.Type.UNLOGGED);

    // Add statements to batch
    batch.add(insertRawStatement.bind(
        reading.getSensorId(),
        date,
        reading.getTimestamp(),
        reading.getTemperature(),
        reading.getHumidity(),
        reading.getPressure(),
        reading.getBatteryLevel(),
        reading.isAlertFlag(),
        reading.getMetadata()
    ));

    batch.add(insertHourlyStatement.bind(
        reading.getSensorId(),
        date,
        hour,
        reading.getTimestamp(),
        reading.getTemperature(),
        reading.getHumidity(),
        reading.getPressure(),
        reading.getBatteryLevel(),
        reading.isAlertFlag(),
        reading.getMetadata()
    ));

    // Add to recent readings table
    int bucketId = reading.getTimestamp().getTime() % 10;
    batch.add(insertRecentStatement.bind(
        bucketId,
        reading.getTimestamp(),
        reading.getSensorId(),
        reading.getTemperature(),
        reading.getHumidity(),
        reading.getPressure(),
```

```
            reading.isAlertFlag()
        ));

        // Add to alerts table if needed
        if (reading.isAlertFlag()) {
            batch.add(insertAlertStatement.bind(
                date,
                reading.getTimestamp(),
                reading.getSensorId(),
                reading.getAlertType(),
                reading.getAlertValue(),
                reading.getThresholdValue()
            ));
        }

        // Execute batch asynchronously
        session.executeAsync(batch);
    }
```

Handling Aggregations

For aggregated statistics, we can use Spark or a dedicated service that periodically computes aggregates:

```
// Pseudocode for statistics aggregation
public void computeHourlyStatistics(LocalDate date, int hour) {
    // Query raw data
    String cql = "SELECT sensor_id, temperature, humidity " +
                 "FROM sensor_readings_by_hour " +
                 "WHERE reading_date = ? AND hour = ?";

    ResultSet results = session.execute(cql, date, hour);

    Map<String, StatisticAccumulator> accumulators = new HashMap<>();

    // Process results
    for (Row row : results) {
        String sensorId = row.getString("sensor_id");
        float temperature = row.getFloat("temperature");
        float humidity = row.getFloat("humidity");

        StatisticAccumulator acc = accumulators.computeIfAbsent(
            sensorId, id -> new StatisticAccumulator()
        );

        acc.addTemperature(temperature);
        acc.addHumidity(humidity);
    }

    // Insert aggregated statistics
    BatchStatement batch = new BatchStatement();

    for (Map.Entry<String, StatisticAccumulator> entry : accumulators.entrySet()) {
        String sensorId = entry.getKey();
```

```
            StatisticAccumulator acc = entry.getValue();

            batch.add(insertHourlyStatStatement.bind(
                sensorId,
                date,
                hour,
                acc.getMinTemperature(),
                acc.getMaxTemperature(),
                acc.getAvgTemperature(),
                acc.getMinHumidity(),
                acc.getMaxHumidity(),
                acc.getAvgHumidity(),
                acc.getCount()
            ));
        }

        session.execute(batch);
    }
```

Data Lifecycle Management

For time-series data, older data often becomes less valuable. Cassandra's Time To Live (TTL) feature can help manage data lifecycle:

```
-- Set TTL for raw data
ALTER TABLE sensor_readings WITH default_time_to_live = 2592000; -- 30 days

-- Keep aggregated data longer
ALTER TABLE sensor_stats_daily WITH default_time_to_live = 31536000; -- 1 year
```

Alternatively, we can use scheduled jobs to delete old data using explicit CQL:

```
-- Delete data older than 30 days
DELETE FROM sensor_readings
WHERE sensor_id = ? AND reading_date < dateOf(now()) - 30;
```

Querying Examples

Here are examples of common queries for this data model:

```
-- Get most recent readings for a specific sensor
SELECT * FROM sensor_readings
WHERE sensor_id = 'device-123'
AND reading_date = toDate(now())
LIMIT 100;

-- Get readings for a specific time range
SELECT * FROM sensor_readings
```

```
WHERE sensor_id = 'device-123'
AND reading_date >= '2023-06-01'
AND reading_date <= '2023-06-07'
AND reading_time >= '2023-06-01 08:00:00'
AND reading_time <= '2023-06-07 17:00:00';

-- Get hourly temperature statistics for a sensor over a month
SELECT * FROM sensor_stats_hourly
WHERE sensor_id = 'device-123'
AND stat_date >= '2023-06-01'
AND stat_date <= '2023-06-30';

-- Get all alerts from today
SELECT * FROM sensor_alerts
WHERE alert_date = toDate(now());

-- Get most recent readings across all sensors
SELECT * FROM recent_readings
WHERE bucket_id IN (0, 1, 2, 3, 4, 5, 6, 7, 8, 9)
LIMIT 100;
```

Performance Optimization Techniques

1. **Read Optimizations**:
   - Use the `ALLOW FILTERING` clause sparingly and only when necessary
   - Leverage materialized views for complex queries
   - Use secondary indexes only for low-cardinality fields
   - Implement proper paging with the `PagingState` object
2. **Write Optimizations**:
   - Use unlogged batches for related updates
   - Set appropriate consistency levels based on importance
   - Consider using the async API for non-critical writes
   - Monitor and tune write timeout settings
3. **Schema Optimizations**:
   - Choose appropriate compaction strategies:

     ```
     ALTER TABLE sensor_readings
     WITH compaction = {'class': 'TimeWindowCompactionStrategy',
                        'compaction_window_size': 1,
                        'compaction_window_unit': 'DAYS'};
     ```

   - Configure proper compression:

     ```
     ALTER TABLE sensor_readings
     WITH compression = {'class': 'LZ4Compressor'};
     ```

Handling Scale and Growth

As the system grows to handle more devices and data, consider:

1. **Partition Sizing**:
    - For time-based partitioning, select appropriate time buckets based on data volume:
        - High-frequency sensors might need hourly partitions
        - Low-frequency sensors could use daily or weekly partitions
    - Monitor partition sizes and adjust as needed
2. **Cluster Expansion**:
    - Add nodes to the Cassandra cluster as data volume grows
    - Use vnodes to evenly distribute data during expansion
    - Regularly run repairs to ensure data consistency
3. **Monitoring and Alerting**:
    - Track key metrics like write latency, read latency, and partition sizes
    - Set up alerts for anomalies in data collection
    - Monitor disk usage and forecast capacity needs

Summary

This Cassandra data model for IoT sensor data incorporates several best practices:

1. **Efficient partitioning strategy**: Using composite partition keys with time bucketing to distribute data evenly and maintain manageable partition sizes
2. **Multiple targeted tables**: Creating separate tables for different query patterns rather than trying to create one table for all use cases
3. **Pre-computed aggregations**: Storing aggregated statistics to avoid expensive read-time computations
4. **Data lifecycle management**: Implementing TTL or scheduled deletions to manage the data lifecycle
5. **Optimized for high write throughput**: The model is designed to handle millions of sensors sending frequent readings

This design provides a scalable, high-performance solution that can handle the demands of large-scale IoT deployments while enabling efficient queries for both real-time and historical data analysis.

Q: How does HBase differ from Cassandra in terms of architecture, data model, and use cases? When would you choose one over the other?

A: HBase and Cassandra are both powerful column-family NoSQL databases designed for large-scale distributed data storage, but they differ significantly in their architecture, consistency models, data models, and ideal use cases. Let me explore these differences comprehensively to help understand when to choose one over the other.

Architectural Differences

Cassandra Architecture

1. **Peer-to-Peer Model**:
    - Fully decentralized, masterless architecture
    - All nodes are equal with no single point of failure
    - Data is distributed across nodes using consistent hashing
    - Any node can accept read or write requests
2. **Ring Topology**:
    - Nodes are arranged in a ring structure
    - Each node is responsible for a range of data determined by its token range
    - Data is replicated across multiple nodes based on the replication factor
3. **Gossip Protocol**:
    - Nodes communicate state information with each other
    - Enables automatic discovery of joining or failing nodes
    - Self-healing capabilities when nodes fail or rejoin
4. **Independent Scaling**:
    - Can scale linearly by simply adding more nodes
    - No need to restart the cluster when adding or removing nodes
    - Automatic data rebalancing (though manual operations are sometimes needed)

HBase Architecture

1. **Master-Slave Model**:
    - HMaster servers coordinate and manage the cluster
    - RegionServers store and serve data
    - ZooKeeper ensemble provides coordination services
    - Clear separation of responsibilities between components
2. **HDFS Dependency**:
    - Relies on Hadoop Distributed File System (HDFS) for data storage
    - Benefits from HDFS's reliability and fault tolerance
    - Inherits HDFS limitations and dependencies
3. **Region-Based Data Distribution**:
    - Data is split into "regions" based on row key ranges
    - Regions are dynamically split as they grow
    - RegionServers host and serve multiple regions
4. **Hierarchical Structure**:
    - Requires a functioning ZooKeeper cluster
    - HMaster manages RegionServer assignments and load balancing
    - Client requests go through coordination servers

Data Model Differences

Cassandra Data Model

1. **CQL (Cassandra Query Language)**:
    - SQL-like query language
    - Familiar syntax for SQL users
    - More abstracted from underlying storage
2. **Keyspace/Table Structure**:
    - Keyspaces contain tables (similar to database/table in SQL)
    - Tables contain rows and columns
    - Clear separation between partition keys, clustering columns, and regular columns
3. **Wide Row Model**:
    - Rows can have many columns (thousands or more)
    - Clustering columns provide ordering within partitions
    - Support for complex data types (collections, user-defined types)
4. **Secondary Indexes and Materialized Views**:
    - Native support for secondary indexes (with limitations)
    - Materialized views for alternative access patterns
    - Indexes managed by the database

HBase Data Model

1. **Key-Value API**:
    - Lower-level, direct access to the data model
    - Operations like Get, Put, Scan rather than SQL-like queries
    - More explicit control over storage details
2. **HBase Shell and Thrift API**:
    - Command-line shell for administration and queries
    - Thrift interface for cross-language client support
    - Phoenix provides SQL-like layer on top of HBase
3. **Table/Row/Column Family Structure**:
    - Tables contain rows identified by row keys
    - Column families must be defined at table creation
    - Columns (qualifiers) can be added dynamically within column families
    - Cell values are versioned with timestamps
4. **Sparse Data Storage**:
    - Only stores cells that contain data
    - No storage overhead for null values
    - Efficient for very sparse datasets

Consistency Model Differences

Cassandra Consistency

1. **Tunable Consistency**:
    - Configurable consistency levels per query
    - Options from ONE (fastest, least consistent) to ALL (slowest, most consistent)
    - Client can choose appropriate tradeoff for each operation
2. **Eventually Consistent by Default**:
    - Optimized for availability and partition tolerance
    - Accepts writes even during network partitions
    - Resolves conflicts using timestamp-based reconciliation (last write wins)
3. **Quorum-Based Operations**:
    - Consistency levels like QUORUM, LOCAL_QUORUM ensure data durability
    - (N/2 + 1) replicas must acknowledge for strong consistency
    - Flexible tradeoffs between consistency and availability

HBase Consistency

1. **Strong Consistency**:
    - Always provides strong consistency for reads and writes
    - No eventual consistency options
    - Single region server handles all requests for a row
2. **ACID Compliance for Single Row**:
    - Atomic row-level operations
    - Consistent view of data
    - Durable write-ahead logging
    - Isolation between operations
3. **Region-Level Locking**:
    - Uses locks to ensure consistency
    - Can impact performance in high-concurrency scenarios
    - No tunable options to trade consistency for performance

Performance Characteristics

Cassandra Performance

1. **Write Optimization**:
    - Extremely fast writes due to append-only log structure
    - Scales linearly as nodes are added
    - Excellent for write-heavy workloads
    - Commitlog ensures durability

2. **Read Performance**:
    - Can be slower than writes, especially for non-primary-key reads
    - Read performance depends on consistency level chosen
    - May require careful data modeling for efficient reads
3. **Multi-Datacenter Support**:
    - Native, built-in multi-datacenter replication
    - Configurable per-datacenter replication factors
    - Ideal for globally distributed applications

HBase Performance

1. **Read Optimization**:
    - Excellent read performance for key-based access
    - In-memory caching with BlockCache
    - Bloom filters to optimize disk reads
2. **Write Performance**:
    - Uses Write-Ahead Log (WAL) for durability
    - MemStore buffers writes in memory
    - Periodic compaction to optimize storage
3. **Scan Performance**:
    - Efficient for range scans by row key
    - Coprocessors for server-side processing
    - Filtering at the server level to reduce network transfer

Use Case Comparison

When to Choose Cassandra

1. **Global, Multi-Datacenter Deployments**:
    - Applications requiring global data distribution
    - Multi-region redundancy with configurable consistency
    - Example: Global user profile service with low latency in all regions

```
CREATE KEYSPACE user_profiles
WITH REPLICATION = {
    'class': 'NetworkTopologyStrategy',
    'us-east': 3,
    'eu-west': 3,
    'ap-south': 3
};
```

2. **Write-Heavy Workloads**:

- High-volume event ingestion
- IoT data collection systems
- Logging and monitoring systems
- Example: Collecting sensor readings from millions of devices

```
INSERT INTO sensor_readings
(sensor_id, reading_date, reading_time, temperature, humidity)
VALUES ('device-123', '2023-06-15', '2023-06-15 14:30:00', 22.5, 45.2);
```

3. **Systems Requiring High Availability**:
   - Services that cannot tolerate downtime
   - Applications where availability trumps strong consistency
   - Example: Shopping cart service that must always accept updates

```
// Write with LOCAL_ONE consistency for maximum availability
session.execute(insertStatement.bind(cartId, itemId, quantity)
    .setConsistencyLevel(ConsistencyLevel.LOCAL_ONE));
```

4. **Time-Series Data with Time-Based Querying**:
   - Applications requiring efficient time-based queries
   - Systems with time-bucketed access patterns
   - Example: Financial tick data or monitoring metrics

```
CREATE TABLE metrics (
    app_id TEXT,
    metric_date DATE,
    metric_hour INT,
    metric_time TIMESTAMP,
    name TEXT,
    value DOUBLE,
    PRIMARY KEY ((app_id, metric_date, metric_hour), metric_time, name)
);
```

5. **Applications Requiring Tunable Consistency**:
   - Systems with varying consistency requirements
   - Use cases where fast writes sometimes matter more than strong consistency
   - Example: Social media status updates vs. financial transactions

```
// Non-critical update with lower consistency
session.execute(updateStatusStatement.bind(userId, status)
    .setConsistencyLevel(ConsistencyLevel.ONE));

// Critical financial operation with strong consistency
```

```
session.execute(transferFundsStatement.bind(fromAccount, toAccount, amount)
    .setConsistencyLevel(ConsistencyLevel.QUORUM));
```

When to Choose HBase

1. **Hadoop Ecosystem Integration**:
   - Applications already using Hadoop/HDFS
   - Need for integration with MapReduce, Spark, Hive
   - Example: Data lake with mixed analytical and operational workloads

```
// Running MapReduce job on HBase data
TableMapReduceUtil.initTableMapperJob(
    "users",                  // Input table
    new Scan(),               // Scan instance
    MyMapper.class,           // Mapper class
    Text.class,               // Mapper output key
    IntWritable.class,        // Mapper output value
    job);
```

2. **Strong Consistency Requirements**:
   - Applications requiring strong consistency guarantees
   - Use cases where eventual consistency is not acceptable
   - Example: Financial record keeping or inventory management

```
// HBase provides strong consistency by default
Get get = new Get(Bytes.toBytes("customer123"));
Result result = table.get(get);
// Result is always consistent with the latest writes
```

3. **Random Access to Big Data**:
   - Applications requiring random access to petabyte-scale datasets
   - Use cases needing both random and sequential access patterns
   - Example: User profile store with billions of users

```
// Random access by key
Get get = new Get(Bytes.toBytes("user123"));
get.addFamily(Bytes.toBytes("profile"));
Result result = table.get(get);

// Scan for range of users
Scan scan = new Scan(Bytes.toBytes("user100"), Bytes.toBytes("user200"));
scan.addFamily(Bytes.toBytes("profile"));
ResultScanner scanner = table.getScanner(scan);
```

4. **Semi-Structured Data with Evolving Schema**:
    - Applications with unpredictable or evolving schemas
    - Need to add new attributes without schema changes
    - Example: Product catalog with varying attributes per product category

```
// Adding new columns dynamically
Put put = new Put(Bytes.toBytes("product123"));
put.addColumn(Bytes.toBytes("details"), Bytes.toBytes("color"),
Bytes.toBytes("red"));
put.addColumn(Bytes.toBytes("details"), Bytes.toBytes("new_attribute"),
Bytes.toBytes("value"));
table.put(put);
```

5. **Coprocessor Support for Server-Side Operations**:
    - Applications requiring custom server-side processing
    - Reducing data transfer for complex operations
    - Example: Real-time aggregation or filtering at data source

```
// Using coprocessors for server-side computations
public static class SumEndpoint extends SumProtocol implements Coprocessor,
CoprocessorService {
    // Implementation for server-side sum calculation
}

// Client invocation
Map<byte[], Long> results = table.coprocessorService(
    SumProtocol.class,
    startKey,
    endKey,
    new Batch.Call<SumProtocol, Long>() {
        public Long call(SumProtocol instance) throws IOException {
            return instance.getSum(Bytes.toBytes("cf"),
Bytes.toBytes("column"));
        }
    }
);
```

Operational Considerations

Cassandra Operations

1. **Node Management**:
    - Simpler to add/remove nodes
    - Built-in support for different workload profiles (transactional vs. analytical)
    - Automatic data rebalancing (with some manual intervention for optimal results)

2. **Multi-Datacenter Management**:

- Native tools for cross-datacenter replication
- Configurable per-datacenter settings

3. **Backup and Recovery**:
   - Snapshots for backups
   - Incremental backups support
   - Point-in-time recovery possible

4. **Maintenance Operations**:
   - Repairs to ensure data consistency
   - Compaction to optimize storage
   - Relatively complex operational requirements

HBase Operations

1. **Dependency Management**:
   - Requires managing HDFS and ZooKeeper
   - More moving parts to maintain
   - Version compatibility concerns with Hadoop ecosystem

2. **Region Management**:
   - Region splits and compactions
   - Region assignment and load balancing
   - Master server failover

3. **Backup and Recovery**:
   - Leverages HDFS replication and snapshots
   - Export/Import utilities
   - Replication for disaster recovery

4. **Operational Complexity**:
   - Generally considered more complex to operate
   - Requires understanding of Hadoop ecosystem
   - More configuration parameters to tune

Decision Framework

When deciding between Cassandra and HBase, consider these key questions:

1. **Consistency Requirements**:
   - Need for strong consistency → HBase
   - Flexible consistency tradeoffs → Cassandra

2. **Write vs. Read Optimization**:
   - Write-heavy workloads → Cassandra
   - Read-heavy workloads with key-based access → HBase

- Mixed workloads → Depends on other factors

3. **Scaling Pattern**:
   - Global, multi-datacenter distribution → Cassandra
   - Single region with massive data volume → Either works, HBase may have edge
4. **Existing Infrastructure**:
   - Already using Hadoop ecosystem → HBase
   - Standalone deployment needed → Cassandra
5. **Operational Resources**:
   - Limited operational team → Cassandra (typically)
   - Strong Hadoop expertise → HBase
6. **Query Patterns**:
   - Complex time-series access patterns → Cassandra
   - Random access by key with strong consistency → HBase
   - Need for SQL-like language → Cassandra (CQL)

Real-World Example

Let's consider a real-world example: building a customer data platform that stores customer profiles, interactions, and events.

Using Cassandra:

```
-- Customer profiles
CREATE TABLE customers (
    customer_id UUID,
    name TEXT,
    email TEXT,
    created_at TIMESTAMP,
    updated_at TIMESTAMP,
    preferences MAP<TEXT, TEXT>,
    PRIMARY KEY (customer_id)
);

-- Customer events with time-based bucketing
CREATE TABLE customer_events (
    customer_id UUID,
    event_date DATE,
    event_time TIMESTAMP,
    event_id UUID,
    event_type TEXT,
    event_data MAP<TEXT, TEXT>,
    PRIMARY KEY ((customer_id, event_date), event_time, event_id)
) WITH CLUSTERING ORDER BY (event_time DESC, event_id ASC);

-- Query: Get customer profile
SELECT * FROM customers WHERE customer_id = 123e4567-e89b-12d3-a456-426614174000;
```

```
-- Query: Get recent customer events
SELECT event_time, event_type, event_data
FROM customer_events
WHERE customer_id = 123e4567-e89b-12d3-a456-426614174000
AND event_date >= '2023-06-01'
AND event_date <= '2023-06-15'
LIMIT 100;
```

Using HBase:

```
// Create tables
admin.createTable(new HTableDescriptor(TableName.valueOf("customers"))
    .addFamily(new HColumnDescriptor("profile"))
    .addFamily(new HColumnDescriptor("preferences")));

admin.createTable(new HTableDescriptor(TableName.valueOf("customer_events"))
    .addFamily(new HColumnDescriptor("events")));

// Store customer profile
Put customerPut = new Put(Bytes.toBytes("customer123"));
customerPut.addColumn(Bytes.toBytes("profile"), Bytes.toBytes("name"),
Bytes.toBytes("John Smith"));
customerPut.addColumn(Bytes.toBytes("profile"), Bytes.toBytes("email"),
Bytes.toBytes("john@example.com"));
customerPut.addColumn(Bytes.toBytes("preferences"), Bytes.toBytes("language"),
Bytes.toBytes("en"));
customersTable.put(customerPut);

// Store customer event
// Row key design: customer_id#reverse_timestamp#event_id
String eventRowKey = "customer123#" + (Long.MAX_VALUE - System.currentTimeMillis())
+ "#event456";
Put eventPut = new Put(Bytes.toBytes(eventRowKey));
eventPut.addColumn(Bytes.toBytes("events"), Bytes.toBytes("type"),
Bytes.toBytes("purchase"));
eventPut.addColumn(Bytes.toBytes("events"), Bytes.toBytes("amount"),
Bytes.toBytes("125.99"));
eventsTable.put(eventPut);

// Get customer profile
Get customerGet = new Get(Bytes.toBytes("customer123"));
Result customerResult = customersTable.get(customerGet);
String name = Bytes.toString(customerResult.getValue(Bytes.toBytes("profile"),
Bytes.toBytes("name")));

// Scan recent customer events
Scan eventScan = new Scan();
eventScan.setRowPrefixFilter(Bytes.toBytes("customer123#"));
eventScan.setMaxResultSize(100);
ResultScanner scanner = eventsTable.getScanner(eventScan);
for (Result result : scanner) {
    // Process events
```

```
}
scanner.close();
```

Which is better?

For this customer data platform:

- **Cassandra advantages**:
  - Better for high-volume event ingestion
  - CQL provides more intuitive querying
  - Easier to set up multiple regional deployments
  - Time-bucketed model fits customer events well
- **HBase advantages**:
  - Strong consistency for customer profile data
  - Potentially better for analytical queries with MapReduce
  - More efficient for very sparse customer attributes
  - Coprocessors could enable server-side data processing

The final decision would depend on additional factors like:

- Global distribution requirements
- Consistency needs for customer data
- Volume of events
- Integration with existing systems
- Operational expertise

Summary

Both Cassandra and HBase are powerful column-family databases designed for large-scale distributed data management, but they make different trade-offs:

- **Cassandra** offers a masterless architecture, tunable consistency, excellent write performance, and built-in multi-datacenter replication. It excels for write-heavy workloads, global deployments, and applications requiring high availability.
- **HBase** provides strong consistency, tight Hadoop integration, excellent random access to big data, and server-side processing via coprocessors. It's ideal for applications requiring strong consistency, random access to massive datasets, and integration with the Hadoop ecosystem.

The choice between them should be based on your specific requirements for consistency, availability, partition tolerance, scaling patterns, and operational constraints.

4.4 NoSQL Data Modeling

NoSQL data modeling differs significantly from relational database modeling, focusing on access patterns rather than normalized relationships.

Core Principles of NoSQL Data Modeling

1. **Denormalization**: Duplicating data to optimize read performance
2. **Aggregation**: Storing related data together in a single document or row
3. **Application-Driven Schema**: Designing data models based on application queries
4. **Hierarchical Data**: Leveraging nested structures for related entities
5. **Query-First Design**: Starting with query patterns, then designing the data model

NoSQL vs. Relational Data Modeling Approaches

Relational Approach	NoSQL Approach
Normalize data to reduce redundancy	Denormalize data to optimize query performance
Design tables around entities	Design collections/tables around query patterns
Joins to retrieve related data	Embed related data or use application-side joins
Rigid schema enforced by database	Flexible schema managed by application
Optimize for storage efficiency	Optimize for read/write patterns
ACID transactions across entities	BASE principles, eventual consistency

Common NoSQL Data Modeling Patterns

1. **Embedding Pattern**:
   - Storing related data within a single document or row
   - Good for one-to-few relationships and frequently accessed together

```
// MongoDB example - Order with embedded line items
{
  "order_id": "ORD12345",
  "customer_id": "CUST789",
  "order_date": "2023-06-15",
  "shipping_address": {
    "street": "123 Main St",
    "city": "Boston",
    "state": "MA",
    "zip": "02101"
  },
  "items": [
    {
      "product_id": "PROD123",
      "product_name": "Smartphone",
      "quantity": 1,
      "price": 699.99
    },
    {
      "product_id": "PROD456",
      "product_name": "Phone Case",
      "quantity": 2,
      "price": 24.99
    }
  ],
```

```
  "total": 749.97
}
```

2. **Reference Pattern**:
   - Storing references to related data
   - Good for one-to-many or many-to-many relationships

```
// MongoDB example - Product with category reference
// Product document
{
  "product_id": "PROD123",
  "name": "Smartphone",
  "price": 699.99,
  "category_id": "CAT_ELEC" // Reference to category
}

// Category document
{
  "category_id": "CAT_ELEC",
  "name": "Electronics",
  "description": "Electronic devices and accessories"
}
```

3. **Computed Pattern**:
   - Storing precomputed values to avoid complex calculations

```
// MongoDB example - Order with computed totals
{
  "order_id": "ORD12345",
  "items": [
    {
      "product_id": "PROD123",
      "quantity": 1,
      "unit_price": 699.99,
      "item_total": 699.99 // Precomputed value
    },
    {
      "product_id": "PROD456",
      "quantity": 2,
      "unit_price": 24.99,
      "item_total": 49.98 // Precomputed value
    }
  ],
  "subtotal": 749.97, // Precomputed value
  "tax": 60.00, // Precomputed value
  "total": 809.97 // Precomputed value
}
```

4. **Subset Pattern**:
   - Including a subset of related entity's data
   - Good for reducing the need for additional queries

```
// MongoDB example - Order with product subset
{
  "order_id": "ORD12345",
  "customer_id": "CUST789",
  "items": [
    {
      "product_id": "PROD123",
      "name": "Smartphone", // Subset of product data
      "category": "Electronics", // Subset of product data
      "quantity": 1,
      "price": 699.99
    }
  ]
}
```

5. **Version Pattern**:
   - Adding version information to track schema changes

```
// MongoDB example - Document with schema version
{
  "user_id": "USER123",
  "schema_version": 2, // Track schema version
  "name": "John Smith",
  "email": "john@example.com",
  "preferences": {
    "theme": "dark",
    "notifications": true
  }
}
```

6. **Tree Pattern**:
   - Modeling hierarchical relationships

```
// MongoDB example - Category hierarchy with path array
{
  "category_id": "CAT_PHONES",
  "name": "Smartphones",
  "parent_id": "CAT_ELEC",
  "ancestors": ["CAT_ROOT", "CAT_ELEC"], // Path to root
  "level": 2
}
```

7. **Extended Reference Pattern**:
    - Including frequently accessed attributes from referenced entities

```
// MongoDB example - Blog post with author reference and subset
{
  "post_id": "POST123",
  "title": "NoSQL Data Modeling",
  "content": "...",
  "author_id": "USER456", // Reference
  "author_name": "Jane Smith", // Extended reference
  "author_avatar": "https://example.com/avatars/jane.jpg" // Extended reference
}
```

8. **Bucketing Pattern**:
    - Grouping related items into buckets or chunks
    - Useful for time series or log data

```
// MongoDB example - Sensor readings bucketed by day
{
  "sensor_id": "SENSOR123",
  "date": "2023-06-15",
  "readings": [
    { "time": "08:00:00", "temperature": 22.5, "humidity": 45 },
    { "time": "08:30:00", "temperature": 23.1, "humidity": 44 },
    { "time": "09:00:00", "temperature": 23.8, "humidity": 43 }
  ]
}
```

Data Modeling for Specific NoSQL Database Types

1. **Document Databases (MongoDB, Couchbase)**:
    - Focus on embedding vs. referencing decisions
    - Consider document size limits (16MB in MongoDB)
    - Use array fields for one-to-many relationships
    - Leverage document validation for schema enforcement

```
// MongoDB schema validation example
db.createCollection('products', {
  validator: {
    $jsonSchema: {
      bsonType: 'object',
      required: ['name', 'price', 'category'],
      properties: {
        name: {
          bsonType: 'string',
          description: 'must be a string and is required',
```

```
        },
        price: {
          bsonType: 'decimal',
          minimum: 0,
          description: 'must be a positive decimal and is required',
        },
        category: {
          bsonType: 'string',
          description: 'must be a string and is required',
        },
      },
    },
  },
});
```

2. **Key-Value Stores (Redis, DynamoDB)**:
   - Design key naming conventions for efficient lookups
   - Use composite keys to model relationships
   - Consider key distribution for sharding

```
# Redis key design patterns
user:1001:profile -> Hash with user profile data
user:1001:sessions -> Set of active session IDs
user:1001:followers -> Set of follower IDs
```

3. **Column-Family Stores (Cassandra, HBase)**:
   - Design row keys for efficient data distribution
   - Use composite keys for querying
   - Create multiple tables for different query patterns
   - Consider time bucketing for time-series data

```
-- Cassandra data modeling example
CREATE TABLE user_by_email (
  email TEXT PRIMARY KEY,
  user_id UUID,
  name TEXT
);

CREATE TABLE user_posts (
  user_id UUID,
  post_id TIMEUUID,
  content TEXT,
  created_at TIMESTAMP,
  PRIMARY KEY (user_id, post_id)
) WITH CLUSTERING ORDER BY (post_id DESC);
```

4. **Graph Databases (Neo4j)**:

- Focus on entity relationships as edges
- Use node labels to categorize entities
- Create relationship types to describe connections
- Keep properties on both nodes and relationships

```
// Neo4j data model
CREATE (u:User {userId: 'USER123', name: 'John'})
CREATE (p:Product {productId: 'PROD456', name: 'Smartphone'})
CREATE (u)-[:PURCHASED {date: '2023-06-15', amount: 699.99}]->(p)
```

NoSQL Data Modeling Process

1. **Identify Access Patterns**:
   - List all queries and operations the application will perform
   - Prioritize by frequency and importance
   - Document read vs. write ratios for each operation
2. **Design Around Query Patterns**:
   - Create initial data models based on primary access patterns
   - Consider denormalization to support efficient reads
   - Evaluate trade-offs between read and write efficiency
3. **Optimize for Write Patterns**:
   - Consider write frequency and volume
   - Evaluate impact of updates on data model
   - Design to minimize write contention
4. **Test with Realistic Data Volumes**:
   - Create performance tests with production-like data volumes
   - Verify that queries meet performance requirements
   - Identify and address bottlenecks
5. **Refine and Iterate**:
   - Adjust data model based on performance testing
   - Consider additional indices or data structures
   - Balance between read optimization and write overhead

Example: E-commerce Data Modeling

MongoDB Example:

```
// Customers collection
{
  "_id": ObjectId("..."),
  "customer_id": "CUST123",
```

```
  "email": "john@example.com",
  "name": "John Smith",
  "addresses": [
    {
      "type": "shipping",
      "street": "123 Main St",
      "city": "Boston",
      "state": "MA",
      "zip": "02101",
      "default": true
    },
    {
      "type": "billing",
      "street": "123 Main St",
      "city": "Boston",
      "state": "MA",
      "zip": "02101"
    }
  ],
  "payment_methods": [
    {
      "type": "credit_card",
      "last4": "1234",
      "expiry": "05/25",
      "default": true
    }
  ],
  "created_at": ISODate("2023-01-15T08:30:00Z")
}

// Products collection
{
  "_id": ObjectId("..."),
  "product_id": "PROD456",
  "name": "Smartphone XYZ",
  "description": "Latest smartphone with advanced features",
  "price": 699.99,
  "category": "Electronics",
  "subcategory": "Smartphones",
  "attributes": {
    "brand": "XYZ",
    "color": "Black",
    "storage": "128GB",
    "screen": "6.5 inch"
  },
  "inventory": {
    "in_stock": 120,
    "reserved": 5,
    "available": 115
  },
  "images": ["url1", "url2", "url3"]
}

// Orders collection
{
```

```
  "_id": ObjectId("..."),
  "order_id": "ORD789",
  "customer_id": "CUST123",
  "customer_info": {  // Extended reference
    "name": "John Smith",
    "email": "john@example.com"
  },
  "order_date": ISODate("2023-06-15T14:30:00Z"),
  "status": "shipped",
  "shipping_address": {
    "street": "123 Main St",
    "city": "Boston",
    "state": "MA",
    "zip": "02101"
  },
  "items": [
    {
      "product_id": "PROD456",
      "product_name": "Smartphone XYZ",  // Extended reference
      "quantity": 1,
      "price": 699.99,
      "subtotal": 699.99
    },
    {
      "product_id": "PROD789",
      "product_name": "Phone Case",  // Extended reference
      "quantity": 2,
      "price": 24.99,
      "subtotal": 49.98
    }
  ],
  "payment": {
    "method": "credit_card",
    "last4": "1234",
    "amount": 749.97,
    "status": "completed"
  },
  "subtotal": 749.97,
  "tax": 60.00,
  "shipping": 0.00,
  "total": 809.97,
  "shipping_info": {
    "carrier": "UPS",
    "tracking_number": "1Z999AA10123456784",
    "shipped_date": ISODate("2023-06-16T10:20:00Z"),
    "estimated_delivery": ISODate("2023-06-19T00:00:00Z")
  }
}
```

DynamoDB Example:

```
// Customers table
{
```

```
  "PK": "CUSTOMER#CUST123",
  "SK": "METADATA",
  "email": "john@example.com",
  "name": "John Smith",
  "created_at": "2023-01-15T08:30:00Z"
}

{
  "PK": "CUSTOMER#CUST123",
  "SK": "ADDRESS#SHIPPING#1",
  "street": "123 Main St",
  "city": "Boston",
  "state": "MA",
  "zip": "02101",
  "default": true
}

// Products table
{
  "PK": "PRODUCT#PROD456",
  "SK": "METADATA",
  "name": "Smartphone XYZ",
  "description": "Latest smartphone with advanced features",
  "price": 699.99,
  "category": "Electronics",
  "attributes": {
    "brand": "XYZ",
    "color": "Black",
    "storage": "128GB"
  }
}

{
  "PK": "PRODUCT#PROD456",
  "SK": "INVENTORY",
  "in_stock": 120,
  "reserved": 5,
  "available": 115,
  "last_updated": "2023-06-15T10:30:00Z"
}

// Orders + Order Items (single table design)
{
  "PK": "ORDER#ORD789",
  "SK": "METADATA",
  "customer_id": "CUST123",
  "order_date": "2023-06-15T14:30:00Z",
  "status": "shipped",
  "shipping_address": {
    "street": "123 Main St",
    "city": "Boston",
    "state": "MA",
    "zip": "02101"
  },
  "payment": {
```

```
    "method": "credit_card",
    "last4": "1234"
  },
  "subtotal": 749.97,
  "tax": 60.00,
  "shipping": 0.00,
  "total": 809.97
}

{
  "PK": "ORDER#ORD789",
  "SK": "ITEM#1",
  "product_id": "PROD456",
  "product_name": "Smartphone XYZ",
  "quantity": 1,
  "price": 699.99,
  "subtotal": 699.99
}

{
  "PK": "ORDER#ORD789",
  "SK": "ITEM#2",
  "product_id": "PROD789",
  "product_name": "Phone Case",
  "quantity": 2,
  "price": 24.99,
  "subtotal": 49.98
}

// Global Secondary Index (GSI) for customer orders
{
  "PK": "ORDER#ORD789",
  "SK": "METADATA",
  "GSI1PK": "CUSTOMER#CUST123",  // GSI partition key
  "GSI1SK": "ORDER#2023-06-15"   // GSI sort key (for time ordering)
}
```

Cassandra Example:

```
-- Customers by ID
CREATE TABLE customers (
    customer_id UUID PRIMARY KEY,
    email TEXT,
    name TEXT,
    created_at TIMESTAMP
);

-- Customers by email (for lookup)
CREATE TABLE customers_by_email (
    email TEXT PRIMARY KEY,
    customer_id UUID,
    name TEXT
);
```

```
-- Customer addresses
CREATE TABLE customer_addresses (
    customer_id UUID,
    address_id UUID,
    address_type TEXT,
    street TEXT,
    city TEXT,
    state TEXT,
    zip TEXT,
    is_default BOOLEAN,
    PRIMARY KEY (customer_id, address_id)
);

-- Products
CREATE TABLE products (
    product_id UUID PRIMARY KEY,
    name TEXT,
    description TEXT,
    price DECIMAL,
    category TEXT,
    subcategory TEXT,
    attributes MAP<TEXT, TEXT>
);

-- Product inventory
CREATE TABLE product_inventory (
    product_id UUID PRIMARY KEY,
    in_stock INT,
    reserved INT,
    available INT,
    last_updated TIMESTAMP
);

-- Orders
CREATE TABLE orders (
    order_id UUID,
    customer_id UUID,
    order_date TIMESTAMP,
    status TEXT,
    shipping_address_street TEXT,
    shipping_address_city TEXT,
    shipping_address_state TEXT,
    shipping_address_zip TEXT,
    payment_method TEXT,
    payment_last4 TEXT,
    subtotal DECIMAL,
    tax DECIMAL,
    shipping DECIMAL,
    total DECIMAL,
    PRIMARY KEY (order_id)
);

-- Order items
CREATE TABLE order_items (
```

```
    order_id UUID,
    item_id UUID,
    product_id UUID,
    product_name TEXT,
    quantity INT,
    price DECIMAL,
    subtotal DECIMAL,
    PRIMARY KEY (order_id, item_id)
);

-- Orders by customer (for querying a customer's orders)
CREATE TABLE orders_by_customer (
    customer_id UUID,
    order_date TIMESTAMP,
    order_id UUID,
    status TEXT,
    total DECIMAL,
    PRIMARY KEY ((customer_id), order_date, order_id)
) WITH CLUSTERING ORDER BY (order_date DESC, order_id ASC);
```

Interview Questions

Q: What are the key differences between data modeling for relational databases and NoSQL databases? How do query patterns influence NoSQL data modeling decisions?

A: Data modeling for relational and NoSQL databases involves fundamentally different approaches, principles, and priorities. Understanding these differences is crucial for effectively designing database schemas that leverage the strengths of each paradigm. Let me explore these differences and explain how query patterns drive NoSQL data modeling decisions.

Key Differences in Data Modeling Approaches

1. Normalization vs. Denormalization

Relational Approach:

- Emphasizes normalization (typically to 3NF or BCNF)
- Reduces data redundancy by organizing data into separate tables
- Maintains referential integrity through foreign keys
- Optimizes for storage efficiency and data consistency

```
-- Normalized relational model
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    name VARCHAR(100),
    email VARCHAR(100) UNIQUE
);

CREATE TABLE addresses (
    address_id INT PRIMARY KEY,
    customer_id INT REFERENCES customers(customer_id),
    street VARCHAR(200),
```

```
    city VARCHAR(100),
    state CHAR(2),
    zip VARCHAR(10),
    address_type VARCHAR(20)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT REFERENCES customers(customer_id),
    order_date TIMESTAMP,
    status VARCHAR(50)
);

CREATE TABLE order_items (
    order_id INT REFERENCES orders(order_id),
    product_id INT REFERENCES products(product_id),
    quantity INT,
    price DECIMAL(10,2),
    PRIMARY KEY (order_id, product_id)
);
```

NoSQL Approach:

- Embraces strategic denormalization
- Duplicates data to optimize query performance
- Focuses on aggregating related data into a single document/entity
- Optimizes for read performance and scalability

```
// Denormalized document model (MongoDB)
{
  "_id": "order123",
  "customer": {
    "id": "cust456",
    "name": "Jane Smith",
    "email": "jane@example.com"
  },
  "shipping_address": {
    "street": "123 Main St",
    "city": "Boston",
    "state": "MA",
    "zip": "02101"
  },
  "order_date": "2023-06-15T14:30:00Z",
  "status": "shipped",
  "items": [
    {
      "product_id": "prod789",
      "product_name": "Smartphone",
      "category": "Electronics",
      "quantity": 1,
      "price": 699.99
    }
  ],
```

```
  "total": 699.99
}
```

2. Schema Approach

Relational Approach:

- Fixed, predefined schema
- Schema changes require ALTER TABLE operations
- All rows in a table must conform to the same structure
- Schema enforced by the database system

```
-- Schema evolution in relational databases
ALTER TABLE customers ADD COLUMN phone_number VARCHAR(20);
ALTER TABLE customers ALTER COLUMN name VARCHAR(150); -- Change length
```

NoSQL Approach:

- Flexible, dynamic schema (especially in document databases)
- Schema changes can be handled at the application level
- Different documents in the same collection can have different structures
- Schema often enforced by the application, not the database

```
// Schema flexibility in document databases
// First document
db.customers.insert({
  name: 'John Doe',
  email: 'john@example.com',
});

// Second document with different fields
db.customers.insert({
  name: 'Jane Smith',
  email: 'jane@example.com',
  phone: '555-123-4567', // New field
  preferences: {
    // Nested structure
    language: 'English',
    notifications: true,
  },
});
```

3. Relationships and Joins

Relational Approach:

- Relationships expressed through foreign keys
- Data retrieved using JOIN operations

- Complex relationships handled through junction tables
- Designed for optimal query flexibility

```
-- Relational query with joins
SELECT o.order_id, c.name, p.product_name, oi.quantity, oi.price
FROM orders o
JOIN customers c ON o.customer_id = c.customer_id
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
WHERE o.order_date > '2023-01-01';
```

NoSQL Approach:

- Relationships handled through embedding or referencing
- Limited or no join support (application-performed joins)
- One-to-many relationships often embedded within parent documents
- Designed for horizontal scaling and partition tolerance

```
// MongoDB: Application-side join
const order = await db.orders.findOne({ _id: 'order123' });
const customer = await db.customers.findOne({ _id: order.customer_id });

// Alternative: Embedding relationship
const orderWithCustomer = await db.orders.findOne(
  { _id: 'order123' },
  { customer: 1, items: 1 }
);
```

4. Transaction Support

Relational Approach:

- Strong ACID transaction support
- Multi-table transactions are standard
- Optimized for data consistency
- Various isolation levels available

```
-- Multi-table transaction in SQL
BEGIN TRANSACTION;
  INSERT INTO orders (order_id, customer_id, order_date, status)
  VALUES (1001, 123, CURRENT_TIMESTAMP, 'processing');

  INSERT INTO order_items (order_id, product_id, quantity, price)
  VALUES (1001, 456, 1, 699.99);

  UPDATE inventory SET stock = stock - 1 WHERE product_id = 456;
COMMIT;
```

NoSQL Approach:

- Varies by database; often limited transaction support
- Many follow BASE principles (Basically Available, Soft state, Eventually consistent)
- Single-document transactions are common, multi-document less so
- Often sacrifices consistency for availability and partition tolerance

```
// MongoDB transaction (newer versions)
const session = client.startSession();
session.startTransaction();
try {
  await db.orders.insertOne({ ... }, { session });
  await db.inventory.updateOne(
    { product_id: "prod456" },
    { $inc: { stock: -1 } },
    { session }
  );
  await session.commitTransaction();
} catch (error) {
  await session.abortTransaction();
  throw error;
} finally {
  session.endSession();
}
```

5. Primary Design Focus

Relational Approach:

- Entity-centric design
- Starts with identifying entities and their attributes
- Focuses on representing the business domain accurately
- Optimizes for data integrity and query flexibility

```
Entity-Relationship Diagram Process:
1. Identify entities (Customer, Order, Product)
2. Define attributes for each entity
3. Establish relationships between entities
4. Normalize to reduce redundancy
5. Add constraints for data integrity
```

NoSQL Approach:

- Query-centric design
- Starts with identifying access patterns
- Focuses on optimizing for specific application needs
- Optimizes for performance, scalability, and particular workloads

```
NoSQL Design Process:
1. Identify query patterns and access needs
2. Design data models around those patterns
3. Denormalize strategically to optimize performance
4. Create multiple representations if needed for different access patterns
5. Consider distribution and partitioning needs
```

How Query Patterns Influence NoSQL Data Modeling

In NoSQL databases, the query patterns fundamentally drive the data model design. This "query-first" approach is one of the most significant mindset shifts when moving from relational to NoSQL modeling.

1. Embedding vs. Referencing Decisions

Query patterns directly determine whether related data should be embedded or referenced:

```
// Scenario 1: Customer and orders accessed together frequently
// Embedded approach (optimized for this query pattern)
{
  "customer_id": "cust123",
  "name": "John Smith",
  "email": "john@example.com",
  "orders": [
    {
      "order_id": "ord456",
      "date": "2023-06-15T14:30:00Z",
      "items": [...]
    }
  ]
}

// Scenario 2: Orders accessed independently, customer rarely needed
// Reference approach (optimized for different query pattern)
{
  "order_id": "ord456",
  "customer_id": "cust123", // Reference only
  "date": "2023-06-15T14:30:00Z",
  "items": [...]
}
```

The decision hinges on questions like:

- How often is the related data accessed together?
- What is the read-to-write ratio?
- How large is the related data?
- Is the relationship one-to-few or one-to-many?

2. Multiple Representations for Different Access Patterns

NoSQL often requires maintaining multiple representations of the same data to support different query patterns efficiently:

```
// DynamoDB example: Single-table design with multiple access patterns

// Access pattern 1: Get order by ID
{
  "PK": "ORDER#123",
  "SK": "METADATA",
  "customer_id": "456",
  "date": "2023-06-15",
  "status": "shipped"
}

// Access pattern 2: Get all orders for a customer
{
  "PK": "CUSTOMER#456",
  "SK": "ORDER#123",
  "date": "2023-06-15",
  "status": "shipped"
}

// Access pattern 3: Get all orders by date
{
  "PK": "DATE#2023-06-15",
  "SK": "ORDER#123",
  "customer_id": "456",
  "status": "shipped"
}
```

This approach duplicates data across different access patterns but optimizes query performance for each pattern.

3. Composite Keys for Range Queries

Query patterns requiring range operations influence key design:

```
-- Cassandra example: Designing for time-range queries

-- Query pattern: Get all sensor readings for a device within a time range
CREATE TABLE sensor_readings (
    sensor_id TEXT,
    date DATE,
    timestamp TIMESTAMP,
    temperature FLOAT,
    humidity FLOAT,
    PRIMARY KEY ((sensor_id, date), timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);

-- This structure allows queries like:
SELECT * FROM sensor_readings
WHERE sensor_id = 'device123'
AND date = '2023-06-15'
```

```
AND timestamp >= '2023-06-15 08:00:00'
AND timestamp <= '2023-06-15 17:00:00';
```

The composite key structure is directly influenced by the need to efficiently query time ranges for a specific sensor.

4. Denormalization for Query Performance

Specific query requirements dictate what data to denormalize:

```
// MongoDB example: Product catalog with reviews

// Query pattern: Display product with top 5 reviews
{
  "product_id": "prod123",
  "name": "Smartphone XYZ",
  "price": 699.99,
  "average_rating": 4.7, // Pre-computed aggregate
  "review_count": 142,   // Pre-computed aggregate
  "top_reviews": [       // Embedded subset for common query
    {
      "user": "Alice",
      "rating": 5,
      "comment": "Great product!",
      "date": "2023-05-20T10:15:00Z"
    },
    // More top reviews...
  ]
}

// All reviews stored separately for pagination/filtering
{
  "review_id": "rev456",
  "product_id": "prod123",
  "user": "Bob",
  "rating": 4,
  "comment": "Good but expensive",
  "date": "2023-06-10T14:30:00Z"
}
```

Here, the most common query pattern (show product with top reviews) drives the decision to embed top reviews and pre-compute aggregates.

5. Data Bucketing for Time-Series Data

Query patterns for time-series data influence how to bucket and partition data:

```
-- Cassandra time-series bucketing
-- Query pattern: Get hourly statistics for a specific sensor
```

```
-- Hourly bucketing for high-frequency sensors
CREATE TABLE sensor_readings_hourly (
    sensor_id TEXT,
    date DATE,
    hour INT,
    min_temp FLOAT,
    max_temp FLOAT,
    avg_temp FLOAT,
    sample_count INT,
    PRIMARY KEY ((sensor_id, date), hour)
);

-- Daily bucketing for low-frequency sensors
CREATE TABLE sensor_readings_daily (
    sensor_id TEXT,
    year INT,
    month INT,
    day INT,
    min_temp FLOAT,
    max_temp FLOAT,
    avg_temp FLOAT,
    sample_count INT,
    PRIMARY KEY ((sensor_id, year), month, day)
);
```

The time granularity of common queries directly influences the bucketing strategy.

6. Application-Side Joins and Materialized Views

When complex queries are needed, NoSQL databases often use application-side joins or materialized views:

```
// Application-side join example
async function getOrderWithDetails(orderId) {
  // Multiple queries to construct the complete view
  const order = await db.orders.findOne({ _id: orderId });
  const customer = await db.customers.findOne({ _id: order.customer_id });
  const items = await db.orderItems.find({ order_id: orderId }).toArray();

  // Combine in application
  return {
    ...order,
    customer: {
      name: customer.name,
      email: customer.email
    },
    items: items.map(item => ({
      ...item,
      total: item.price * item.quantity
    }))
  };
}

// Materialized view approach
```

```
// Pre-compute and store the joined view
{
  "_id": "order123",
  "date": "2023-06-15",
  "customer": {
    "id": "cust456",
    "name": "Jane Smith",
    "email": "jane@example.com"
  },
  "items": [
    {
      "product_id": "prod789",
      "name": "Smartphone",
      "quantity": 1,
      "price": 699.99,
      "total": 699.99
    }
  ],
  "total": 699.99
}
```

The complexity and frequency of joins drive the decision to either perform application-side joins or materialize the joined view.

Practical Example: E-commerce System

Let's compare relational vs. NoSQL modeling for an e-commerce system with these query patterns:

1. Display product details with specifications and inventory
2. Show customer order history with basic product information
3. Process new orders, updating inventory
4. Generate sales reports by category and time period

Relational Approach:

```
-- Products and their details
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    name VARCHAR(100),
    description TEXT,
    price DECIMAL(10,2),
    category_id INT REFERENCES categories(category_id)
);

CREATE TABLE product_specifications (
    product_id INT REFERENCES products(product_id),
    spec_name VARCHAR(50),
    spec_value VARCHAR(100),
    PRIMARY KEY (product_id, spec_name)
);

CREATE TABLE inventory (
    product_id INT PRIMARY KEY REFERENCES products(product_id),
```

```
    quantity INT,
    last_updated TIMESTAMP
);

-- Orders and their items
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    customer_id INT REFERENCES customers(customer_id),
    order_date TIMESTAMP,
    status VARCHAR(20)
);

CREATE TABLE order_items (
    order_id INT REFERENCES orders(order_id),
    product_id INT REFERENCES products(product_id),
    quantity INT,
    price DECIMAL(10,2),
    PRIMARY KEY (order_id, product_id)
);

-- Query for product details
SELECT p.*, i.quantity, array_agg(ps.spec_name || ': ' || ps.spec_value) as
specifications
FROM products p
JOIN inventory i ON p.product_id = i.product_id
LEFT JOIN product_specifications ps ON p.product_id = ps.product_id
WHERE p.product_id = 123
GROUP BY p.product_id, i.quantity;

-- Query for customer order history
SELECT o.order_id, o.order_date, o.status,
       oi.product_id, p.name, oi.quantity, oi.price
FROM orders o
JOIN order_items oi ON o.order_id = oi.order_id
JOIN products p ON oi.product_id = p.product_id
WHERE o.customer_id = 456
ORDER BY o.order_date DESC;
```

NoSQL Approach (MongoDB):

```
// Products collection optimized for product display
{
  "_id": "prod123",
  "name": "Smartphone XYZ",
  "description": "Latest smartphone model",
  "price": 699.99,
  "category": "Electronics",
  "specifications": {
    "screen": "6.5 inch",
    "processor": "Snapdragon 888",
    "ram": "8GB",
    "storage": "128GB"
  },
```

```
  "inventory": {
    "quantity": 42,
    "last_updated": "2023-06-14T10:30:00Z"
  },
  "images": ["url1", "url2"]
}

// Orders collection optimized for order history display
{
  "_id": "order789",
  "customer_id": "cust456",
  "order_date": "2023-06-15T14:30:00Z",
  "status": "shipped",
  "items": [
    {
      "product_id": "prod123",
      "name": "Smartphone XYZ",  // Denormalized product name
      "quantity": 1,
      "price": 699.99
    },
    {
      "product_id": "prod456",
      "name": "Phone Case",      // Denormalized product name
      "quantity": 2,
      "price": 24.99
    }
  ],
  "total": 749.97,
  "shipping_address": {
    "street": "123 Main St",
    "city": "Boston",
    "state": "MA",
    "zip": "02101"
  }
}

// Customer collection with references to orders
{
  "_id": "cust456",
  "name": "Jane Smith",
  "email": "jane@example.com",
  "addresses": [
    {
      "type": "shipping",
      "street": "123 Main St",
      "city": "Boston",
      "state": "MA",
      "zip": "02101",
      "default": true
    }
  ],
  "recent_orders": [  // Limited embedded subset for quick access
    {
      "order_id": "order789",
      "date": "2023-06-15T14:30:00Z",
```

```
      "total": 749.97,
      "status": "shipped"
    }
  ]
}

// Sales aggregations collection for reporting
{
  "_id": "sales:2023:06:electronics",
  "year": 2023,
  "month": 6,
  "category": "Electronics",
  "total_sales": 42789.50,
  "order_count": 57,
  "product_counts": {
    "prod123": 48,
    "prod456": 36
  },
  "daily_sales": [
    { "day": 1, "amount": 1245.67 },
    { "day": 2, "amount": 2456.78 },
    // ...
  ]
}
```

Query Pattern Influence:

1. **Product Display Query**:
   - Relational approach requires joining 3 tables
   - NoSQL embeds specifications and inventory in the product document
   - Result: Single document read vs. complex join
2. **Order History Query**:
   - Relational approach joins orders, items, and products
   - NoSQL embeds item details in the order document
   - Result: Query simplification and performance improvement
3. **Order Processing**:
   - Relational approach requires multi-table transaction
   - NoSQL requires updating product document and creating order document
   - Result: Trade-off between transactional guarantees and write performance
4. **Sales Reporting**:
   - Relational approach requires complex aggregation queries
   - NoSQL uses pre-aggregated data updated incrementally
   - Result: Read optimization at the cost of write complexity

Summary of Key Principles for NoSQL Data Modeling

1. **Start with query patterns**: Identify all access patterns before designing the data model

2. **Optimize for the common case**: Design for the most frequent queries, even if it means duplication
3. **Denormalize strategically**: Duplicate data where it improves read performance for critical queries
4. **Consider write patterns too**: Balance read optimization with write efficiency
5. **Create multiple representations**: Use different structures for different access patterns
6. **Manage duplication carefully**: Implement strategies to maintain consistency across duplicated data
7. **Think about distribution**: Design keys that distribute data evenly and support efficient queries
8. **Plan for evolution**: Consider how the data model will accommodate changing requirements

The fundamental principle is that in NoSQL databases, the data model serves the application's query needs, rather than attempting to represent the business domain in a normalized form. This query-first approach leads to very different data structures than what would be optimal in a relational database, but enables much better performance and scalability for specific access patterns.

Q: How would you design a DynamoDB schema for a social media application that needs to support user profiles, posts, comments, and follows? What considerations would drive your design decisions?

A: Designing a DynamoDB schema for a social media application requires carefully considering access patterns, performance needs, and DynamoDB's specific characteristics and constraints. I'll walk through a comprehensive approach to designing this schema, explaining key considerations at each step.

Understanding DynamoDB's Core Concepts

Before diving into the design, let's review key DynamoDB concepts that influence our schema decisions:

1. **Single-Table Design**: Unlike relational databases, DynamoDB typically uses a single table for multiple entity types, leveraging composite primary keys and secondary indexes.
2. **Partition Key and Sort Key**: The primary key consists of a partition key (determining data distribution) and an optional sort key (ordering items within a partition).
3. **Global Secondary Indexes (GSIs)**: Additional indexes that can have different partition and sort keys from the base table.
4. **Local Secondary Indexes (LSIs)**: Alternative sort keys that share the same partition key as the base table.
5. **Item Size Limit**: 400KB maximum size for any single item.
6. **No Joins**: DynamoDB doesn't support joins; related data must be denormalized or retrieved with multiple queries.
7. **Read/Write Capacity**: Provisioned or on-demand capacity is allocated at the table and GSI level.

Identifying Access Patterns

For a social media application, here are the critical access patterns we need to support:

1. **User Profiles**:
    - Get user profile by user ID

- Get user profile by username (lookup)
- Update user profile
- Check if username is available

2. **Posts**:
   - Create new post
   - Get post by post ID
   - Get all posts by a specific user
   - Get user's feed (posts from followed users)
   - Get trending/popular posts
3. **Comments**:
   - Add comment to a post
   - Get all comments for a post
   - Get comments made by a specific user
4. **Social Graph**:
   - Follow/unfollow a user
   - Get followers of a user
   - Get users followed by a user
   - Check if user A follows user B
5. **Notifications**:
   - Get user's notifications
   - Mark notifications as read

Schema Design Approach

I'll use a single-table design with careful selection of primary keys and secondary indexes to efficiently support these access patterns.

Base Table Design

```
Primary Key:
- Partition Key (PK): Entity type and ID (e.g., "USER#123", "POST#456")
- Sort Key (SK): Entity metadata or relationship (e.g., "METADATA", "FOLLOWS#789")
```

Entity Types and Key Patterns

1. **User Entity**:
   - PK: "USER#[user_id]"
   - SK: "METADATA"
2. **Post Entity**:
   - PK: "POST#[post_id]"

- SK: "METADATA"

3. **User Post**:
   - PK: "USER#[user_id]"
   - SK: "POST#[timestamp]#[post_id]"
4. **Comment Entity**:
   - PK: "POST#[post_id]"
   - SK: "COMMENT#[timestamp]#[comment_id]"
5. **Follow Relationship**:
   - PK: "USER#[follower_id]"
   - SK: "FOLLOWS#[followed_id]"
6. **Follower Relationship**:
   - PK: "USER#[followed_id]"
   - SK: "FOLLOWER#[follower_id]"
7. **Notification**:
   - PK: "USER#[user_id]"
   - SK: "NOTIFICATION#[timestamp]#[notification_id]"

Global Secondary Indexes (GSIs)

1. **GSI1: Username Index**
   - GSI1PK: "USERNAME#[username]"
   - GSI1SK: "USER#[user_id]"
2. **GSI2: Inverted Index**
   - GSI2PK: SK (original sort key)
   - GSI2SK: PK (original partition key)
3. **GSI3: Feed Index**
   - GSI3PK: "FEED#[user_id]"
   - GSI3SK: "[timestamp]#[post_id]"

Detailed Schema Design

Let's implement this design with sample items for each entity and access pattern:

User Profiles

```
{
  "PK": "USER#123",
  "SK": "METADATA",
  "user_id": "123",
```

```
  "username": "johndoe",
  "display_name": "John Doe",
  "email": "john@example.com",
  "bio": "Social media enthusiast",
  "profile_image": "https://example.com/images/123.jpg",
  "created_at": "2023-01-15T14:30:00Z",
  "follower_count": 42,
  "following_count": 35,
  "post_count": 128,
  "GSI1PK": "USERNAME#johndoe",
  "GSI1SK": "USER#123"
}
```

Access Pattern Implementation:

- Get user by user ID: Query on PK="USER#123", SK="METADATA"
- Get user by username: Query GSI1 with GSI1PK="USERNAME#johndoe"
- Check username availability: Query GSI1 with GSI1PK="USERNAME#newname"

Posts

```
// Post metadata
{
  "PK": "POST#456",
  "SK": "METADATA",
  "post_id": "456",
  "user_id": "123",
  "username": "johndoe",
  "display_name": "John Doe",
  "content": "This is my awesome post about DynamoDB!",
  "image_urls": ["https://example.com/images/post456_1.jpg"],
  "created_at": "2023-06-15T10:30:00Z",
  "like_count": 25,
  "comment_count": 8,
  "GSI2PK": "METADATA",
  "GSI2SK": "POST#456"
}

// Post tied to user (for user's posts)
{
  "PK": "USER#123",
  "SK": "POST#20230615103000#456",
  "post_id": "456",
  "content": "This is my awesome post about DynamoDB!",
  "created_at": "2023-06-15T10:30:00Z",
  "like_count": 25,
  "comment_count": 8,
  "GSI2PK": "POST#20230615103000#456",
  "GSI2SK": "USER#123",
  "GSI3PK": "FEED#123",
  "GSI3SK": "20230615103000#456"
}
```

Access Pattern Implementation:

- Get post by ID: Query on PK="POST#456", SK="METADATA"
- Get all posts by user: Query on PK="USER#123", SK begins_with "POST#"
- Get user's feed: Query GSI3 for multiple followed users (requires application logic)

Comments

```
{
  "PK": "POST#456",
  "SK": "COMMENT#20230615113000#789",
  "comment_id": "789",
  "user_id": "124",
  "username": "janedoe",
  "display_name": "Jane Doe",
  "content": "Great post about DynamoDB!",
  "created_at": "2023-06-15T11:30:00Z",
  "GSI2PK": "COMMENT#20230615113000#789",
  "GSI2SK": "POST#456"
}

// User's comments (optional, for user's comment history)
{
  "PK": "USER#124",
  "SK": "COMMENT#20230615113000#789",
  "post_id": "456",
  "comment_id": "789",
  "content": "Great post about DynamoDB!",
  "created_at": "2023-06-15T11:30:00Z",
  "GSI2PK": "COMMENT#20230615113000#789",
  "GSI2SK": "USER#124"
}
```

Access Pattern Implementation:

- Add comment to post: Put item with PK="POST#456", SK="COMMENT#[timestamp]#[id]"
- Get all comments for post: Query on PK="POST#456", SK begins_with "COMMENT#"
- Get comments by user: Query on PK="USER#124", SK begins_with "COMMENT#"

Social Graph (Follows/Followers)

```
// User 123 follows user 125
{
  "PK": "USER#123",
  "SK": "FOLLOWS#125",
  "followed_id": "125",
  "followed_username": "alicesmith",
  "followed_at": "2023-05-20T09:15:00Z",
  "GSI2PK": "FOLLOWS#125",
  "GSI2SK": "USER#123"
```

```
}

// User 125 has follower 123
{
  "PK": "USER#125",
  "SK": "FOLLOWER#123",
  "follower_id": "123",
  "follower_username": "johndoe",
  "followed_at": "2023-05-20T09:15:00Z",
  "GSI2PK": "FOLLOWER#123",
  "GSI2SK": "USER#125"
}
```

Access Pattern Implementation:

- Follow a user: Put two items (FOLLOWS and FOLLOWER)
- Unfollow a user: Delete two items
- Get users followed by user: Query PK="USER#123", SK begins_with "FOLLOWS#"
- Get followers of user: Query PK="USER#125", SK begins_with "FOLLOWER#"
- Check if user follows another: Get item with PK="USER#123", SK="FOLLOWS#125"

Feed Generation

For the feed, we have two implementation options:

1. On-demand feed generation (query-time):

```
// Pseudocode for generating feed
async function generateFeed(userId) {
  // Get all users that this user follows
  const follows = await queryItems({
    KeyConditionExpression: 'PK = :pk AND begins_with(SK, :sk)',
    ExpressionAttributeValues: {
      ':pk': `USER#${userId}`,
      ':sk': 'FOLLOWS#',
    },
  });

  // Get recent posts from each followed user
  const followedUserIds = follows.map((f) => f.followed_id);
  const feedItems = [];

  for (const followedId of followedUserIds) {
    const posts = await queryItems({
      KeyConditionExpression: 'PK = :pk AND begins_with(SK, :sk)',
      ExpressionAttributeValues: {
        ':pk': `USER#${followedId}`,
        ':sk': 'POST#',
      },
      Limit: 10, // Get 10 most recent posts
    });
    feedItems.push(...posts);
```

```
  }

  // Sort by timestamp (descending)
  return feedItems
    .sort((a, b) => {
      const aTime = a.SK.split('#')[1];
      const bTime = b.SK.split('#')[1];
      return bTime.localeCompare(aTime);
    })
    .slice(0, 50); // Return top 50
}
```

2. Pre-computed feed (write-time):

```
// Feed item written when user creates a post
{
  "PK": "USER#124", // Follower
  "SK": "FEED#20230615103000#456",
  "post_id": "456",
  "user_id": "123",
  "username": "johndoe",
  "content": "This is my awesome post about DynamoDB!",
  "created_at": "2023-06-15T10:30:00Z",
  "GSI3PK": "FEED#124",
  "GSI3SK": "20230615103000#456"
}
```

Access Pattern Implementation:

- Get user's feed: Query GSI3 with GSI3PK="FEED#124", sorted by GSI3SK

Notifications

```
{
  "PK": "USER#123",
  "SK": "NOTIFICATION#20230615120000#abc",
  "notification_id": "abc",
  "type": "like",
  "actor_id": "125",
  "actor_username": "alicesmith",
  "content": "Alice Smith liked your post",
  "reference_id": "456", // Post ID
  "created_at": "2023-06-15T12:00:00Z",
  "read": false,
  "GSI2PK": "NOTIFICATION#20230615120000#abc",
  "GSI2SK": "USER#123"
}
```

Access Pattern Implementation:

- Get user's notifications: Query PK="USER#123", SK begins_with "NOTIFICATION#"
- Mark notification as read: Update item with PK="USER#123", SK="NOTIFICATION#[timestamp]#[id]"

Key Design Considerations

Here are the critical considerations that drove this design:

1. Item Collection Size Limits

DynamoDB limits each partition to 10GB. For high-scale users (celebrities with millions of followers/posts), we need strategies to avoid hitting this limit:

```
// For high-scale accounts, partition by time period
// Instead of USER#123 for all posts, use:
{
  "PK": "USER#123#2023Q2", // Quarterly partitioning
  "SK": "POST#20230615103000#456",
  // ...
}
```

2. Efficient Feed Generation

The feed implementation is particularly challenging. Options to consider:

1. **Query-time approach**: Query posts from all followed users at read time
    - Pros: Always up-to-date, no duplicated storage
    - Cons: Slow for users following many accounts, requires multiple queries
2. **Write-time approach**: Fan-out posts to followers' feeds when posted
    - Pros: Fast reads for feed, single query
    - Cons: Storage duplication, challenging for high-follower accounts
3. **Hybrid approach**: Fan-out for regular users, query-time for high-follower accounts
    - Balance between read performance and write efficiency
    - Requires application logic to determine approach

For this design, I've included both options, but the hybrid approach would be most appropriate for a real-world implementation.

3. Counter Management

For counts like likes, comments, followers:

```
// Atomic counter updates
await dynamoDb.update({
  TableName: 'SocialMedia',
  Key: { PK: 'USER#123', SK: 'METADATA' },
  UpdateExpression: 'SET follower_count = follower_count + :val',
```

```
  ExpressionAttributeValues: { ':val': 1 },
});
```

For high-contention counters (viral posts), consider:

```
// Sharded counter approach
const shardId = Math.floor(Math.random() * 10); // 10 shards
await dynamoDb.update({
  TableName: 'SocialMedia',
  Key: { PK: `POST#456#COUNTER#${shardId}`, SK: 'LIKES' },
  UpdateExpression: 'SET count = count + :val',
  ExpressionAttributeValues: { ':val': 1 },
});

// Reading requires aggregating all shards
```

4. Managing Heavy Access Patterns

For heavily accessed items like trending posts:

1. **Caching layer**: Use DAX or application-level caching
2. **Read replicas**: Consider DynamoDB Global Tables
3. **Time partitioning**: Segment data by time periods

```
// Read distribution for trending posts
const hour = new Date().getHours();
const shardId = hour % 4; // Distribute by hour of day

const trendingPosts = await dynamoDb.query({
  TableName: 'SocialMedia',
  IndexName: 'TrendingIndex',
  KeyConditionExpression: 'GSI4PK = :pk',
  ExpressionAttributeValues: {
    ':pk': `TRENDING#${shardId}`,
  },
});
```

5. Transaction Support

For operations requiring atomicity across items:

```
// Atomic follow/unfollow operation
await dynamoDb.transactWrite({
  TransactItems: [
    {
      Put: {
        TableName: 'SocialMedia',
        Item: {
```

```
          PK: 'USER#123',
          SK: 'FOLLOWS#125',
          // other attributes
        },
      },
    },
    {
      Put: {
        TableName: 'SocialMedia',
        Item: {
          PK: 'USER#125',
          SK: 'FOLLOWER#123',
          // other attributes
        },
      },
    },
    {
      Update: {
        TableName: 'SocialMedia',
        Key: { PK: 'USER#123', SK: 'METADATA' },
        UpdateExpression: 'SET following_count = following_count + :val',
        ExpressionAttributeValues: { ':val': 1 },
      },
    },
    {
      Update: {
        TableName: 'SocialMedia',
        Key: { PK: 'USER#125', SK: 'METADATA' },
        UpdateExpression: 'SET follower_count = follower_count + :val',
        ExpressionAttributeValues: { ':val': 1 },
      },
    },
  ],
});
```

6. GSI Projection Optimization

For secondary indexes, carefully choose projected attributes to minimize cost and improve performance:

```
// GSI definition with projection
{
  "AttributeName": "GSI1PK",
  "KeyType": "HASH"
},
{
  "AttributeName": "GSI1SK",
  "KeyType": "RANGE"
},
{
  "IndexName": "UsernameIndex",
  "KeySchema": [
    { "AttributeName": "GSI1PK", "KeyType": "HASH" },
    { "AttributeName": "GSI1SK", "KeyType": "RANGE" }
```

```
    ],
    "Projection": {
      "ProjectionType": "INCLUDE",
      "NonKeyAttributes": [
        "username", "display_name", "email", "profile_image"
      ]
    }
  }
```

Query Examples for Key Access Patterns

Here are concrete examples of the most important queries:

1. Get User Profile

```
// By user ID
const user = await dynamoDb.get({
  TableName: 'SocialMedia',
  Key: {
    PK: 'USER#123',
    SK: 'METADATA',
  },
});

// By username
const userByUsername = await dynamoDb.query({
  TableName: 'SocialMedia',
  IndexName: 'GSI1',
  KeyConditionExpression: 'GSI1PK = :username',
  ExpressionAttributeValues: {
    ':username': 'USERNAME#johndoe',
  },
});
```

2. Get User's Posts

```
const userPosts = await dynamoDb.query({
  TableName: 'SocialMedia',
  KeyConditionExpression: 'PK = :userId AND begins_with(SK, :prefix)',
  ExpressionAttributeValues: {
    ':userId': 'USER#123',
    ':prefix': 'POST#',
  },
  ScanIndexForward: false, // Descending order (newest first)
  Limit: 20,
});
```

3. Get Post with Comments

```
// Get post
const post = await dynamoDb.get({
  TableName: 'SocialMedia',
  Key: {
    PK: 'POST#456',
    SK: 'METADATA',
  },
});

// Get comments
const comments = await dynamoDb.query({
  TableName: 'SocialMedia',
  KeyConditionExpression: 'PK = :postId AND begins_with(SK, :prefix)',
  ExpressionAttributeValues: {
    ':postId': 'POST#456',
    ':prefix': 'COMMENT#',
  },
  ScanIndexForward: true, // Ascending order (oldest first)
});
```

4. Get User's Feed (Pre-computed Approach)

```
const feed = await dynamoDb.query({
  TableName: 'SocialMedia',
  IndexName: 'GSI3',
  KeyConditionExpression: 'GSI3PK = :feedId',
  ExpressionAttributeValues: {
    ':feedId': 'FEED#123',
  },
  ScanIndexForward: false, // Descending order (newest first)
  Limit: 50,
});
```

5. Check if User A Follows User B

```
const relationship = await dynamoDb.get({
  TableName: 'SocialMedia',
  Key: {
    PK: 'USER#123',
    SK: 'FOLLOWS#125',
  },
});

const isFollowing = !!relationship.Item;
```

Scaling and Performance Considerations

1. **Write Capacity Planning**:

- Identify write-heavy operations (post creation, likes)
- Consider using on-demand capacity for spiky workloads
- Monitor consumed capacity and hot partitions

2. **Read Capacity Planning**:
   - Feeds and popular content will dominate reads
   - Consider caching frequently accessed items
   - Use consistent reads only when necessary

3. **Data Lifecycle Management**:
   - Implement TTL for temporary data like notifications
   - Archive old posts and comments to S3 for cost optimization
   - Consider periodic aggregation of historical data

4. **Cost Optimization**:
   - Minimize secondary index count and projections
   - Use sparse indexes where possible
   - Compress large text fields (content, descriptions)

Implementation Challenges and Solutions

1. **Challenge: Celebrity Problem (Users with Millions of Followers)**

   Solution: Use a hybrid approach for feed generation

```
async function addPostToFeeds(userId, postId, postData) {
  const followers = await getFollowers(userId);

  if (followers.length > CELEBRITY_THRESHOLD) {
    // For celebrities, don't fan out
    // Just mark this as a celebrity post
    await dynamoDb.put({
      TableName: 'SocialMedia',
      Item: {
        PK: 'CELEBRITY#POST',
        SK: `${userId}#${postId}`,
        ...postData,
      },
    });
  } else {
    // Fan out to all followers' feeds
    const writeRequests = followers.map((followerId) => ({
      PutRequest: {
        Item: {
          PK: `USER#${followerId}`,
          SK: `FEED#${timestamp}#${postId}`,
          GSI3PK: `FEED#${followerId}`,
          GSI3SK: `${timestamp}#${postId}`,
          ...postData,
        },
      },
```

```
    }));

    // Batch writes in chunks of 25 (DynamoDB limit)
    await batchWriteInChunks(writeRequests);
  }
}

async function getUserFeed(userId) {
  // Get pre-computed feed items
  const regularFeed = await getFeedItems(userId);

  // Get posts from celebrities this user follows
  const celebritiesFollowed = await getCelebritiesFollowed(userId);
  const celebrityPosts = await getCelebrityPosts(celebritiesFollowed);

  // Merge and sort both feeds
  return mergeSortFeedItems(regularFeed, celebrityPosts);
}
```

2. **Challenge: Comment Threading and Pagination**

   Solution: Use hierarchical sort keys

```
// Parent comment
{
  "PK": "POST#456",
  "SK": "COMMENT#20230615120000#789",
  "content": "Great post!"
}

// Reply to comment
{
  "PK": "POST#456",
  "SK": "COMMENT#20230615120000#789#REPLY#20230615130000#790",
  "content": "Thanks for your feedback!"
}
```

3. **Challenge: Real-time Notifications**

   Solution: Combine DynamoDB with WebSockets or SNS

```
async function createNotification(userId, notificationData) {
  // Store in DynamoDB
  await dynamoDb.put({
    TableName: 'SocialMedia',
    Item: {
      PK: `USER#${userId}`,
      SK: `NOTIFICATION#${timestamp}#${notificationId}`,
      ...notificationData,
    },
  });
```

```
  // Publish to SNS for real-time delivery
  await sns.publish({
    TopicArn: `arn:aws:sns:region:account:user-${userId}-notifications`,
    Message: JSON.stringify(notificationData),
  });
}
```

Summary and Best Practices

1. **Design Around Access Patterns**:
   - Start by identifying and prioritizing access patterns
   - Design key schema to efficiently support these patterns
   - Use GSIs strategically for alternative access paths
2. **Effective Partitioning**:
   - Choose partition keys that distribute data evenly
   - Avoid hot partitions through time-based or sharded keys
   - Consider workload characteristics when designing partitions
3. **Optimize for Cost and Performance**:
   - Minimize the number of requests per operation
   - Use the right capacity mode for your workload
   - Implement appropriate caching strategies
   - Consider DAX for read-heavy workloads
4. **Data Modeling Best Practices**:
   - Denormalize data strategically to optimize access patterns
   - Use composite sort keys for hierarchical data
   - Design with future growth in mind
   - Implement version attributes for schema evolution
5. **Maintenance Strategy**:
   - Plan for data archiving and lifecycle management
   - Implement monitoring for capacity and performance
   - Regularly review and optimize your data model as usage patterns evolve

This DynamoDB schema design for a social media application balances performance, cost, and complexity to efficiently support the required access patterns while addressing common challenges in social media workloads. The single-table design with carefully selected keys and indexes makes it possible to implement a complex application with minimal database overhead and maximum scalability.

4.5 Consistency Models

Consistency models define how database systems handle concurrent operations and maintain data integrity across distributed systems.

CAP Theorem

The CAP theorem states that a distributed database system can only guarantee two out of three properties simultaneously:

1. **Consistency**: All nodes see the same data at the same time
2. **Availability**: Every request receives a response (success or failure)
3. **Partition Tolerance**: The system continues to operate despite network partitions

Database systems make different trade-offs among these properties:

- **CP Systems**: Prioritize consistency and partition tolerance over availability
- **AP Systems**: Prioritize availability and partition tolerance over consistency
- **CA Systems**: Prioritize consistency and availability over partition tolerance (rare in distributed systems)

Types of Consistency Models

1. **Strong Consistency**:
    - All reads reflect the most recent write
    - All nodes see the same data at the same time
    - Examples: HBase, MongoDB (with appropriate settings), relational databases
2. **Eventual Consistency**:
    - Given enough time without updates, all replicas will converge
    - Different nodes may return different values temporarily
    - Examples: Cassandra (with lower consistency levels), DynamoDB (default)
3. **Causal Consistency**:
    - Operations causally related appear in the same order to all nodes
    - Independent operations may be seen in different orders
    - Examples: MongoDB (causal consistency sessions)
4. **Session Consistency**:
    - Within a session, reads reflect previous writes
    - Different sessions may see different states
    - Examples: DynamoDB (with consistent reads)
5. **Monotonic Read Consistency**:
    - If a process reads a value, subsequent reads will return that value or a newer one
    - Examples: Various databases with client-side tracking
6. **Read-your-writes Consistency**:
    - A client always sees its own writes
    - Examples: Cassandra (with LOCAL_ONE for writes and LOCAL_QUORUM for reads)

Consistency in NoSQL Databases

Cassandra Consistency Levels:

```
// Setting consistency levels in Cassandra
Statement statement = new SimpleStatement("SELECT * FROM users WHERE user_id =
123");

// Strong consistency for a critical read
statement.setConsistencyLevel(ConsistencyLevel.ALL);

// Eventual consistency for a non-critical read
statement.setConsistencyLevel(ConsistencyLevel.ONE);

// Quorum-based consistency for a balanced approach
statement.setConsistencyLevel(ConsistencyLevel.QUORUM);

// Write consistency
PreparedStatement insertStmt = session.prepare("INSERT INTO users (user_id, name)
VALUES (?, ?)");
BoundStatement boundStatement = insertStmt.bind(123, "John Doe");
boundStatement.setConsistencyLevel(ConsistencyLevel.QUORUM);
```

MongoDB Read and Write Concerns:

```
// Strong consistency
db.collection.find().readConcern("majority");
db.collection.insertOne({...}, {writeConcern: {w: "majority"}});

// Local consistency (eventual)
db.collection.find().readConcern("local");
db.collection.insertOne({...}, {writeConcern: {w: 1}});

// Causal consistency
const session = db.getMongo().startSession({causalConsistency: true});
session.startTransaction();
const coll = session.getDatabase("mydb").getCollection("mycoll");
coll.insertOne({...});
const result = coll.find({...}).toArray();
session.commitTransaction();
```

DynamoDB Consistency Options:

```
// Eventually consistent read (default)
const params = {
  TableName: 'Products',
  Key: {
    ProductId: '123',
  },
};

// Strongly consistent read
const params = {
  TableName: 'Products',
```

```
  Key: {
    ProductId: '123',
  },
  ConsistentRead: true,
};
```

Conflict Resolution Strategies

When conflicts occur in distributed systems, various strategies are used to resolve them:

1. **Last-Write-Wins (LWW)**:
   - The update with the latest timestamp wins
   - Simple but may lose data
   - Used by Cassandra by default
2. **Vector Clocks**:
   - Track causality between different versions
   - More complex but preserves causality
   - Used by Riak and some other distributed systems
3. **Custom Conflict Resolution**:
   - Application-defined logic to resolve conflicts
   - Most flexible but requires custom implementation
   - Used in advanced distributed database configurations

```
// Custom conflict resolution in DynamoDB Streams
exports.handler = async (event) => {
  for (const record of event.Records) {
    if (record.eventName === 'MODIFY') {
      const newImage = AWS.DynamoDB.Converter.unmarshall(
        record.dynamodb.NewImage
      );
      const oldImage = AWS.DynamoDB.Converter.unmarshall(
        record.dynamodb.OldImage
      );

      // Implement custom conflict resolution logic
      if (isConflict(oldImage, newImage)) {
        const resolvedData = resolveConflict(oldImage, newImage);

        // Update with resolved data
        await dynamoDb.put({
          TableName: record.eventSourceARN.split('/')[1],
          Item: AWS.DynamoDB.Converter.marshall(resolvedData),
        });
      }
    }
  }
```

```
    return { status: 'success' };
  };
```

Interview Questions

Q: How do different consistency models in NoSQL databases affect application design and performance? When would you choose a weaker consistency model over strong consistency?

A: Consistency models in NoSQL databases represent different guarantees about when and how updates become visible to readers. These models profoundly affect application design, performance, availability, and scalability. Understanding these trade-offs is crucial for designing distributed systems that meet both functional and non-functional requirements.

Understanding Consistency Models in NoSQL Databases

Strong Consistency

Definition: All reads reflect the most recent write, regardless of which replica is accessed.

Characteristics:

- All nodes see the same data at the same time
- Reads always return the latest committed write
- Behaves like a single-node system despite distribution

Examples:

- MongoDB with `{writeConcern: {w: "majority"}, readConcern: "majority"}`
- DynamoDB with `ConsistentRead: true`
- HBase (always strongly consistent)
- Redis (single node)

Implementation:

```
// MongoDB strong consistency example
const session = client.startSession();
try {
  const options = {
    readConcern: { level: 'majority' },
    writeConcern: { w: 'majority' },
  };

  session.startTransaction(options);

  await db
    .collection('accounts')
    .updateOne(
      { _id: fromAccountId },
      { $inc: { balance: -amount } },
      { session }
    );
```

```
    await db
      .collection('accounts')
      .updateOne(
        { _id: toAccountId },
        { $inc: { balance: amount } },
        { session }
      );

    await session.commitTransaction();
  } catch (error) {
    await session.abortTransaction();
    throw error;
  } finally {
    session.endSession();
  }
```

Eventual Consistency

Definition: Given sufficient time without updates, all replicas will converge to the same state.

Characteristics:

- Replicas may temporarily provide different values
- Updates propagate asynchronously to all replicas
- No guarantees on when convergence occurs
- Higher availability and lower latency than strong consistency

Examples:

- Amazon DynamoDB (default read behavior)
- Cassandra with consistency level ONE
- MongoDB with `{readConcern: "local"}` or with `{readPreference: "primary"}`

Implementation:

```
// DynamoDB eventual consistency example
const params = {
  TableName: 'Products',
  Key: {
    ProductId: '123',
  },
  // No ConsistentRead parameter (defaults to false)
};

// Read from DynamoDB with eventual consistency
const result = await dynamoDb.get(params).promise();
```

Causal Consistency

Definition: Operations causally related are seen in the same order by all nodes, but concurrent operations may be seen in different orders.

Characteristics:

- Preserves cause-effect relationships
- Weaker than strong consistency, stronger than eventual
- Ensures writes made during a session are visible to reads in that session
- Requires tracking causal relationships

Examples:

- MongoDB causal consistency sessions
- Some distributed databases with vector clocks

Implementation:

```
// MongoDB causal consistency example
const session = client.startSession({ causalConsistency: true });

// First operation
await db
  .collection('blogs')
  .insertOne({ title: 'New Post', content: '...' }, { session });

// Second operation that depends on the first
// This operation will see the results of the first, even on different servers
const posts = await db
  .collection('blogs')
  .find({})
  .sort({ _id: -1 })
  .limit(10)
  .session(session)
  .toArray();
```

Session Consistency

Definition: Within a single client session, reads reflect all previous writes, but different sessions may see different views.

Characteristics:

- Simpler to implement than causal consistency
- Guarantees read-your-writes consistency within a session
- No guarantees across different sessions
- Good compromise between performance and usability

Examples:

- DynamoDB Global Tables with session stickiness
- Cassandra with session-aware drivers

Implementation:

```
// Cassandra session consistency example
CqlSession session = CqlSession.builder()
    .withKeyspace("mykeyspace")
    .build();

// Write in a session
session.execute(
    "INSERT INTO users (user_id, name, email) VALUES (?, ?, ?)",
    UUID.randomUUID(), "John Doe", "john@example.com"
);

// Read in the same session (will see the write)
ResultSet rs = session.execute(
    "SELECT * FROM users WHERE user_id = ?", user_id
);
```

Monotonic Read Consistency

Definition: Once a process reads a value, all subsequent reads will return that value or a more recent one.

Characteristics:

- Prevents "time travel" where newer data is followed by older data
- Weaker than session consistency
- Usually implemented with client-side tracking

Examples:

- Client-side timestamping in various NoSQL systems
- Redis with client-side version tracking

Implementation:

```
// Client-side implementation for monotonic read consistency
class MonotonicClient {
  constructor(dbClient) {
    this.dbClient = dbClient;
    this.lastVersionSeen = {}; // Track last version seen for each key
  }

  async read(key) {
    const result = await this.dbClient.get(key);

    // Check if this result is newer than what we've seen before
    if (
      !this.lastVersionSeen[key] ||
      result.version > this.lastVersionSeen[key]
    ) {
      this.lastVersionSeen[key] = result.version;
      return result.value;
    } else {
      // Result is stale, try reading again
```

```
        // In production, this might need backoff or retry limits
        return this.read(key);
      }
    }
  }
```

Impact on Application Design

The choice of consistency model significantly impacts application architecture, error handling, and user experience. Here's how different models affect application design:

1. Error Handling and Retry Logic

Strong Consistency:

- Simpler error handling (fewer inconsistency scenarios)
- Focus on transient failures like timeouts or connection issues
- Explicit transaction boundaries

```
// With strong consistency, retries are primarily for network issues
async function transferFunds(fromAccount, toAccount, amount) {
  const maxRetries = 3;
  let attempt = 0;

  while (attempt < maxRetries) {
    try {
      // Transaction with strong consistency guarantees
      return await performStronglyConsistentTransaction();
    } catch (error) {
      if (isTransientError(error) && attempt < maxRetries - 1) {
        attempt++;
        await exponentialBackoff(attempt);
        continue;
      }
      throw error;
    }
  }
}
```

Eventual Consistency:

- More complex error handling (must account for stale data)
- Need for application-level conflict detection and resolution
- Optimistic concurrency control patterns

```
// With eventual consistency, must handle potential conflicts
async function updateUserProfile(userId, updates) {
  let attempt = 0;
  const maxRetries = 5;
```

```
  while (attempt < maxRetries) {
    try {
      // Read current version
      const currentProfile = await db.get(`user:${userId}`);
      const currentVersion = currentProfile.version || 0;

      // Optimistic concurrency control
      const result = await db.update(
        `user:${userId}`,
        {
          ...updates,
          version: currentVersion + 1,
        },
        {
          conditionExpression: 'version = :currentVersion',
          expressionValues: { ':currentVersion': currentVersion },
        }
      );

      return result;
    } catch (error) {
      if (isConflictError(error) && attempt < maxRetries - 1) {
        attempt++;
        await exponentialBackoff(attempt);
        continue;
      }
      throw error;
    }
  }
}
```

2. Data Modeling

Strong Consistency:

- Simpler data models (can rely on consistency guarantees)
- Focus on normalization and references
- Transactions for maintaining invariants

```
// With strong consistency, can use normalized data models
const orderSchema = {
  order_id: String,
  customer_id: String, // Reference to customer
  items: [
    {
      product_id: String, // Reference to product
      quantity: Number,
      price: Number,
    },
  ],
  total: Number,
  status: String,
};
```

Eventual Consistency:

- More denormalization and duplication
- Aggregate-oriented design (minimize cross-entity consistency needs)
- Design for idempotent operations

```
// With eventual consistency, more denormalization
const orderSchema = {
  order_id: String,
  customer: {
    // Denormalized customer data
    customer_id: String,
    name: String,
    email: String,
  },
  items: [
    {
      product_id: String,
      product_name: String, // Denormalized product data
      product_category: String, // Denormalized product data
      quantity: Number,
      price: Number,
    },
  ],
  total: Number,
  status: String,
};
```

3. User Experience Design

Strong Consistency:

- Direct feedback (changes are immediately visible)
- Simpler UI flows (fewer edge cases to handle)
- More blocking operations (may impact perceived performance)

```
// With strong consistency, UI flow is straightforward
async function handleSubmitOrder() {
  setLoading(true);
  try {
    // Wait for order to be fully processed
    const result = await api.createOrder(orderData);
    // Safe to navigate away, order is confirmed
    router.push(`/orders/${result.orderId}`);
  } catch (error) {
    setError(error.message);
  } finally {
    setLoading(false);
```

```
  }
}
```

Eventual Consistency:

- Optimistic updates (show changes before they're confirmed)
- Progressive disclosure of consistency state
- Non-blocking operations with background reconciliation

```
// With eventual consistency, use optimistic updates
async function handleSubmitOrder() {
  // Generate client-side ID
  const tempOrderId = generateUUID();

  // Optimistically update UI
  dispatch({
    type: 'ADD_ORDER',
    payload: { id: tempOrderId, status: 'pending', ...orderData },
  });

  // Navigate immediately for better perceived performance
  router.push(`/orders/${tempOrderId}`);

  try {
    // Process in background
    const result = await api.createOrder(orderData);

    // Update with real order ID and status
    dispatch({
      type: 'UPDATE_ORDER',
      payload: {
        tempId: tempOrderId,
        realId: result.orderId,
        status: 'confirmed',
      },
    });
  } catch (error) {
    // Handle failure but maintain good UX
    dispatch({
      type: 'UPDATE_ORDER',
      payload: { id: tempOrderId, status: 'failed', error: error.message },
    });

    // Show recovery options
    showRecoveryNotification(error, orderData);
  }
}
```

Performance Implications

Different consistency models have significant performance implications:

1. Latency

Strong Consistency:

- Higher latency (must coordinate across nodes)
- Synchronous operations block until consensus
- Performance degrades with network issues

```
Strong Consistency Path:
Client → Primary Node → Wait for replication to quorum of nodes → Acknowledge
```

Eventual Consistency:

- Lower latency (can respond immediately)
- Asynchronous replication happens in background
- Less affected by network partitions

```
Eventual Consistency Path:
Client → Any Available Node → Acknowledge → Asynchronous replication
```

Benchmark Comparison (Hypothetical):

Consistency Model	Average Write Latency	Average Read Latency	Throughput (ops/sec)
Strong	120ms	80ms	5,000
Causal	80ms	60ms	8,000
Session	50ms	40ms	12,000
Eventual	15ms	10ms	20,000

2. Throughput

Strong Consistency:

- Lower throughput (coordination overhead)
- More resource-intensive operations
- Performance bottlenecked by coordination

Eventual Consistency:

- Higher throughput (minimal coordination)
- More efficient resource utilization
- Can scale almost linearly with node count

3. Availability

Strong Consistency:

- Lower availability during network partitions
- May become unavailable to preserve consistency
- Must maintain quorum to function

Eventual Consistency:

- Higher availability during network partitions
- Continues operation even with node failures
- Can serve stale data rather than no data

When to Choose Weaker Consistency Models

Now that we understand the trade-offs, here are specific scenarios where choosing a weaker consistency model makes sense:

1. High Traffic Consumer Applications

Scenario: Social media platforms, content delivery, news feeds

Why Eventual Consistency Works:

- User experience prioritizes responsiveness over immediate consistency
- Content changes are rarely critical (seeing an old post is acceptable)
- High read-to-write ratio benefits from read optimization
- Scale requirements demand performance at massive scale

Example Implementation:

```
// Social media feed with eventual consistency
async function getUserFeed(userId) {
  // Fast, eventually consistent read for main feed
  const feedItems = await db.query({
    TableName: 'UserFeeds',
    KeyConditionExpression: 'userId = :uid',
    ExpressionAttributeValues: { ':uid': userId },
    ConsistentRead: false, // Eventual consistency for better performance
  });

  // For the user's own posts, use stronger consistency
  const userPosts = await db.query({
    TableName: 'UserPosts',
    KeyConditionExpression: 'userId = :uid',
    ExpressionAttributeValues: { ':uid': userId },
    ConsistentRead: true, // Strong consistency for user's own content
  });

  // Merge and return
  return mergeFeedItems(feedItems, userPosts);
}
```

2. Monitoring and Analytics Systems

Scenario: Dashboards, metrics collection, log aggregation

Why Eventual Consistency Works:

- Real-time exactness is less critical than trends and patterns
- Data is often statistical in nature (slight variations acceptable)
- Extremely high write volume requires performance optimization
- Historical data doesn't change frequently

Example Implementation:

```
// Cassandra metrics collection with eventual consistency
public void recordMetric(String metricName, double value, long timestamp) {
  // Use low consistency for high-throughput writes
  PreparedStatement stmt = session.prepare(
    "INSERT INTO metrics (metric_name, timestamp, value) VALUES (?, ?, ?)"
  );

  session.execute(
    stmt.bind(metricName, timestamp, value)
      .setConsistencyLevel(ConsistencyLevel.ONE)  // Eventual consistency for speed
  );
}

public MetricAggregate getMetricAggregates(String metricName, long startTime, long
endTime) {
  // Use higher consistency for aggregate views
  PreparedStatement stmt = session.prepare(
    "SELECT avg(value), min(value), max(value) FROM metrics " +
    "WHERE metric_name = ? AND timestamp >= ? AND timestamp <= ?"
  );

  return session.execute(
    stmt.bind(metricName, startTime, endTime)
      .setConsistencyLevel(ConsistencyLevel.QUORUM)  // Stronger consistency for
aggregates
  );
}
```

3. Content Caching and Delivery

Scenario: Content distribution networks, media serving, static assets

Why Eventual Consistency Works:

- Content changes infrequently relative to read volume
- Slight staleness (minutes or hours) has minimal impact
- Global distribution requires eventual propagation anyway
- Performance and cost optimizations are critical at scale

Example Implementation:

```
// CDN-like content delivery with TTL-based freshness
class ContentDeliveryService {
  constructor(database, cache) {
    this.database = database;
    this.cache = cache;
  }

  async getContent(contentId) {
    // Try cache first (eventual consistency through TTL)
    const cachedContent = await this.cache.get(contentId);
    if (cachedContent) {
      return cachedContent;
    }

    // Cache miss, get from database (can be eventually consistent)
    const content = await this.database.getContent(contentId);

    // Update cache with TTL
    const ttl = this.getTtlForContent(content.type);
    await this.cache.set(contentId, content, ttl);

    return content;
  }

  // Explicit refresh for important content updates
  async updateContent(contentId, newContent) {
    // Update database
    await this.database.updateContent(contentId, newContent);

    // Explicitly invalidate cache to accelerate consistency
    await this.cache.invalidate(contentId);

    // For critical content, optionally pre-warm the cache
    if (newContent.priority === 'high') {
      await this.cache.set(contentId, newContent);
    }
  }
}
```

4. IoT and Sensor Networks

Scenario: Device telemetry, sensor readings, status updates

Why Eventual Consistency Works:

- Data volume is extremely high (millions of devices sending data)
- Individual readings are rarely critical (patterns matter more)
- Network connectivity may be intermittent
- Local processing may continue during connectivity issues

Example Implementation:

```
// IoT data collection with Cassandra
public class SensorDataService {
  private CassandraClient cassandra;
  private LocalBuffer buffer;

  public void recordReading(String deviceId, Reading reading) {
    try {
      // Try to send immediately with eventual consistency
      cassandra.execute(
        "INSERT INTO sensor_readings (device_id, timestamp, temperature, humidity) "
+
        "VALUES (?, ?, ?, ?)",
        ConsistencyLevel.ONE,  // Fast writes with eventual consistency
        deviceId, reading.timestamp, reading.temperature, reading.humidity
      );
    } catch (ConnectionException e) {
      // Buffer locally if connectivity issues
      buffer.store(deviceId, reading);
      scheduleReconnectionAttempt();
    }
  }

  // Critical readings use higher consistency
  public void recordCriticalAlert(String deviceId, Alert alert) {
    cassandra.execute(
      "INSERT INTO device_alerts (device_id, alert_id, timestamp, type, reading) " +
      "VALUES (?, ?, ?, ?, ?)",
      ConsistencyLevel.QUORUM,  // Stronger consistency for critical alerts
      deviceId, alert.id, alert.timestamp, alert.type, alert.reading
    );
  }
}
```

5. Shopping Cart and Checkout Flow

Scenario: E-commerce applications, particularly pre-checkout activities

Why Eventual Consistency Works:

- Shopping cart updates are frequent and high-volume
- Cart state is temporary and mutable by nature
- Final checkout can use stronger consistency when needed
- User experience benefits from responsiveness

Example Implementation:

```
// Hybrid consistency for e-commerce
class ShoppingService {
  // Cart operations use eventual consistency
  async updateCart(userId, item, quantity) {
    return await dynamoDb.update({
      TableName: 'ShoppingCarts',
```

```
      Key: { userId: userId },
      UpdateExpression: 'SET items.#itemId = :itemDetails',
      ExpressionAttributeNames: { '#itemId': item.id },
      ExpressionAttributeValues: {
        ':itemDetails': {
          id: item.id,
          name: item.name,
          price: item.price,
          quantity: quantity,
        },
      },
      // No ConsistentRead parameter (defaults to eventual consistency)
    });
  }

  // Checkout uses strong consistency
  async processCheckout(userId, paymentDetails) {
    // Begin transaction with strong consistency
    const checkoutTransaction = await this.beginTransaction();

    try {
      // Get cart with strong consistency
      const cart = await dynamoDb.get({
        TableName: 'ShoppingCarts',
        Key: { userId: userId },
        ConsistentRead: true, // Strong consistency for checkout
      });

      // Verify inventory with strong consistency
      await this.verifyInventory(cart.items, checkoutTransaction);

      // Process payment
      const paymentResult = await this.processPayment(
        paymentDetails,
        cart.total
      );

      // Update inventory and create order
      await this.createOrder(
        userId,
        cart,
        paymentResult.transactionId,
        checkoutTransaction
      );

      // Clear cart
      await this.clearCart(userId, checkoutTransaction);

      // Commit transaction
      await checkoutTransaction.commit();

      return { success: true, orderId: orderId };
    } catch (error) {
      await checkoutTransaction.rollback();
      throw error;
```

```
    }
  }
}
```

Real-World Industry Examples

Here are some notable examples of how major technology companies leverage consistency models:

Amazon DynamoDB

Amazon defaults to eventual consistency for reads but offers strong consistency as an option. This design allows applications to choose the appropriate model for each operation:

```
// Product catalog - eventual consistency for browsing
const productListing = await dynamoDb.query({
  TableName: 'Products',
  IndexName: 'CategoryIndex',
  KeyConditionExpression: 'category = :cat',
  ExpressionAttributeValues: { ':cat': 'Electronics' },
  // Default: eventually consistent
});

// Order processing - strong consistency for checkout
const orderDetails = await dynamoDb.get({
  TableName: 'Orders',
  Key: { orderId: orderId },
  ConsistentRead: true, // Strong consistency
});
```

Netflix

Netflix extensively uses eventual consistency in their microservices architecture, implementing the "AP" side of the CAP theorem:

```
// Netflix-style eventually consistent service
@Service
public class ContentCatalogService {
  @HystrixCommand(fallbackMethod = "getContentFromCache")
  public Content getContent(String contentId) {
    // Eventually consistent read from distributed database
    return contentRepository.findById(contentId);
  }

  // Fallback to cache during outages - sacrificing consistency for availability
  public Content getContentFromCache(String contentId) {
    return cacheService.getContent(contentId);
  }
}
```

Cassandra at Apple

Apple uses Cassandra for various services, choosing different consistency levels based on the criticality of operations:

```
// Hypothetical example inspired by Apple's Cassandra usage
public class UserPreferencesService {
  // User preferences - eventual consistency is acceptable
  public void updatePreference(String userId, String key, String value) {
    session.execute(
      "UPDATE user_preferences SET value = ? WHERE user_id = ? AND key = ?",
      value, userId, key
    ).setConsistencyLevel(ConsistencyLevel.ONE);
  }

  // Account security settings - stronger consistency required
  public void updateSecuritySettings(String userId, SecuritySettings settings) {
    session.execute(
      "UPDATE security_settings SET settings = ? WHERE user_id = ?",
      settings.toJson(), userId
    ).setConsistencyLevel(ConsistencyLevel.QUORUM);
  }
}
```

Conclusion and Best Practices

When choosing consistency models, follow these guidelines:

1. **Identify Critical vs. Non-Critical Operations**:
   - Use stronger consistency for financial transactions, permissions changes, and unique constraints
   - Use weaker consistency for analytics, content presentation, and high-volume telemetry
2. **Consider Visibility Requirements**:
   - If immediate visibility of changes is required, use stronger consistency
   - If some delay is acceptable, eventual consistency may offer better performance
3. **Analyze Read vs. Write Patterns**:
   - Read-heavy workloads often benefit more from eventual consistency
   - Write-heavy workloads with complex integrity constraints may need stronger models
4. **Plan for Failure Scenarios**:
   - Determine acceptable behavior during network partitions
   - Decide whether availability or consistency is more important for each operation
5. **Layer Consistency Strategies**:
   - Use different consistency levels for different operations
   - Implement application-level validation for critical invariants
   - Consider versioning and conflict resolution for concurrent updates

By thoughtfully selecting the appropriate consistency model for each part of your application, you can balance performance, availability, and correctness to build systems that scale effectively while meeting business requirements.

4.6 Query Patterns and Optimization

Effective query patterns and optimization strategies are essential for getting the most performance out of NoSQL databases.

Common NoSQL Query Patterns

1. **Key-Based Access**:
   - Direct lookup by primary key
   - Most efficient query pattern in NoSQL databases

```
// MongoDB key-based lookup
db.users.findOne({ _id: 'user123' });

// DynamoDB key-based lookup
const params = {
  TableName: 'Users',
  Key: {
    UserId: 'user123',
  },
};
```

2. **Range Queries**:
   - Query data within a specified range
   - Commonly used with time series data

```
// MongoDB range query
db.events
  .find({
    userId: 'user123',
    timestamp: { $gte: ISODate('2023-06-01'), $lt: ISODate('2023-07-01') },
  })
  .sort({ timestamp: -1 });

// DynamoDB range query
const params = {
  TableName: 'Events',
  KeyConditionExpression:
    'UserId = :uid AND EventTime BETWEEN :start AND :end',
  ExpressionAttributeValues: {
    ':uid': 'user123',
    ':start': '2023-06-01T00:00:00Z',
    ':end': '2023-06-30T23:59:59Z',
  },
};
```

3. **Filtering**:
   - Apply conditions to restrict results
   - Efficiency depends on indexing and database capabilities

```
// MongoDB filtering
db.products.find({
  category: 'Electronics',
  price: { $lt: 500 },
  inStock: true,
});

// DynamoDB filtering (less efficient than key conditions)
const params = {
  TableName: 'Products',
  KeyConditionExpression: 'Category = :cat',
  FilterExpression: 'Price < :price AND InStock = :inStock',
  ExpressionAttributeValues: {
    ':cat': 'Electronics',
    ':price': 500,
    ':inStock': true,
  },
};
```

4. **Aggregation**:
   - Compute summary statistics
   - Varies widely in efficiency across NoSQL databases

```
// MongoDB aggregation
db.orders.aggregate([
  { $match: { orderDate: { $gte: ISODate('2023-01-01') } } },
  {
    $group: {
      _id: '$productId',
      totalSales: { $sum: '$amount' },
      orderCount: { $sum: 1 },
    },
  },
  { $sort: { totalSales: -1 } },
]);

// Cassandra aggregation
const query = `
  SELECT product_id, SUM(amount) AS total_sales, COUNT(*) AS order_count
  FROM orders
  WHERE order_date >= '2023-01-01'
  GROUP BY product_id
`;
```

5. **Pagination**:
    - Retrieve results in manageable chunks
    - Important for large result sets

```
// MongoDB pagination
db.products
  .find({ category: 'Electronics' })
  .sort({ price: 1 })
  .skip(20)
  .limit(10);

// DynamoDB pagination
// Initial query
const params = {
  TableName: 'Products',
  KeyConditionExpression: 'Category = :cat',
  ExpressionAttributeValues: {
    ':cat': 'Electronics',
  },
  Limit: 10,
};
const result = await dynamoDb.query(params).promise();

// Subsequent query with pagination token
if (result.LastEvaluatedKey) {
  const nextPageParams = {
    ...params,
    ExclusiveStartKey: result.LastEvaluatedKey,
  };
  const nextPage = await dynamoDb.query(nextPageParams).promise();
}
```

6. **Full-Text Search**:
    - Search for text across multiple fields
    - Often requires specialized indexes or external services

```
// MongoDB text search
db.products.createIndex({ name: 'text', description: 'text' });
db.products.find({ $text: { $search: 'wireless headphones' } });

// Elasticsearch integration example
const searchResults = await elastic.search({
  index: 'products',
  body: {
    query: {
      multi_match: {
        query: 'wireless headphones',
        fields: ['name^2', 'description'],
      },
    },
```

```
  },
});
```

Query Optimization Techniques

1. **Indexing Strategies**:
   - Create indexes based on query patterns
   - Consider composite indexes for complex queries
   - Be mindful of index overhead for writes

```
// MongoDB index creation
db.products.createIndex({ category: 1, price: -1 });

// Cassandra index creation
CREATE INDEX ON products(category);

// DynamoDB secondary index
const params = {
  AttributeDefinitions: [
    { AttributeName: "Category", AttributeType: "S" },
    { AttributeName: "Price", AttributeType: "N" }
  ],
  GlobalSecondaryIndexUpdates: [
    {
      Create: {
        IndexName: "CategoryPriceIndex",
        KeySchema: [
          { AttributeName: "Category", KeyType: "HASH" },
          { AttributeName: "Price", KeyType: "RANGE" }
        ],
        Projection: {
          ProjectionType: "ALL"
        }
      }
    }
  ]
};
```

2. **Query Analysis**:
   - Use query profiling tools to identify bottlenecks
   - Analyze execution plans for complex queries

```
// MongoDB query explanation
db.products
  .find({ category: 'Electronics', price: { $lt: 500 } })
  .explain('executionStats');

// Cassandra query tracing
```

```
const query = 'SELECT * FROM products WHERE category = ? LIMIT 10';
const result = await session.execute(query, ['Electronics'], {
  tracing: true,
});
console.log(result.info.traceId);
```

3. **Data Modeling for Query Efficiency**:
   - Design data models around access patterns
   - Denormalize strategically for query performance

```
// Original separate collections
// Users collection
{ "_id": "user123", "name": "John Smith", "email": "john@example.com" }

// Orders collection
{ "_id": "order456", "userId": "user123", "product": "Headphones", "amount":
99.99 }

// Denormalized for query efficiency
// Orders with user details embedded
{
  "_id": "order456",
  "userId": "user123",
  "userName": "John Smith",
  "userEmail": "john@example.com",
  "product": "Headphones",
  "amount": 99.99
}
```

4. **Caching**:
   - Implement application-level caching for frequent queries
   - Consider time-to-live (TTL) for cached data

```
// Redis caching example
async function getProduct(productId) {
  // Try to get from cache
  const cachedProduct = await redisClient.get(`product:${productId}`);
  if (cachedProduct) {
    return JSON.parse(cachedProduct);
  }

  // Not in cache, get from database
  const product = await db.products.findOne({ _id: productId });

  // Save to cache with TTL
  await redisClient.setex(
    `product:${productId}`,
    3600,
    JSON.stringify(product)
```

```
  );

  return product;
}
```

5. **Query Batching**:
   - Group multiple related queries together
   - Reduces network round trips

```
// MongoDB batched queries
db.products.bulkWrite([
  {
    updateOne: {
      filter: { _id: 'prod1' },
      update: { $set: { inStock: false } },
    },
  },
  {
    updateOne: {
      filter: { _id: 'prod2' },
      update: { $set: { inStock: true } },
    },
  },
  {
    updateOne: {
      filter: { _id: 'prod3' },
      update: { $set: { price: 149.99 } },
    },
  },
]);

// DynamoDB batch operations
const params = {
  RequestItems: {
    Products: [
      {
        PutRequest: {
          Item: {
            /* item 1 details */
          },
        },
      },
      {
        PutRequest: {
          Item: {
            /* item 2 details */
          },
        },
      },
    ],
    Orders: [
      {
```

```
        PutRequest: {
          Item: {
            /* order 1 details */
          },
        },
      },
    ],
  },
};
await dynamoDb.batchWrite(params).promise();
```

6. **Data Distribution Strategies**:
   - Understand how data is distributed across nodes
   - Avoid hot spots with proper key design

```
// DynamoDB - Avoiding hot partition with composite keys

// Poor key design (might create hot partition)
"PK": "ORDER#" + getCurrentDate()  // Many orders on same day

// Better key design (distributes load)
"PK": "ORDER#" + getCurrentDate() + "#" + random(1, 10)

// Cassandra - Ensuring even data distribution
CREATE TABLE events (
  bucket INT,  // Add a bucket number for distribution
  event_id UUID,
  timestamp TIMESTAMP,
  data TEXT,
  PRIMARY KEY ((bucket, event_id))
);

// Insert with randomized bucket for distribution
INSERT INTO events (bucket, event_id, timestamp, data)
VALUES (RAND() % 10, uuid(), toTimestamp(now()), 'Event data');
```

7. **Query Result Limiting**:
   - Always limit result sets for better performance
   - Use pagination for large result sets

```
// MongoDB result limiting
db.logs.find({ level: 'ERROR' }).sort({ timestamp: -1 }).limit(100);

// Cassandra result limiting
const query =
  "SELECT * FROM logs WHERE level = 'ERROR' ORDER BY timestamp DESC LIMIT 100";
```

Database-Specific Optimization Techniques

MongoDB:

1. **Projection** to limit returned fields:

```
db.users.find({ age: { $gt: 25 } }, { name: 1, email: 1, _id: 0 });
```

2. **Covered Queries** that can be satisfied entirely by an index:

```
// Create index on both query fields and projected fields
db.users.createIndex({ age: 1, name: 1, email: 1 });

// Query can be satisfied entirely by the index
db.users.find({ age: { $gt: 25 } }, { name: 1, email: 1, _id: 0 });
```

3. **Compound Indexes** for multi-field queries:

```
// Create compound index
db.products.createIndex({ category: 1, price: -1, name: 1 });

// Queries that can utilize this index
db.products.find({ category: 'Electronics' }).sort({ price: -1 });
db.products.find({ category: 'Electronics', price: { $lt: 500 } });
```

4. **Aggregation Pipeline Optimization**:

```
// Original pipeline
db.orders.aggregate([
  { $match: { status: 'completed' } },
  { $unwind: '$items' },
  {
    $group: {
      _id: '$items.product_id',
      totalSales: { $sum: '$items.quantity' },
    },
  },
  { $sort: { totalSales: -1 } },
  { $limit: 10 },
]);

// Optimized pipeline - $match early, $limit before $sort
db.orders.aggregate([
  { $match: { status: 'completed' } },
  { $unwind: '$items' },
  {
    $group: {
      _id: '$items.product_id',
```

```
      totalSales: { $sum: '$items.quantity' },
    },
  },
  { $sort: { totalSales: -1 } },
  { $limit: 10 },
]);
```

DynamoDB:

1. **Sparse Indexes** for selective querying:

```
// Only index items with 'premium' attribute
const params = {
  AttributeDefinitions: [
    { AttributeName: 'UserId', AttributeType: 'S' },
    { AttributeName: 'PremiumStatus', AttributeType: 'S' },
  ],
  GlobalSecondaryIndexUpdates: [
    {
      Create: {
        IndexName: 'PremiumUsersIndex',
        KeySchema: [
          { AttributeName: 'PremiumStatus', KeyType: 'HASH' },
          { AttributeName: 'UserId', KeyType: 'RANGE' },
        ],
        Projection: {
          ProjectionType: 'INCLUDE',
          NonKeyAttributes: ['SubscriptionType', 'ExpiryDate'],
        },
      },
    },
  ],
};
```

2. **Careful Projection** to minimize data transfer:

```
// Only return specific attributes
const params = {
  TableName: 'Users',
  KeyConditionExpression: 'UserId = :uid',
  ExpressionAttributeValues: {
    ':uid': 'user123',
  },
  ProjectionExpression: 'FirstName, LastName, Email',
};
```

3. **Write Sharding** to distribute load:

```
// Distribute writes across partitions
function getShardedKey(baseId) {
  const shardId = Math.floor(Math.random() * 10); // 10 logical shards
  return `${shardId}#${baseId}`;
}

// Insert item with sharded key
const params = {
  TableName: 'HighVolumeData',
  Item: {
    ShardedId: getShardedKey('order123'),
    OriginalId: 'order123',
    // other attributes
  },
};
```

Cassandra:

1. **Materialized Views** for alternative access patterns:

```
-- Base table
CREATE TABLE users_by_id (
  user_id UUID PRIMARY KEY,
  email TEXT,
  name TEXT
);

-- Materialized view for lookup by email
CREATE MATERIALIZED VIEW users_by_email AS
  SELECT user_id, email, name FROM users_by_id
  WHERE email IS NOT NULL
  PRIMARY KEY (email, user_id);
```

2. **Denormalized Tables** for specific query patterns:

```
-- User posts with nested user data
CREATE TABLE user_posts (
  user_id UUID,
  post_id TIMEUUID,
  post_content TEXT,
  user_name TEXT,  -- Denormalized user data
  user_email TEXT,  -- Denormalized user data
  PRIMARY KEY (user_id, post_id)
) WITH CLUSTERING ORDER BY (post_id DESC);
```

3. **ALLOW FILTERING** for Occasional Queries (use sparingly):

```
-- Non-optimal query with filtering
SELECT * FROM users
WHERE age > 30 AND country = 'USA'
ALLOW FILTERING;

-- Better approach: Create table for this access pattern
CREATE TABLE users_by_country_age (
  country TEXT,
  age INT,
  user_id UUID,
  name TEXT,
  email TEXT,
  PRIMARY KEY ((country), age, user_id)
) WITH CLUSTERING ORDER BY (age ASC, user_id ASC);

-- Efficient query
SELECT * FROM users_by_country_age
WHERE country = 'USA' AND age > 30;
```

Performance Testing and Monitoring

Systematic performance testing and monitoring are essential for optimizing NoSQL database performance.

1. **Establish Performance Baselines**:
    - Measure current performance metrics
    - Document expected performance levels
    - Create realistic test data
2. **Load Testing**:
    - Simulate realistic workloads
    - Test with different data volumes
    - Identify bottlenecks under load
3. **Monitoring Key Metrics**:
    - Query latency and throughput
    - Read vs. write performance
    - Cache hit rates
    - Storage utilization
4. **Tools and Approaches**:
    - Database-specific monitoring tools
    - APM (Application Performance Monitoring) solutions
    - Custom instrumentation for critical paths

```
// MongoDB monitoring example with node.js
const { MongoClient } = require('mongodb');
const client = new MongoClient(uri);
```

```
// Create a custom logger for monitoring
function monitoredQuery(collection, query, options = {}) {
  const startTime = Date.now();
  const result = collection.find(query, options).toArray();
  const duration = Date.now() - startTime;

  // Log performance metrics
  logger.info({
    operation: 'query',
    collection: collection.collectionName,
    query: JSON.stringify(query),
    duration,
    resultCount: result.length,
  });

  return result;
}

// Monitor connection pool
setInterval(() => {
  const stats = client.db().admin().serverStatus();
  logger.info({
    connections: stats.connections,
    activeClients: stats.globalLock.activeClients,
    currentQueue: stats.globalLock.currentQueue,
  });
}, 60000);
```

Interview Questions

Q: How would you design a NoSQL database schema and optimize queries for a high-volume e-commerce application? What specific approaches would you use for product catalogs, shopping carts, and order processing?

A: Designing a NoSQL database schema for a high-volume e-commerce application requires carefully analyzing access patterns, understanding scaling requirements, and making strategic decisions about data modeling and query optimization. Let me outline a comprehensive approach that addresses the specific needs of product catalogs, shopping carts, and order processing.

Core Design Principles for E-commerce NoSQL Schema

Before diving into specific components, I'll outline key principles that will guide our design:

1. **Access Pattern-First Design**: Model data based on how it will be accessed, not just how it logically relates
2. **Denormalization for Performance**: Strategically duplicate data to optimize read performance
3. **Partition Key Design for Scale**: Ensure even distribution of data and workload
4. **Hybrid Consistency Model**: Use different consistency levels based on operation criticality
5. **Query Efficiency Optimization**: Minimize the need for table scans and complex joins

For this e-commerce application, I'll use a combination of document databases (like MongoDB) and key-value stores (like Redis or DynamoDB) to handle different aspects of the application.

Product Catalog Design

The product catalog needs to support:

- Product browsing by category, brand, attributes
- Searching with filters and sorting
- Product detail views
- Inventory status checks

Document Model (MongoDB):

```
// Products collection
{
  "_id": "prod12345",
  "sku": "SM-BK-L",
  "name": "Premium Ergonomic Office Chair",
  "slug": "premium-ergonomic-office-chair",
  "description": "Ergonomic office chair with lumbar support...",
  "price": {
    "current": 299.99,
    "original": 349.99,
    "currency": "USD"
  },
  "category": {
    "id": "cat789",
    "name": "Office Furniture",
    "path": ["Home & Garden", "Office", "Office Furniture"]
  },
  "brand": {
    "id": "brand456",
    "name": "ErgoComfort"
  },
  "attributes": {
    "color": "Black",
    "size": "Large",
    "material": "Mesh",
    "weight": "35 lbs",
    "dimensions": "28 x 25 x 45 inches"
  },
  "images": [
    {
      "url": "https://example.com/images/prod12345-1.jpg",
      "alt": "Front view",
      "is_primary": true
    },
    {
      "url": "https://example.com/images/prod12345-2.jpg",
      "alt": "Side view",
      "is_primary": false
    }
  ],
  "inventory": {
    "status": "in_stock",  // in_stock, low_stock, out_of_stock, backorder
```

```
    "quantity": 42,
    "warehouse_location": "WH-005-B12"
  },
  "metadata": {
    "created_at": ISODate("2023-01-15T08:30:00Z"),
    "updated_at": ISODate("2023-06-10T14:20:00Z"),
    "search_keywords": ["office chair", "ergonomic", "lumbar support", "mesh chair"]
  },
  "ratings": {
    "average": 4.7,
    "count": 128
  }
}

// Categories collection
{
  "_id": "cat789",
  "name": "Office Furniture",
  "slug": "office-furniture",
  "description": "Furniture for your office and workspace",
  "parent_id": "cat456",  // Parent category ID
  "path": ["cat123", "cat456", "cat789"],  // Category hierarchy path
  "path_names": ["Home & Garden", "Office", "Office Furniture"],
  "is_active": true,
  "attributes": ["color", "material", "size"],  // Filterable attributes
  "image_url": "https://example.com/images/categories/office-furniture.jpg"
}

// Brands collection
{
  "_id": "brand456",
  "name": "ErgoComfort",
  "slug": "ergocomfort",
  "description": "Ergonomic furniture and accessories",
  "logo_url": "https://example.com/images/brands/ergocomfort.jpg",
  "is_featured": true
}
```

Key-Value Model for Inventory (Redis):

```
# Real-time inventory tracking
HSET product:prod12345:inventory quantity 42 reserved 5 available 37

# Inventory change rate tracking (for predictive analysis)
ZADD product:prod12345:inventory:history 1623456789 42
ZADD product:prod12345:inventory:history 1623457000 40

# Low stock alert threshold
SET product:prod12345:low_stock_threshold 10
```

Secondary Indexes (MongoDB):

```
// Create indexes for common query patterns
db.products.createIndex({ 'category.id': 1, 'price.current': 1 });
db.products.createIndex({ 'brand.id': 1, 'price.current': 1 });
db.products.createIndex({ 'inventory.status': 1 });
db.products.createIndex({ 'attributes.color': 1, 'attributes.size': 1 });

// Text search index
db.products.createIndex(
  {
    name: 'text',
    description: 'text',
    'metadata.search_keywords': 'text',
  },
  {
    weights: {
      name: 10,
      description: 5,
      'metadata.search_keywords': 3,
    },
  }
);
```

Query Optimization Strategies:

1. **Product Browse Optimization**:

```
// Efficient category browsing
db.products
  .find({
    'category.id': 'cat789',
    'inventory.status': { $in: ['in_stock', 'low_stock'] },
  })
  .sort({ 'price.current': 1 })
  .limit(20)
  .project({
    _id: 1,
    name: 1,
    'price.current': 1,
    images: { $slice: [1] }, // Only fetch primary image
    ratings: 1,
  });
```

2. **Product Search with Facets**:

```
// Aggregation pipeline for search with facets
db.products.aggregate([
  // Match based on search text
  {
    $match: {
```

```
        $text: { $search: 'ergonomic chair' },
        'category.path': 'Office Furniture',
        'inventory.status': { $ne: 'out_of_stock' },
      },
    },
    // Facet aggregation in one operation
    {
      $facet: {
        // Return paginated results
        results: [
          { $sort: { score: { $meta: 'textScore' }, 'ratings.average': -1 } },
          { $skip: 0 },
          { $limit: 20 },
          {
            $project: {
              _id: 1,
              name: 1,
              'price.current': 1,
              images: { $slice: ['$images', 1] },
              ratings: 1,
              'inventory.status': 1,
            },
          },
        ],
        // Calculate facets for filtering UI
        colorFacets: [
          { $group: { _id: '$attributes.color', count: { $sum: 1 } } },
          { $sort: { count: -1 } },
        ],
        brandFacets: [
          { $group: { _id: '$brand.name', count: { $sum: 1 } } },
          { $sort: { count: -1 } },
        ],
        priceFacets: [
          {
            $bucket: {
              groupBy: '$price.current',
              boundaries: [0, 50, 100, 200, 500, 1000],
              default: '1000+',
              output: { count: { $sum: 1 } },
            },
          },
        ],
      },
    },
  ]);
```

3. **Inventory Check Optimization**:

```
// Use Redis for real-time inventory checks
async function checkProductAvailability(productId, quantity) {
  const available = await redis.hget(
    `product:${productId}:inventory`,
```

```
    'available'
  );
  return parseInt(available, 10) >= quantity;
}

// Batch inventory check for cart validation
async function validateCartInventory(cartItems) {
  const pipeline = redis.pipeline();

  // Queue up all inventory checks
  cartItems.forEach((item) => {
    pipeline.hget(`product:${item.productId}:inventory`, 'available');
  });

  // Execute all checks in a single round-trip
  const results = await pipeline.exec();

  // Validate each item
  return cartItems.map((item, index) => {
    const available = parseInt(results[index][1], 10);
    return {
      productId: item.productId,
      quantity: item.quantity,
      isAvailable: available >= item.quantity,
      availableQuantity: available,
    };
  });
}
```

Shopping Cart Design

Shopping carts need to support:

- High-frequency updates
- Session persistence
- Merging anonymous and user carts
- Price and availability validation

Key-Value Model (Redis for Active Carts):

```
# Cart structure using Redis hashes
HSET cart:user123 last_updated 1686839400
HMSET cart:user123:items product:prod12345 '{"id":"prod12345","name":"Premium
Chair","price":299.99,"quantity":1}'
HMSET cart:user123:items product:prod67890 '{"id":"prod67890","name":"Desk
Lamp","price":49.99,"quantity":2}'

# Set TTL for anonymous carts
EXPIRE cart:session789xyz 86400  # 24 hours
```

Document Model (MongoDB for Persistent Carts):

```
// Saved carts collection
{
  "_id": "cart123",
  "user_id": "user123",
  "status": "active",  // active, saved_for_later, abandoned, converted
  "items": [
    {
      "product_id": "prod12345",
      "quantity": 1,
      "added_at": ISODate("2023-06-15T14:30:00Z"),
      "price_snapshot": 299.99,
      "product_snapshot": {
        "name": "Premium Ergonomic Office Chair",
        "image_url": "https://example.com/images/prod12345-1.jpg",
        "attributes": {
          "color": "Black",
          "size": "Large"
        }
      }
    },
    {
      "product_id": "prod67890",
      "quantity": 2,
      "added_at": ISODate("2023-06-15T14:35:00Z"),
      "price_snapshot": 49.99,
      "product_snapshot": {
        "name": "Desk Lamp",
        "image_url": "https://example.com/images/prod67890-1.jpg",
        "attributes": {
          "color": "Silver"
        }
      }
    }
  ],
  "created_at": ISODate("2023-06-15T14:30:00Z"),
  "updated_at": ISODate("2023-06-15T14:35:00Z"),
  "metadata": {
    "source": "web",
    "user_agent": "Mozilla/5.0...",
    "ip_address": "192.168.1.1"
  },
  "summary": {
    "item_count": 3,
    "subtotal": 399.97,
    "currency": "USD"
  }
}
```

Query Optimization Strategies:

1. **Cart Updates with Atomic Operations**:

```
// Redis atomic cart updates
async function updateCartItem(userId, productId, quantity) {
  const product = await getProductDetails(productId);

  // Run multiple commands in a pipeline
  const pipeline = redis.pipeline();

  if (quantity > 0) {
    // Update cart item
    pipeline.hset(
      `cart:${userId}:items`,
      `product:${productId}`,
      JSON.stringify({
        id: productId,
        name: product.name,
        price: product.price.current,
        quantity: quantity,
      })
    );
  } else {
    // Remove cart item
    pipeline.hdel(`cart:${userId}:items`, `product:${productId}`);
  }

  // Update last modified timestamp
  pipeline.hset(
    `cart:${userId}`,
    'last_updated',
    Math.floor(Date.now() / 1000)
  );

  // Execute all operations atomically
  return pipeline.exec();
}
```

2. **Efficient Cart Retrieval**:

```
// Redis efficient cart retrieval
async function getCart(userId) {
  // Get cart metadata and items in parallel
  const [metaResult, itemsResult] = await Promise.all([
    redis.hgetall(`cart:${userId}`),
    redis.hgetall(`cart:${userId}:items`),
  ]);

  // Process and return cart data
  const items = Object.entries(itemsResult || {}).map(([key, value]) => {
    return JSON.parse(value);
  });

  // Calculate summary
  const subtotal = items.reduce(
```

```
    (sum, item) => sum + item.price * item.quantity,
    0
  );

  return {
    user_id: userId,
    last_updated: metaResult.last_updated,
    items: items,
    summary: {
      item_count: items.reduce((count, item) => count + item.quantity, 0),
      subtotal: subtotal,
      currency: 'USD',
    },
  };
}
```

3. **Cart Merging Optimization**:

```
// Merge anonymous cart with user cart
async function mergeAnonymousCart(anonymousCartId, userId) {
  // Get both carts
  const [anonymousCart, userCart] = await Promise.all([
    getCart(anonymousCartId),
    getCart(userId),
  ]);

  // Create a map of user cart items for quick lookup
  const userCartMap = {};
  userCart.items.forEach((item) => {
    userCartMap[item.id] = item;
  });

  // Merge items with Redis pipeline
  const pipeline = redis.pipeline();

  // Process anonymous cart items
  for (const item of anonymousCart.items) {
    const productId = item.id;
    const anonymousQuantity = item.quantity;

    if (userCartMap[productId]) {
      // Increment existing product quantity
      const newQuantity = userCartMap[productId].quantity + anonymousQuantity;
      pipeline.hset(
        `cart:${userId}:items`,
        `product:${productId}`,
        JSON.stringify({
          ...userCartMap[productId],
          quantity: newQuantity,
        })
      );
    } else {
      // Add new product to user cart
```

```
        pipeline.hset(
          `cart:${userId}:items`,
          `product:${productId}`,
          JSON.stringify(item)
        );
      }
    }

    // Update last modified timestamp
    pipeline.hset(
      `cart:${userId}`,
      'last_updated',
      Math.floor(Date.now() / 1000)
    );

    // Delete anonymous cart
    pipeline.del(`cart:${anonymousCartId}`);
    pipeline.del(`cart:${anonymousCartId}:items`);

    // Execute all operations
    await pipeline.exec();

    // Return updated cart
    return getCart(userId);
  }
```

Order Processing Design

Order processing requires:

- Transactional integrity
- Historical lookup
- Status tracking
- Integration with inventory and payment systems

Document Model (MongoDB):

```
// Orders collection
{
  "_id": "order789",
  "order_number": "ORD-20230615-789",
  "user_id": "user123",
  "user_info": {
    "name": "John Smith",
    "email": "john@example.com",
    "phone": "+1-555-123-4567"
  },
  "status": "processing",  // pending, processing, shipped, delivered, cancelled
  "payment_status": "paid",  // pending, authorized, paid, refunded, failed
  "shipping_status": "preparing",  // not_started, preparing, shipped, delivered
  "created_at": ISODate("2023-06-15T15:30:00Z"),
  "updated_at": ISODate("2023-06-15T15:35:00Z"),
```

```
    "items": [
      {
        "product_id": "prod12345",
        "sku": "SM-BK-L",
        "name": "Premium Ergonomic Office Chair",
        "quantity": 1,
        "unit_price": 299.99,
        "subtotal": 299.99,
        "attributes": {
          "color": "Black",
          "size": "Large"
        }
      },
      {
        "product_id": "prod67890",
        "sku": "DL-SV",
        "name": "Desk Lamp",
        "quantity": 2,
        "unit_price": 49.99,
        "subtotal": 99.98,
        "attributes": {
          "color": "Silver"
        }
      }
    ],
    "pricing": {
      "subtotal": 399.97,
      "tax": 32.00,
      "shipping": 15.00,
      "discounts": [
        {
          "code": "SUMMER10",
          "type": "percentage",
          "amount": 40.00,
          "description": "10% Summer Sale"
        }
      ],
      "total": 406.97,
      "currency": "USD"
    },
    "shipping_address": {
      "name": "John Smith",
      "street1": "123 Main St",
      "street2": "Apt 4B",
      "city": "Boston",
      "state": "MA",
      "postal_code": "02101",
      "country": "US",
      "phone": "+1-555-123-4567"
    },
    "billing_address": {
      "name": "John Smith",
      "street1": "123 Main St",
      "street2": "Apt 4B",
      "city": "Boston",
```

```
      "state": "MA",
      "postal_code": "02101",
      "country": "US",
      "phone": "+1-555-123-4567"
    },
    "payment": {
      "method": "credit_card",
      "transaction_id": "txn_4567890",
      "amount": 406.97,
      "currency": "USD",
      "card_brand": "Visa",
      "last4": "4242"
    },
    "shipping": {
      "method": "standard",
      "carrier": "UPS",
      "tracking_number": "1Z999AA10123456784",
      "estimated_delivery": ISODate("2023-06-20T00:00:00Z")
    },
    "history": [
      {
        "timestamp": ISODate("2023-06-15T15:30:00Z"),
        "status": "pending",
        "comment": "Order placed"
      },
      {
        "timestamp": ISODate("2023-06-15T15:35:00Z"),
        "status": "processing",
        "comment": "Payment successful"
      }
    ],
    "metadata": {
      "source": "web",
      "ip_address": "192.168.1.1",
      "user_agent": "Mozilla/5.0..."
    }
  }

  // User Orders Index Collection (for faster user order lookup)
  {
    "_id": "user123",
    "orders": [
      {
        "order_id": "order789",
        "order_number": "ORD-20230615-789",
        "date": ISODate("2023-06-15T15:30:00Z"),
        "total": 406.97,
        "status": "processing"
      },
      {
        "order_id": "order456",
        "order_number": "ORD-20230520-456",
        "date": ISODate("2023-05-20T10:15:00Z"),
        "total": 124.95,
        "status": "delivered"
```

```
    }
  ]
}
```

Time-Series Data for Analytics (Separate Collection):

```
// Order stats collection (for analytics)
{
  "_id": ObjectId("..."),
  "date": ISODate("2023-06-15T00:00:00Z"),
  "hour": 15,
  "order_count": 42,
  "total_revenue": 12567.89,
  "average_order_value": 299.23,
  "top_categories": [
    { "id": "cat789", "name": "Office Furniture", "order_count": 15, "revenue":
4500.00 },
    { "id": "cat456", "name": "Office Supplies", "order_count": 27, "revenue":
8067.89 }
  ],
  "top_products": [
    { "id": "prod12345", "name": "Premium Chair", "order_count": 10, "revenue":
2999.90 }
  ]
}
```

Query Optimization Strategies:

1. **Order Creation with Inventory Update**:

```
// MongoDB multi-document transaction for order creation
async function createOrder(userId, cartId, paymentDetails, addresses) {
  const session = client.startSession();

  try {
    session.startTransaction();

    // Get user cart
    const cart = await getCartDetails(cartId);

    // Validate inventory
    const inventoryCheck = await validateInventory(cart.items);
    if (!inventoryCheck.allAvailable) {
      throw new Error('Some items are no longer available');
    }

    // Create order document
    const orderData = {
      order_number: generateOrderNumber(),
      user_id: userId,
```

```
    user_info: await getUserBasicInfo(userId),
    status: 'pending',
    payment_status: 'pending',
    shipping_status: 'not_started',
    created_at: new Date(),
    updated_at: new Date(),
    items: cart.items.map((item) => ({
      product_id: item.id,
      sku: item.sku,
      name: item.name,
      quantity: item.quantity,
      unit_price: item.price,
      subtotal: item.price * item.quantity,
      attributes: item.attributes,
    })),
    pricing: calculateOrderPricing(cart, addresses.shipping.postal_code),
    shipping_address: addresses.shipping,
    billing_address: addresses.billing,
    history: [
      {
        timestamp: new Date(),
        status: 'pending',
        comment: 'Order placed',
      },
    ],
    metadata: {
      source: 'web',
      ip_address: '192.168.1.1',
    },
  };

  // Insert order
  const order = await db
    .collection('orders')
    .insertOne(orderData, { session });

  // Update inventory
  const inventoryUpdates = cart.items.map((item) => {
    return {
      updateOne: {
        filter: { _id: item.id },
        update: {
          $inc: { 'inventory.quantity': -item.quantity },
          $set: { 'inventory.updated_at': new Date() },
        },
      },
    };
  });

  await db.collection('products').bulkWrite(inventoryUpdates, { session });

  // Update user orders index
  await db.collection('user_orders').updateOne(
    { _id: userId },
    {
```

```
        $push: {
          orders: {
            order_id: order.insertedId,
            order_number: orderData.order_number,
            date: orderData.created_at,
            total: orderData.pricing.total,
            status: orderData.status,
          },
        },
      },
      { upsert: true, session }
    );

    // Process payment
    const paymentResult = await processPayment(
      orderData.pricing.total,
      paymentDetails,
      { orderId: order.insertedId, orderNumber: orderData.order_number }
    );

    // Update order with payment info
    await db.collection('orders').updateOne(
      { _id: order.insertedId },
      {
        $set: {
          payment: {
            method: paymentDetails.method,
            transaction_id: paymentResult.transactionId,
            amount: orderData.pricing.total,
            currency: 'USD',
            card_brand: paymentResult.cardBrand,
            last4: paymentResult.last4,
          },
          payment_status: 'paid',
          status: 'processing',
          updated_at: new Date(),
          $push: {
            history: {
              timestamp: new Date(),
              status: 'processing',
              comment: 'Payment successful',
            },
          },
        },
      },
      { session }
    );

    // Clear cart
    await deleteCart(cartId);

    // Update real-time inventory in Redis
    const redisUpdates = cart.items.map((item) => {
      return [
        'HINCRBY',
```

```
        `product:${item.id}:inventory`,
        'quantity',
        -item.quantity,
      ];
    });

    // Add order to Redis for quick status lookups
    redisUpdates.push([
      'SETEX',
      `order:${order.insertedId}:status`,
      86400 * 30, // 30 days
      'processing',
    ]);

    await redis.multi(redisUpdates).exec();

    // Commit transaction
    await session.commitTransaction();

    return {
      order_id: order.insertedId,
      order_number: orderData.order_number,
      status: 'processing',
    };
  } catch (error) {
    await session.abortTransaction();
    throw error;
  } finally {
    session.endSession();
  }
}
```

2. **Efficient Order Lookup**:

```
// Order lookup by user (using index collection)
async function getUserOrders(userId, page = 1, limit = 10) {
  // Get order summaries from index collection
  const userOrderIndex = await db
    .collection('user_orders')
    .findOne({ _id: userId });

  if (!userOrderIndex || !userOrderIndex.orders.length) {
    return { orders: [], total: 0 };
  }

  // Sort by date and paginate
  const orders = userOrderIndex.orders
    .sort((a, b) => b.date - a.date)
    .slice((page - 1) * limit, page * limit);

  return {
    orders,
    total: userOrderIndex.orders.length,
```

```
  };
}

// Detailed order lookup (with Redis cache for active orders)
async function getOrderDetails(orderId) {
  // Try Redis cache first for recent/active orders
  const cachedStatus = await redis.get(`order:${orderId}:status`);

  if (
    cachedStatus &&
    cachedStatus !== 'completed' &&
    cachedStatus !== 'cancelled'
  ) {
    // Order is active, get fresh data from MongoDB
    return db.collection('orders').findOne({ _id: orderId });
  } else {
    // Check MongoDB cache collection for completed orders
    const cachedOrder = await db
      .collection('orders_cache')
      .findOne({ _id: orderId });

    if (cachedOrder) {
      return cachedOrder;
    }

    // Fall back to main orders collection
    return db.collection('orders').findOne({ _id: orderId });
  }
}
```

3. **Order Status Updates with History**:

```
// Update order status with atomic operations
async function updateOrderStatus(orderId, newStatus, comment) {
  const timestamp = new Date();

  // Update MongoDB
  const result = await db.collection('orders').updateOne(
    { _id: orderId },
    {
      $set: {
        status: newStatus,
        updated_at: timestamp,
      },
      $push: {
        history: {
          timestamp,
          status: newStatus,
          comment: comment || `Status updated to ${newStatus}`,
        },
      },
    }
  );
```

```
  // Update Redis cache
  if (result.modifiedCount > 0) {
    // Update status in Redis
    await redis.setex(`order:${orderId}:status`, 86400 * 30, newStatus);

    // Update user orders index
    await db
      .collection('user_orders')
      .updateOne(
        { 'orders.order_id': orderId },
        { $set: { 'orders.$.status': newStatus } }
      );

    return true;
  }

  return false;
}
```

Performance Optimization Strategies

To ensure our e-commerce application can handle high volumes, let's implement several performance optimization strategies:

1. Caching Strategy

```
// Multi-level caching for products
async function getProductDetails(productId) {
  const cacheKey = `product:${productId}:details`;

  // Level 1: Local memory cache (using node-cache)
  const localCached = memoryCache.get(cacheKey);
  if (localCached) {
    return localCached;
  }

  // Level 2: Redis cache
  const redisCached = await redis.get(cacheKey);
  if (redisCached) {
    const product = JSON.parse(redisCached);
    // Store in local cache for future requests
    memoryCache.set(cacheKey, product, 60); // 60 seconds
    return product;
  }

  // Level 3: Database
  const product = await db.collection('products').findOne({ _id: productId });

  if (product) {
    // Update caches
    redis.setex(cacheKey, 300, JSON.stringify(product)); // 5 minutes
```

```
    memoryCache.set(cacheKey, product, 60); // 60 seconds
  }

  return product;
}
```

2. Indexing Strategy

```
// Ensure proper indexing for all collections
function setupIndexes() {
  // Products collection indexes
  db.products.createIndex({ 'category.id': 1, 'price.current': 1 });
  db.products.createIndex({ 'brand.id': 1 });
  db.products.createIndex({ 'inventory.status': 1 });
  db.products.createIndex({ 'attributes.color': 1, 'attributes.size': 1 });
  db.products.createIndex({ slug: 1 }, { unique: true });

  // Full-text search index
  db.products.createIndex(
    {
      name: 'text',
      description: 'text',
      'metadata.search_keywords': 'text',
    },
    {
      weights: {
        name: 10,
        description: 5,
        'metadata.search_keywords': 3,
      },
    }
  );

  // Orders collection indexes
  db.orders.createIndex({ user_id: 1 });
  db.orders.createIndex({ order_number: 1 }, { unique: true });
  db.orders.createIndex({ created_at: 1 });
  db.orders.createIndex({ status: 1 });
  db.orders.createIndex({ 'payment.transaction_id': 1 });

  // User orders index
  db.user_orders.createIndex({ 'orders.order_number': 1 });
  db.user_orders.createIndex({ 'orders.date': 1 });
}
```

3. Read/Write Splitting

```
// Read/write separation for scaling
class ProductService {
  constructor() {
```

```
    // For high-volume services, use separate connections for reads and writes
    this.writeConnection = mongodb.connect(PRIMARY_CONNECTION_STRING);
    this.readConnection = mongodb.connect(REPLICA_CONNECTION_STRING);
  }

  async getProduct(productId) {
    // Use read replica for queries
    const db = await this.readConnection;
    return db.collection('products').findOne({ _id: productId });
  }

  async updateProductInventory(productId, quantity) {
    // Use primary for writes
    const db = await this.writeConnection;
    return db
      .collection('products')
      .updateOne(
        { _id: productId },
        { $inc: { 'inventory.quantity': quantity } }
      );
  }
}
```

4. Batch Processing

```
// Batch process inventory updates
async function processInventoryUpdates(updates) {
  // Group updates by product
  const productUpdates = {};

  updates.forEach((update) => {
    const { productId, quantity } = update;
    if (!productUpdates[productId]) {
      productUpdates[productId] = 0;
    }
    productUpdates[productId] += quantity;
  });

  // Prepare bulk operations
  const bulkOps = Object.entries(productUpdates).map(
    ([productId, quantity]) => {
      return {
        updateOne: {
          filter: { _id: productId },
          update: {
            $inc: { 'inventory.quantity': quantity },
            $set: { 'inventory.updated_at': new Date() },
          },
        },
      };
    }
  );
```

```
  // Execute bulk update
  if (bulkOps.length > 0) {
    return db.collection('products').bulkWrite(bulkOps);
  }

  return { acknowledged: true, modifiedCount: 0 };
}
```

5. Query Optimization for Analytics

```
// Optimize analytics queries with materialized views
async function updateDailySalesSnapshot() {
  const yesterday = new Date();
  yesterday.setDate(yesterday.getDate() - 1);
  yesterday.setHours(0, 0, 0, 0);

  const endOfYesterday = new Date(yesterday);
  endOfYesterday.setHours(23, 59, 59, 999);

  // Aggregate sales data
  const salesData = await db
    .collection('orders')
    .aggregate([
      {
        $match: {
          created_at: { $gte: yesterday, $lte: endOfYesterday },
          status: { $nin: ['cancelled', 'refunded'] },
        },
      },
      {
        $group: {
          _id: null,
          order_count: { $sum: 1 },
          total_revenue: { $sum: '$pricing.total' },
          item_count: { $sum: { $size: '$items' } },
        },
      },
      {
        $project: {
          _id: 0,
          date: yesterday,
          order_count: 1,
          total_revenue: 1,
          item_count: 1,
          average_order_value: { $divide: ['$total_revenue', '$order_count'] },
        },
      },
    ])
    .toArray();

  // Category breakdown
  const categorySales = await db
    .collection('orders')
```

```
    .aggregate([
      {
        $match: {
          created_at: { $gte: yesterday, $lte: endOfYesterday },
          status: { $nin: ['cancelled', 'refunded'] },
        },
      },
      { $unwind: '$items' },
      {
        $lookup: {
          from: 'products',
          localField: 'items.product_id',
          foreignField: '_id',
          as: 'product',
        },
      },
      { $unwind: '$product' },
      {
        $group: {
          _id: '$product.category.id',
          category_name: { $first: '$product.category.name' },
          order_count: { $sum: 1 },
          item_count: { $sum: '$items.quantity' },
          revenue: {
            $sum: { $multiply: ['$items.unit_price', '$items.quantity'] },
          },
        },
      },
      { $sort: { revenue: -1 } },
      { $limit: 10 },
    ])
    .toArray();

  // Insert into sales snapshot collection
  await db.collection('sales_snapshots').insertOne({
    date: yesterday,
    ...salesData[0],
    top_categories: categorySales,
  });
}
```

Summary and Best Practices

Here are the key principles that guided our e-commerce NoSQL database design:

1. **Access Pattern-Driven Design**: We designed the schema based on how data will be accessed, not just how it naturally relates.
2. **Strategic Denormalization**: We duplicated data in specific places (user info in orders, product details in cart items) to optimize read performance.
3. **Hybrid Database Approach**: Using MongoDB for complex document storage and Redis for real-time data and caching.

4. **Multi-Level Caching**: Implementing memory, Redis, and database caching layers for different access patterns.
5. **Optimized Indexing**: Creating targeted indexes based on query patterns to ensure performance.
6. **Consistency Management**: Using transactions for critical operations like order creation while using eventual consistency for less critical operations.
7. **Batch Processing**: Implementing bulk writes and pipelines to minimize network round trips.
8. **Real-Time vs. Historical Data Separation**: Keeping active/hot data in Redis and historical data in MongoDB.
9. **Analytics Optimization**: Creating materialized views and snapshots for efficient reporting queries.
10. **Scalability Considerations**: Designing for horizontal scaling with proper key distribution and read/write splitting.

This comprehensive approach creates a database design that can efficiently handle high-volume e-commerce workloads while remaining maintainable and scalable as the application grows.

Q: Explain the concept of eventual consistency in NoSQL databases and discuss strategies for handling consistency issues in a distributed database system. Provide examples of scenarios where eventual consistency might be problematic and how to address them.

A: Eventual consistency is a consistency model that allows database nodes to temporarily diverge in their views of data, with the guarantee that they will eventually converge to the same state. This concept is fundamental to distributed NoSQL databases and represents a critical trade-off in the context of the CAP theorem. Let me explain the concept in depth and discuss strategies for handling the challenges it presents.

Understanding Eventual Consistency

Formal Definition

Eventual consistency guarantees that, given no new updates to an object, all accesses to that object will eventually return the latest updated value. In other words, if we stop writing to the database, all reads will eventually return consistent results.

Key characteristics:

- Updates propagate asynchronously through the system
- Different nodes may return different values during the convergence period
- No guarantee about how long convergence will take (though practical systems define reasonable bounds)
- The system is always available for both reads and writes

Comparison with Strong Consistency

To understand eventual consistency, it's helpful to contrast it with strong consistency:

Aspect	Strong Consistency	Eventual Consistency
Read Guarantees	All reads reflect the most recent write	Reads might return stale data

Aspect	Strong Consistency	Eventual Consistency
Write Acknowledgment	Must wait for replication to quorum/all nodes	Can acknowledge immediately after local write
Availability During Partitions	May become unavailable to maintain consistency	Remains available (sacrifices consistency)
Performance	Higher latency	Lower latency
Use Cases	Financial transactions, inventory, authentication	Social media, content delivery, analytics

CAP Theorem Context

The CAP theorem states that during a network partition, a distributed system must choose between consistency and availability. Eventual consistency prioritizes availability over consistency, making it a characteristic of "AP" (Available, Partition-tolerant) systems in CAP terms.

How Eventual Consistency Works

1. Conflict Detection

When multiple writes occur concurrently, the system needs mechanisms to detect conflicts:

```
Time:  t1        t2        t3        t4
Node1: [A:1] --> [A:2] --------------->
Node2: [A:1] ----------> [A:3] -------> Conflict between A:2 and A:3
Node3: [A:1] -------------------------->
```

2. Conflict Resolution

Once conflicts are detected, they must be resolved:

Last-Write-Wins (Timestamp-based):

```
Node1: [A:2] @timestamp 100
Node2: [A:3] @timestamp 105

Resolution: A:3 wins (later timestamp)
```

Vector Clocks:

```
Node1: [A:value1] @version [1,0,0]
Node2: [A:value2] @version [0,1,0]
Node3: [A:value3] @version [1,1,0]

Resolution: value3 wins (descendent of both value1 and value2)
```

3. Propagation Mechanisms

Updates must propagate throughout the system:

- **Anti-entropy process**: Background process that synchronizes data between nodes
- **Read repair**: Fixing inconsistencies when they're detected during reads
- **Gossip protocols**: Nodes periodically exchange state information

4. Consistency Spectrum in Practice

Databases often provide options along a consistency spectrum:

Cassandra Consistency Levels:

```
// Strongest consistency - all nodes must acknowledge
statement.setConsistencyLevel(ConsistencyLevel.ALL);

// Middle ground - quorum must acknowledge
statement.setConsistencyLevel(ConsistencyLevel.QUORUM);

// Weakest consistency - single node acknowledges
statement.setConsistencyLevel(ConsistencyLevel.ONE);
```

DynamoDB:

```
// Eventually consistent read (default)
const params = {
  TableName: 'Products',
  Key: { ProductId: '1234' },
};

// Strongly consistent read
const params = {
  TableName: 'Products',
  Key: { ProductId: '1234' },
  ConsistentRead: true,
};
```

Problematic Scenarios and Solutions

Let's examine scenarios where eventual consistency can cause issues and strategies to address them:

Scenario 1: User Profile Updates and Immediate Display

Problem: A user updates their profile information and is redirected to their profile page, but sees the old data because the read happens on a node that hasn't received the update yet.

```
Time: t1          t2          t3
      User updates Profile page Node hasn't
      profile     loads       received update yet
      [Node1]     [Node2]     [Node2 shows old data]
```

Solutions:

1. **Read-After-Write Consistency**: Direct reads to the same node that handled the write or to nodes known to have the update.

```
// Track write node in session
async function updateProfile(userId, profileData) {
  const result = await db
    .collection('users')
    .updateOne({ _id: userId }, { $set: profileData });

  // Store write version or timestamp in session
  session.lastWriteTime = Date.now();
  return result;
}

// Route subsequent reads to appropriate nodes
async function getProfile(userId) {
  const options = {};

  // If this is right after a write, ensure consistency
  if (session.lastWriteTime && Date.now() - session.lastWriteTime < 5000) {
    options.readPreference = 'primary'; // Read from primary in MongoDB
    // or options.consistentRead = true; // Use strong consistency in DynamoDB
  }

  return db.collection('users').findOne({ _id: userId }, options);
}
```

2. **Client-Side Caching**: Have the client cache the written value and use it locally before the system converges.

```
// Client-side state management with React
function ProfilePage() {
  const [profile, setProfile] = useState(null);
  const [pendingUpdates, setPendingUpdates] = useState({});

  useEffect(() => {
    // Fetch profile from server
    fetchProfile().then((serverProfile) => {
      // Merge with any pending updates
      setProfile({ ...serverProfile, ...pendingUpdates });
    });
  }, []);
```

```
  const updateProfile = async (updates) => {
    // Optimistically update local state
    setPendingUpdates({ ...pendingUpdates, ...updates });
    setProfile({ ...profile, ...updates });

    // Perform actual update
    await api.updateProfile(updates);

    // Clear pending updates after confirmation
    setPendingUpdates({});
  };

  // Render profile with pending updates applied
}
```

3. **Version-Based Approach**: Include a version number with each update and only display newer versions.

```
async function updateProfile(userId, profileData) {
  // Get current version and increment
  const user = await db.collection('users').findOne({ _id: userId });
  const newVersion = (user.version || 0) + 1;

  // Update with new version
  const result = await db.collection('users').updateOne(
    { _id: userId },
    {
      $set: { ...profileData, version: newVersion },
      $currentDate: { updated_at: true },
    }
  );

  return { ...result, version: newVersion };
}

// Client side state management
let currentProfileVersion = 0;

function handleProfileData(profileData) {
  // Only update UI if we receive a newer version
  if (profileData.version > currentProfileVersion) {
    currentProfileVersion = profileData.version;
    updateUI(profileData);
  }
}
```

Scenario 2: Inventory Management and Overselling

Problem: Multiple customers might be able to purchase the last item in stock because nodes haven't synchronized the decreased inventory count.

```
Time: t1             t2             t3
      Customer1 buys  Customer2 buys Both orders
      last item       "last" item    confirmed
      [Node1]         [Node2]        [Oversold!]
```

Solutions:

1. **Centralized Inventory Management**: Route all inventory operations to a single node or shard to maintain strong consistency.

```
// Use sharding/routing to direct all operations for a product to the same node
function getInventoryShardKey(productId) {
  // Consistent hash to always route same product to same shard
  return crypto
    .createHash('md5')
    .update(productId)
    .digest('hex')
    .substring(0, 8);
}

async function checkAndReserveInventory(productId, quantity) {
  const shardKey = getInventoryShardKey(productId);

  // Connect to the specific shard for this product
  const shard = getShardConnection(shardKey);

  // Perform atomic check and update on this shard
  const result = await shard.runCommand({
    findAndModify: 'inventory',
    query: {
      productId: productId,
      availableQuantity: { $gte: quantity },
    },
    update: {
      $inc: { availableQuantity: -quantity, reservedQuantity: quantity },
    },
    new: true,
  });

  if (result.value) {
    return { success: true, remainingQuantity: result.value.availableQuantity };
  } else {
    return { success: false, message: 'Insufficient inventory' };
  }
}
```

2. **Optimistic Concurrency Control**: Use version checks to ensure the inventory hasn't changed since it was last read.

```
async function reserveInventory(productId, quantity, retries = 3) {
  for (let attempt = 0; attempt < retries; attempt++) {
    // Get current inventory with version
    const inventory = await db.collection('inventory').findOne({ productId });

    if (inventory.quantity < quantity) {
      return { success: false, message: 'Insufficient inventory' };
    }

    // Try to update with version check
    const result = await db.collection('inventory').updateOne(
      {
        productId,
        version: inventory.version, // Concurrency control
      },
      {
        $inc: { quantity: -quantity },
        $set: { version: inventory.version + 1 },
      }
    );

    if (result.modifiedCount === 1) {
      return { success: true };
    }

    // If update failed, retry (version mismatch means someone else updated)
    console.log(`Concurrency conflict on attempt ${attempt + 1}, retrying...`);
  }

  return { success: false, message: 'Failed due to concurrent updates' };
}
```

3. **Two-Phase Inventory Operations**: Split inventory operations into reservation and confirmation phases.

```
// Phase 1: Reserve inventory with expiration
async function reserveInventory(productId, quantity, orderId) {
  const result = await db.collection('inventory').updateOne(
    { productId, availableQuantity: { $gte: quantity } },
    {
      $inc: { availableQuantity: -quantity },
      $push: {
        reservations: {
          orderId: orderId,
          quantity: quantity,
          timestamp: new Date(),
          expiresAt: new Date(Date.now() + 15 * 60000), // 15 minutes
        },
      },
    }
  );

  return { success: result.modifiedCount === 1 };
```

```
}

// Phase 2: Confirm reservation
async function confirmReservation(productId, orderId) {
  const result = await db.collection('inventory').updateOne(
    {
      productId,
      'reservations.orderId': orderId,
    },
    {
      $pull: { reservations: { orderId: orderId } },
      $inc: { committedQuantity: quantity },
    }
  );

  return { success: result.modifiedCount === 1 };
}

// Cleanup expired reservations (run periodically)
async function cleanupExpiredReservations() {
  const now = new Date();

  const expiredReservations = await db
    .collection('inventory')
    .find({
      'reservations.expiresAt': { $lt: now },
    })
    .toArray();

  for (const item of expiredReservations) {
    const expired = item.reservations.filter((r) => r.expiresAt < now);
    const quantityToRestore = expired.reduce((sum, r) => sum + r.quantity, 0);

    await db.collection('inventory').updateOne(
      { productId: item.productId },
      {
        $inc: { availableQuantity: quantityToRestore },
        $pull: { reservations: { expiresAt: { $lt: now } } },
      }
    );
  }
}
```

Scenario 3: Comment Ordering on Social Media

Problem: On a social media platform, comments may appear in different orders to different users, or a user's comment might not appear in their own view immediately.

```
Time: t1            t2            t3
      User posts    User refreshes Comment isn't
      comment       to see it      visible yet
      [Node1]       [Node2]        [Node3 hasn't
                                   received it yet]
```

Solutions:

1. **Client-Side Comment Composition**: Have the client add the new comment to the UI immediately while waiting for server confirmation.

```
async function addComment(postId, commentText) {
  // Generate a temporary client-side ID
  const tempId = 'temp-' + Date.now();

  // Add to local state immediately
  const optimisticComment = {
    id: tempId,
    text: commentText,
    user: currentUser,
    timestamp: new Date(),
    pending: true,
  };

  // Update UI immediately
  addCommentToUI(optimisticComment);

  try {
    // Send to server
    const result = await api.createComment(postId, commentText);

    // Replace temporary comment with real one
    updateCommentInUI(tempId, {
      id: result.id,
      text: result.text,
      user: result.user,
      timestamp: result.timestamp,
      pending: false,
    });
  } catch (error) {
    // Handle error - mark comment as failed
    updateCommentInUI(tempId, { error: true, errorMessage: error.message });
  }
}
```

2. **Comment Versioning and Ordering**: Use logical timestamps or sequence numbers to ensure comments are displayed in the same order everywhere.

```
// Server-side
async function createComment(postId, userId, text) {
  // Get current highest sequence number for this post
  const post = await db
    .collection('posts')
    .findOne({ _id: postId }, { projection: { commentSequence: 1 } });

  const sequence = (post.commentSequence || 0) + 1;
```

```
  // Update post with new sequence number
  await db
    .collection('posts')
    .updateOne({ _id: postId }, { $set: { commentSequence: sequence } });

  // Create comment with sequence
  const comment = {
    postId,
    userId,
    text,
    sequence,
    createdAt: new Date(),
  };

  await db.collection('comments').insertOne(comment);

  return comment;
}

// Client-side - always sort by sequence
function getComments(postId) {
  return db
    .collection('comments')
    .find({ postId })
    .sort({ sequence: 1 })
    .toArray();
}
```

3. **Comment Source Routing**: Direct comment requests to the node that owns the original post.

```
// Consistent routing for post and its comments
function getPostShardKey(postId) {
  return crypto.createHash('md5').update(postId).digest('hex').substring(0, 8);
}

function getShardForPost(postId) {
  const shardKey = getPostShardKey(postId);
  return shards[shardKey % shards.length];
}

async function addCommentToPost(postId, userId, commentText) {
  // Route to the shard that owns this post
  const shard = getShardForPost(postId);

  // All operations for this post happen on the same shard
  return shard.runCommand({
    insert: 'comments',
    documents: [
      {
        postId,
        userId,
        text: commentText,
```

```
        createdAt: new Date(),
      },
    ],
  });
}
```

Scenario 4: Counters and Aggregations (Likes, Follower Counts)

Problem: Distributed counters (such as likes, views, or follower counts) can be inconsistent across nodes, leading to fluctuating counts or incorrect aggregations.

```
Time: t1            t2            t3
      User1 likes   User2 likes   Both users see
      post (count=1) post (count=1) different counts
      [Node1]       [Node2]        [Node1: 1, Node2: 1]
```

Solutions:

1. **Eventual Consistency with Delta Updates**: Use specialized data structures that can merge concurrent updates correctly.

```
// Using a CRDT (Conflict-free Replicated Data Type) counter
function incrementCounter(counterId, amount = 1) {
  // Each node maintains its own counter
  const nodeId = getNodeId();

  // Update the local increment count for this node
  db.collection('counters').updateOne(
    { _id: counterId },
    {
      $inc: { [`increments.${nodeId}`]: amount },
      $currentDate: { lastUpdated: true },
    },
    { upsert: true }
  );

  // Return the approximate count (may be stale)
  return getApproximateCount(counterId);
}

// Get total by summing all node contributions
async function getApproximateCount(counterId) {
  const counter = await db.collection('counters').findOne({ _id: counterId });

  if (!counter || !counter.increments) {
    return 0;
  }

  // Sum all node contributions
  return Object.values(counter.increments).reduce(
```

```
    (sum, value) => sum + value,
    0
  );
}

// Periodic reconciliation process
async function reconcileCounters() {
  const counters = await db.collection('counters').find({}).toArray();

  for (const counter of counters) {
    // Sum all increments across nodes
    const total = Object.values(counter.increments || {}).reduce(
      (sum, val) => sum + val,
      0
    );

    // Update the reconciled total
    await db.collection('counters_reconciled').updateOne(
      { _id: counter._id },
      {
        $set: {
          total: total,
          last_reconciled: new Date(),
        },
      },
      { upsert: true }
    );
  }
}
```

2. **Two-Tier Counter Architecture**: Use local counters for writes and a global counter for eventually consistent reads.

```
// Increment local shard counter
async function incrementCounter(counterId, amount = 1) {
  const shardId = getShardId();

  await db
    .collection('counter_shards')
    .updateOne(
      { counterId, shardId },
      { $inc: { count: amount } },
      { upsert: true }
    );

  // Approximate increment of global counter for immediate feedback
  await db.collection('counters_global').updateOne(
    { _id: counterId },
    {
      $inc: { approximateCount: amount },
      $set: { needsReconciliation: true },
    },
    { upsert: true }
```

```
    );
  }

  // Read from global counter
  async function getCounter(counterId) {
    const counter = await db
      .collection('counters_global')
      .findOne({ _id: counterId });
    return counter ? counter.count || counter.approximateCount || 0 : 0;
  }

  // Periodic reconciliation job
  async function reconcileCounters() {
    // Find counters needing reconciliation
    const countersToUpdate = await db
      .collection('counters_global')
      .find({ needsReconciliation: true })
      .toArray();

    for (const counter of countersToUpdate) {
      // Get all shards for this counter
      const shards = await db
        .collection('counter_shards')
        .find({ counterId: counter._id })
        .toArray();

      // Sum all shard counts
      const total = shards.reduce((sum, shard) => sum + shard.count, 0);

      // Update the global counter
      await db.collection('counters_global').updateOne(
        { _id: counter._id },
        {
          $set: {
            count: total,
            approximateCount: total,
            needsReconciliation: false,
            lastReconciled: new Date(),
          },
        }
      );
    }
  }
```

3. **Probabilistic Counting for High-Volume Metrics**: For metrics where absolute precision isn't required, use probabilistic data structures.

```
// Using HyperLogLog for approximate counting
const redis = require('redis');
const client = redis.createClient();

// Add to the counter (incredibly scalable)
async function incrementViewCounter(contentId, userId) {
```

```
  // HyperLogLog can count unique items with minimal memory
  await client.pfadd(`views:${contentId}`, userId);

  // For immediate feedback, also update a regular counter
  // that might be slightly off due to eventual consistency
  await client.incr(`views_approx:${contentId}`);
}

// Get the approximate unique count (very accurate for large counts)
async function getUniqueViewCount(contentId) {
  return client.pfcount(`views:${contentId}`);
}

// Get the approximate raw count (faster but less accurate)
async function getApproximateViewCount(contentId) {
  return client.get(`views_approx:${contentId}`);
}
```

Real-World Examples of Consistency Strategies

Here are examples of how major platforms handle eventual consistency:

Amazon Shopping Cart

Amazon uses a highly available shopping cart that prioritizes availability over consistency:

```
// Amazon-style shopping cart with conflict resolution
class ShoppingCartService {
  async addItem(userId, item) {
    // Write to multiple regions for availability
    await Promise.all([
      this.primaryRegion.addToCart(userId, item),
      this.backupRegion.addToCart(userId, item),
    ]);

    // Return success fast - eventual consistency is fine for cart
    return { success: true };
  }

  async getCart(userId) {
    try {
      // Try primary region first
      return await this.primaryRegion.getCart(userId);
    } catch (error) {
      // Fall back to backup region
      return await this.backupRegion.getCart(userId);
    }
  }

  // During checkout, resolve any conflicts
  async finalizeCart(userId) {
    const primaryCart = await this.primaryRegion.getCart(userId);
    const backupCart = await this.backupRegion.getCart(userId);
```

```
    // Merge carts, taking the union of all items
    const mergedItems = this.mergeCartItems(
      primaryCart.items,
      backupCart.items
    );

    // Write back the reconciled cart to both regions
    await Promise.all([
      this.primaryRegion.updateCart(userId, { items: mergedItems }),
      this.backupRegion.updateCart(userId, { items: mergedItems }),
    ]);

    return { items: mergedItems };
  }

  // Merge strategy: take max quantity for each item
  mergeCartItems(items1, items2) {
    const itemMap = {};

    // Process all items from first cart
    items1.forEach((item) => {
      itemMap[item.productId] = item;
    });

    // Merge with items from second cart
    items2.forEach((item) => {
      if (itemMap[item.productId]) {
        // Take max quantity
        itemMap[item.productId].quantity = Math.max(
          itemMap[item.productId].quantity,
          item.quantity
        );
      } else {
        itemMap[item.productId] = item;
      }
    });

    return Object.values(itemMap);
  }
}
```

Facebook's Notification System

Facebook implements a tiered consistency model for different features:

```
// Facebook-style eventual consistency for notifications
class NotificationService {
  constructor() {
    this.localCounters = {};
    this.reconciliationInterval = setInterval(
      () => this.reconcileCounters(),
      60000
```

```
    );
  }

  // Low consistency - notification counter can temporarily differ across devices
  async incrementNotificationCounter(userId) {
    // Update local counter immediately for responsiveness
    if (!this.localCounters[userId]) {
      this.localCounters[userId] = 0;
    }
    this.localCounters[userId]++;

    // Fire and forget update to global counter
    this.globalDatabase
      .incrementCounter(`notifications:${userId}`, 1, {
        consistencyLevel: 'EVENTUAL',
      })
      .catch((err) => console.error('Counter update failed:', err));

    return this.localCounters[userId];
  }

  // Medium consistency - message delivery status
  async markMessageAsDelivered(messageId, recipientId) {
    // Update with higher consistency level
    return this.globalDatabase.updateMessage(
      messageId,
      { deliveryStatus: 'delivered', deliveredAt: new Date() },
      { consistencyLevel: 'SESSION' }
    );
  }

  // Strong consistency - privacy settings
  async updatePrivacySettings(userId, settings) {
    // Critical settings use strong consistency
    return this.globalDatabase.updateUserSettings(
      userId,
      { privacySettings: settings },
      { consistencyLevel: 'STRONG' }
    );
  }

  // Periodic reconciliation of counters
  async reconcileCounters() {
    const userIds = Object.keys(this.localCounters);

    for (const userId of userIds) {
      try {
        // Get authoritative count from database
        const globalCount = await this.globalDatabase.getCounter(
          `notifications:${userId}`,
          { consistencyLevel: 'STRONG' }
        );

        // Update local cache with authoritative value
        this.localCounters[userId] = globalCount;
```

```
        } catch (error) {
          console.error(`Failed to reconcile counter for user ${userId}:`, error);
        }
      }
    }
  }
```

Best Practices for Managing Eventual Consistency

1. **Identify Consistency Requirements by Feature**: Not every feature needs the same consistency level. Categorize your data and operations:
    - Strong consistency: Authentication, payments, critical user settings
    - Eventual consistency: Social features, analytics, recommendations
2. **Design for Idempotence**: Operations should be safely repeatable without causing errors or corruption.

```
// Idempotent API design
app.put('/api/orders/:orderId/status', async (req, res) => {
  const { orderId } = req.params;
  const { status, version } = req.body;

  try {
    // Include version check for safety
    const result = await db.collection('orders').updateOne(
      {
        _id: orderId,
        // Only update if the current version is less than or equal to our version
        $or: [{ version: { $lt: version } }, { version: { $exists: false } }],
      },
      {
        $set: {
          status: status,
          version: version,
          updatedAt: new Date(),
        },
      }
    );

    if (result.matchedCount === 0) {
      // Handle version conflict or non-existent order
      return res.status(409).json({
        error: 'Conflict',
        message: 'The order has been modified with a newer version',
      });
    }

    return res.status(200).json({ success: true });
  } catch (error) {
    return res.status(500).json({ error: error.message });
  }
});
```

3. **Implement Conflict Resolution Strategies**: Choose appropriate conflict resolution based on data type:
    - Last-writer-wins (timestamp-based)
    - Merge semantics (combining changes)
    - Application-specific resolution logic
4. **Make Consistency Visible to Users**: When appropriate, expose consistency status to users to manage expectations.

```
// React component showing consistency status
function CommentSection({ postId }) {
  const [comments, setComments] = useState([]);
  const [pendingComments, setPendingComments] = useState([]);
  const [syncStatus, setSyncStatus] = useState('synced'); // 'synced', 'syncing',
'offline'

  // Add new comment
  const addComment = async (text) => {
    // Generate temporary ID
    const tempId = `temp-${Date.now()}`;

    // Add to pending comments immediately
    const tempComment = {
      id: tempId,
      text,
      author: currentUser,
      createdAt: new Date(),
      pending: true,
    };

    setPendingComments([...pendingComments, tempComment]);
    setSyncStatus('syncing');

    try {
      // Send to server
      const savedComment = await api.createComment(postId, text);

      // Replace temporary comment with saved one
      setPendingComments(pendingComments.filter((c) => c.id !== tempId));
      setComments([...comments, savedComment]);
      setSyncStatus('synced');
    } catch (error) {
      // Mark as failed but keep in list
      setPendingComments(
        pendingComments.map((c) =>
          c.id === tempId
            ? { ...c, error: true, errorMessage: error.message }
            : c
        )
      );
      setSyncStatus('offline');
    }
  };
```

```
  // Render comments with visual indicators for pending/failed
  return (
    <div className="comment-section">
      <div className="sync-status">
        {syncStatus === 'synced' && (
          <span className="synced">All changes saved</span>
        )}
        {syncStatus === 'syncing' && (
          <span className="syncing">Saving changes...</span>
        )}
        {syncStatus === 'offline' && (
          <span className="offline">Working offline</span>
        )}
      </div>

      <div className="comments-list">
        {comments.map((comment) => (
          <Comment key={comment.id} comment={comment} />
        ))}

        {pendingComments.map((comment) => (
          <Comment
            key={comment.id}
            comment={comment}
            isPending={!comment.error}
            hasError={comment.error}
            errorMessage={comment.errorMessage}
          />
        ))}
      </div>

      <CommentForm onSubmit={addComment} disabled={syncStatus === 'offline'} />
    </div>
  );
}
```

5. **Implement Background Reconciliation**: Use background processes to detect and resolve inconsistencies.

```
// Background reconciliation service
class DataReconciliationService {
  constructor(db, options = {}) {
    this.db = db;
    this.interval = options.interval || 3600000; // Default: hourly
    this.batchSize = options.batchSize || 100;
    this.running = false;
  }

  start() {
    this.timer = setInterval(() => this.runReconciliation(), this.interval);
    console.log('Reconciliation service started');
  }

  stop() {
```

```
    if (this.timer) {
      clearInterval(this.timer);
      this.timer = null;
      console.log('Reconciliation service stopped');
    }
  }

  async runReconciliation() {
    if (this.running) {
      console.log('Reconciliation already in progress, skipping');
      return;
    }

    this.running = true;
    console.log('Starting reconciliation process');

    try {
      // Find records that need reconciliation
      const recordsToCheck = await this.db
        .collection('reconciliation_queue')
        .find({ nextCheckTime: { $lte: new Date() } })
        .limit(this.batchSize)
        .toArray();

      console.log(`Processing ${recordsToCheck.length} records`);

      for (const record of recordsToCheck) {
        await this.reconcileRecord(record);
      }

      console.log('Reconciliation completed successfully');
    } catch (error) {
      console.error('Error during reconciliation:', error);
    } finally {
      this.running = false;
    }
  }

  async reconcileRecord(record) {
    console.log(`Reconciling record ${record._id} of type ${record.type}`);

    try {
      switch (record.type) {
        case 'user_counter':
          await this.reconcileUserCounter(record.userId, record.counterName);
          break;
        case 'inventory':
          await this.reconcileInventory(record.productId);
          break;
        // Add other reconciliation types as needed
        default:
          console.warn(`Unknown reconciliation type: ${record.type}`);
      }

      // Update next check time
```

```
      await this.db.collection('reconciliation_queue').updateOne(
        { _id: record._id },
        {
          $set: {
            lastCheckTime: new Date(),
            nextCheckTime: new Date(Date.now() + record.checkInterval),
            lastStatus: 'success',
          },
        }
      );
    } catch (error) {
      console.error(`Failed to reconcile record ${record._id}:`, error);

      // Update with error status
      await this.db.collection('reconciliation_queue').updateOne(
        { _id: record._id },
        {
          $set: {
            lastCheckTime: new Date(),
            nextCheckTime: new Date(Date.now() + 60000), // Retry in 1 minute
            lastStatus: 'error',
            lastError: error.message,
          },
          $inc: { errorCount: 1 },
        }
      );
    }
  }

  // Example reconciliation logic for user counters
  async reconcileUserCounter(userId, counterName) {
    // Get all counter fragments from different shards
    const shards = await this.db
      .collection('counter_shards')
      .find({ userId, counterName })
      .toArray();

    // Calculate the true count
    const totalCount = shards.reduce((sum, shard) => sum + shard.count, 0);

    // Update the master counter
    await this.db
      .collection('user_counters')
      .updateOne(
        { userId, counterName },
        { $set: { count: totalCount, lastReconciled: new Date() } },
        { upsert: true }
      );

    console.log(`Reconciled ${counterName} for user ${userId}: ${totalCount}`);
  }

  // Example reconciliation logic for inventory
  async reconcileInventory(productId) {
    // Get reserved inventory from all order shards
```

```
    const reservations = await this.db
      .collection('inventory_reservations')
      .find({ productId, status: 'active' })
      .toArray();

    // Calculate total reserved
    const totalReserved = reservations.reduce(
      (sum, res) => sum + res.quantity,
      0
    );

    // Get physical inventory
    const inventory = await this.db
      .collection('inventory')
      .findOne({ productId });

    if (!inventory) {
      throw new Error(`Inventory record not found for product ${productId}`);
    }

    // Calculate available inventory
    const available = inventory.physicalCount - totalReserved;

    // Update available count
    await this.db.collection('inventory').updateOne(
      { productId },
      {
        $set: {
          availableCount: available,
          reservedCount: totalReserved,
          lastReconciled: new Date(),
        },
      }
    );

    console.log(
      `Reconciled inventory for product ${productId}: ${available} available`
    );
  }
}
```

4.6 Query Patterns and Optimization

NoSQL databases require different query patterns and optimization techniques compared to relational databases. Let's explore common patterns and optimization strategies.

Document Database Query Patterns (MongoDB)

1. **Single Document Queries**: The most efficient pattern, retrieving one document by its primary identifier.

```
// Find a user by ID (most efficient)
db.users.findOne({ _id: ObjectId('507f1f77bcf86cd799439011') });
```

```
// Find a user by unique email (efficient with index)
db.users.findOne({ email: 'user@example.com' });
```

2. **Multi-Document Queries with Filtering**: Retrieve multiple documents based on filtering criteria.

```
// Find all active users in a specific city
db.users
  .find({
    status: 'active',
    'address.city': 'New York',
  })
  .sort({ lastName: 1 });

// Find products below a certain price with specific attributes
db.products
  .find({
    category: 'electronics',
    price: { $lt: 500 },
    'attributes.color': 'black',
  })
  .limit(20);
```

3. **Aggregation Queries**: Perform complex transformations and calculations on data.

```
// Calculate average order value by customer segment
db.orders.aggregate([
  // Match orders from the last 30 days
  {
    $match: {
      orderDate: { $gte: new Date(Date.now() - 30 * 24 * 60 * 60 * 1000) },
    },
  },
  // Join with customers collection to get segment
  {
    $lookup: {
      from: 'customers',
      localField: 'customerId',
      foreignField: '_id',
      as: 'customer',
    },
  },
  // Unwind the customer array (from lookup)
  { $unwind: '$customer' },
  // Group by customer segment and calculate metrics
  {
    $group: {
      _id: '$customer.segment',
      averageOrderValue: { $avg: '$totalAmount' },
      orderCount: { $sum: 1 },
      totalRevenue: { $sum: '$totalAmount' },
    },
```

```
  },
  // Sort by total revenue
  { $sort: { totalRevenue: -1 } },
]);
```

4. **Faceted Search**: Enable search with dynamic filtering options.

```
// Product search with facets for filters
db.products.aggregate([
  // Match stage - initial filtering
  {
    $match: {
      category: 'clothing',
      active: true,
    },
  },
  // Facet stage - compute multiple aggregations in one query
  {
    $facet: {
      // Results - the actual products to display
      results: [
        { $sort: { popularity: -1 } },
        { $limit: 20 },
        {
          $project: {
            name: 1,
            price: 1,
            imageUrl: 1,
            rating: 1,
          },
        },
      ],
      // Facets - for filter UI
      categoryFacets: [
        { $group: { _id: '$subCategory', count: { $sum: 1 } } },
        { $sort: { count: -1 } },
      ],
      brandFacets: [
        { $group: { _id: '$brand', count: { $sum: 1 } } },
        { $sort: { count: -1 } },
      ],
      priceFacets: [
        {
          $bucket: {
            groupBy: '$price',
            boundaries: [0, 25, 50, 100, 200, 500],
            default: '500+',
            output: { count: { $sum: 1 } },
          },
        },
      ],
    },
```

```
  },
]);
```

5. **Geospatial Queries**: Find documents based on geographic location.

```
// Find restaurants within 5km of a location
db.restaurants.find({
  location: {
    $near: {
      $geometry: {
        type: 'Point',
        coordinates: [-73.9667, 40.78], // Longitude, latitude
      },
      $maxDistance: 5000, // 5km in meters
    },
  },
});

// Find stores within a polygon (service area)
db.stores.find({
  location: {
    $geoWithin: {
      $geometry: {
        type: 'Polygon',
        coordinates: [
          [
            [-74.0, 40.7],
            [-74.0, 40.8],
            [-73.9, 40.8],
            [-73.9, 40.7],
            [-74.0, 40.7],
          ],
        ],
      },
    },
  },
});
```

Key-Value Database Query Patterns (Redis, DynamoDB)

1. **Simple Key-Value Operations**: Direct access by exact key.

```
// Redis key-value operations
// Set user profile
await redis.set(`user:${userId}:profile`, JSON.stringify(userProfile));

// Get user profile
const userProfileJson = await redis.get(`user:${userId}:profile`);
const userProfile = JSON.parse(userProfileJson);

// DynamoDB key-value operations
```

```
// Get product by ID
const params = {
  TableName: 'Products',
  Key: {
    ProductId: 'ABC123',
  },
};
const result = await dynamoDb.get(params).promise();
const product = result.Item;
```

2. **Compound Key Patterns**: Using structured keys to facilitate more complex queries.

```
// Redis compound keys
// Store order by user and date
const orderKey = `orders:user:${userId}:date:${orderDate}:id:${orderId}`;
await redis.set(orderKey, JSON.stringify(orderDetails));

// Get all keys matching a pattern (user's orders in a date range)
const keys = await redis.keys(`orders:user:${userId}:date:2023-06*`);

// DynamoDB composite keys
const params = {
  TableName: 'UserOrders',
  KeyConditionExpression:
    'UserId = :uid AND begins_with(OrderDateId, :datePrefix)',
  ExpressionAttributeValues: {
    ':uid': userId,
    ':datePrefix': '2023-06',
  },
};
const result = await dynamoDb.query(params).promise();
```

3. **List and Set Operations**: Working with collections of values.

```
// Redis list operations
// Add product to user's wishlist
await redis.lpush(`user:${userId}:wishlist`, productId);

// Get user's wishlist (paginated)
const wishlist = await redis.lrange(`user:${userId}:wishlist`, 0, 9);

// Redis set operations
// Add user to product's followers
await redis.sadd(`product:${productId}:followers`, userId);

// Check if user follows a product
const isFollowing = await redis.sismember(
  `product:${productId}:followers`,
  userId
);
```

```
// Get intersection of users following two products
const commonFollowers = await redis.sinter(
  `product:${productId1}:followers`,
  `product:${productId2}:followers`
);
```

4. **Sorted Set Operations**: Maintaining sorted collections for ranking and time-based access.

```
// Redis sorted set for leaderboard
// Add score for user
await redis.zadd('leaderboard', score, userId);

// Get top 10 users
const topUsers = await redis.zrevrange('leaderboard', 0, 9, 'WITHSCORES');

// Get user's rank
const userRank = await redis.zrevrank('leaderboard', userId);

// Time-series data with sorted sets
// Record temperature reading with timestamp as score
await redis.zadd(`sensor:${sensorId}:temperature`, timestamp, temperatureValue);

// Get readings between timestamps
const readings = await redis.zrangebyscore(
  `sensor:${sensorId}:temperature`,
  startTimestamp,
  endTimestamp
);
```

5. **Secondary Index Patterns**: Creating auxiliary data structures to support additional access patterns.

```
// Redis secondary indexes
// Add email lookup index
await redis.set(`email:${email}:userId`, userId);

// Look up user by email
const userId = await redis.get(`email:${email}:userId`);
const userProfile = JSON.parse(await redis.get(`user:${userId}:profile`));

// DynamoDB global secondary index
const params = {
  TableName: 'Products',
  IndexName: 'CategoryPriceIndex',
  KeyConditionExpression: 'Category = :cat AND Price < :price',
  ExpressionAttributeValues: {
    ':cat': 'Electronics',
    ':price': 500,
  },
};
const result = await dynamoDb.query(params).promise();
```

Column-Family Store Query Patterns (Cassandra)

1. **Row Key Queries**: Retrieving data by primary key.

```
-- Cassandra CQL: Get user by ID
SELECT * FROM users WHERE user_id = 'user123';

-- Get user activity for a specific date
SELECT * FROM user_activity
WHERE user_id = 'user123' AND activity_date = '2023-06-15';
```

2. **Range Queries on Clustering Columns**: Querying ordered data within a partition.

```
-- Get user activity within a date range
SELECT * FROM user_activity
WHERE user_id = 'user123'
AND activity_date >= '2023-06-01'
AND activity_date <= '2023-06-30'
ORDER BY activity_date DESC;

-- Get product reviews sorted by rating
SELECT * FROM product_reviews
WHERE product_id = 'prod456'
AND rating >= 4
ORDER BY rating DESC;
```

3. **Secondary Index Queries**: Queries on non-primary key columns (use sparingly).

```
-- Find users by email (with secondary index)
SELECT * FROM users WHERE email = 'user@example.com';

-- Find active products by category (with secondary index)
SELECT * FROM products WHERE category = 'Electronics' AND status = 'active' ALLOW
FILTERING;
```

4. **Denormalized Table Queries**: Querying denormalized data models designed for specific access patterns.

```
-- Original data model would require joins, which Cassandra doesn't support
-- Instead, we denormalize by creating a view for each query pattern

-- Find orders by customer
SELECT * FROM orders_by_customer
WHERE customer_id = 'cust123'
ORDER BY order_date DESC;

-- Find orders by product
SELECT * FROM orders_by_product
```

```
WHERE product_id = 'prod456'
ORDER BY order_date DESC;
```

5. **Time-Series Patterns**: Managing and querying time-series data.

```
-- Store metrics in time buckets to avoid hot partitions
CREATE TABLE metrics (
  sensor_id text,
  bucket_hour timestamp,
  ts timestamp,
  value double,
  PRIMARY KEY ((sensor_id, bucket_hour), ts)
);

-- Query for metrics in a specific hour
SELECT * FROM metrics
WHERE sensor_id = 'sensor001'
AND bucket_hour = '2023-06-15 14:00:00'
AND ts >= '2023-06-15 14:00:00'
AND ts < '2023-06-15 15:00:00';
```

Query Optimization Techniques

MongoDB Query Optimization

1. **Indexing Strategy**: Create the right indexes for your query patterns.

```
// Single field index
db.users.createIndex({ email: 1 }, { unique: true });

// Compound index for queries that filter and sort
db.products.createIndex({ category: 1, price: -1 });

// Text index for search
db.articles.createIndex({ title: 'text', content: 'text' });

// Partial index for increased efficiency
db.orders.createIndex(
  { orderDate: 1 },
  { partialFilterExpression: { status: 'active' } }
);
```

2. **Query Structure Optimization**: Structure queries to leverage indexes effectively.

```
// Less efficient (can't use index efficiently)
db.users.find({
  $or: [{ status: 'active' }, { email: 'user@example.com' }],
});
```

```
// More efficient (use two separate queries or restructure data model)
const activeUsers = db.users.find({ status: 'active' });
const specificUser = db.users.findOne({ email: 'user@example.com' });

// Covered queries (satisfied entirely by index)
db.users
  .find(
    { age: { $gt: 21 } },
    { _id: 0, name: 1, email: 1 } // Projection
  )
  .hint({ age: 1, name: 1, email: 1 }); // Use specific index
```

3. **Aggregation Pipeline Optimization**: Structure pipelines for maximum efficiency.

```
// Less efficient pipeline
db.orders.aggregate([
  { $match: { status: 'completed' } },
  { $unwind: '$items' },
  { $sort: { orderDate: -1 } },
  { $match: { 'items.product': 'Widget X' } },
  { $limit: 100 },
]);

// More efficient pipeline
db.orders.aggregate([
  // Place $match stages as early as possible
  { $match: { status: 'completed' } },
  // Sort before unwind if possible
  { $sort: { orderDate: -1 } },
  // Limit early to reduce documents flowing through pipeline
  { $limit: 100 },
  // Operations that increase document count come later
  { $unwind: '$items' },
  { $match: { 'items.product': 'Widget X' } },
]);
```

4. **Projection Optimization**: Return only the fields you need.

```
// Return only necessary fields
db.products.find(
  { category: 'Electronics' },
  { name: 1, price: 1, imageUrl: 1, _id: 1 }
);

// Exclude large fields
db.articles.find(
  { status: 'published' },
  { content: 0, comments: 0 } // Exclude large fields
);
```

5. **Explain Plan Analysis**: Use explain to understand query performance.

```
// Analyze query execution
db.products
  .find({ category: 'Electronics', price: { $lt: 500 } })
  .explain('executionStats');

// Analyze aggregation pipeline
db.orders
  .explain('executionStats')
  .aggregate([
    { $match: { status: 'completed' } },
    { $group: { _id: '$customerId', totalSpent: { $sum: '$total' } } },
  ]);
```

Redis Optimization Techniques

1. **Key Design Optimization**: Design keys to support your access patterns.

```
// Use delimiters in keys to enable pattern matching
// Format: object-type:id:field
await redis.set('user:1000:profile', profileJson);
await redis.set('user:1000:preferences', preferencesJson);

// Use prefixes for logical grouping
await redis.set('session:abc123', sessionData);
await redis.set('cart:user:1000', cartData);
```

2. **Data Structure Selection**: Choose the right data structure for your use case.

```
// String: simple values, serialized objects
await redis.set('user:1000', JSON.stringify(user));

// Hash: fields of an object (more efficient than storing serialized object)
await redis.hset('user:1000', 'name', 'John', 'email', 'john@example.com');

// List: ordered collection with duplicates
await redis.lpush('recent-products', productId); // Stack (newest first)
await redis.rpush('processing-queue', jobId); // Queue (FIFO)

// Set: unique items
await redis.sadd('user:1000:permissions', 'read', 'write');

// Sorted Set: ordered unique items with score
await redis.zadd('leaderboard', 1000, 'user:1000');
```

3. **Pipelining and Transactions**: Reduce network overhead with batched operations.

```
// Pipeline multiple operations
const pipeline = redis.pipeline();
pipeline.set('key1', 'value1');
pipeline.set('key2', 'value2');
pipeline.get('key3');
const results = await pipeline.exec();

// Transaction (atomic operations)
const transaction = redis.multi();
transaction.set('account:100:balance', 500);
transaction.decrby('account:200:balance', 500);
transaction.rpush('transactions', 'transfer:100:200:500');
const txResults = await transaction.exec();
```

4. **Expiration and Eviction Policies**: Manage memory usage with TTLs and appropriate eviction.

```
// Set with expiration
await redis.setex('session:abc123', 3600, sessionData); // 1 hour TTL

// Add expiration to existing key
await redis.expire('temporary-data', 86400); // 1 day

// Configure optimal eviction policy in redis.conf
// volatile-lru: Evict keys with TTL using LRU
// allkeys-lru: Evict any key using LRU
```

5. **Specialized Commands**: Use Redis-specific optimized commands.

```
// Use SCAN instead of KEYS for production environments
let cursor = '0';
do {
  const [nextCursor, keys] = await redis.scan(
    cursor,
    'MATCH',
    'user:1000:*',
    'COUNT',
    100
  );
  cursor = nextCursor;
  // Process keys batch
} while (cursor !== '0');

// Use specialized commands like BITFIELD for counters
await redis.bitfield(
  'counters',
  'INCRBY',
  'u32',
  '0',
  '1', // 32-bit counter at offset 0, increment by 1
  'INCRBY',
```

```
  'u16',
  '4',
  '1' // 16-bit counter at offset 4, increment by 1
);
```

DynamoDB Optimization Techniques

1. **Table Design for Access Patterns**: Design your primary key and indexes to match query patterns.

```
// Create table with composite key for user orders
const params = {
  TableName: 'UserOrders',
  KeySchema: [
    { AttributeName: 'UserId', KeyType: 'HASH' }, // Partition key
    { AttributeName: 'OrderDate', KeyType: 'RANGE' }, // Sort key
  ],
  AttributeDefinitions: [
    { AttributeName: 'UserId', AttributeType: 'S' },
    { AttributeName: 'OrderDate', AttributeType: 'S' },
  ],
  ProvisionedThroughput: {
    ReadCapacityUnits: 5,
    WriteCapacityUnits: 5,
  },
};
```

2. **Sparse Index Pattern**: Create indexes that only include items with the indexed attribute.

```
// Global secondary index for premium users only
const params = {
  TableName: 'Users',
  GlobalSecondaryIndexes: [
    {
      IndexName: 'PremiumUserIndex',
      KeySchema: [
        { AttributeName: 'UserType', KeyType: 'HASH' },
        { AttributeName: 'SubscriptionExpiry', KeyType: 'RANGE' },
      ],
      Projection: {
        ProjectionType: 'INCLUDE',
        NonKeyAttributes: ['Name', 'Email', 'Plan'],
      },
      ProvisionedThroughput: {
        ReadCapacityUnits: 5,
        WriteCapacityUnits: 5,
      },
    },
  ],
};

// Only premium users get the UserType attribute, making the index sparse
```

```
const premiumUser = {
  UserId: 'user123',
  Name: 'John Doe',
  Email: 'john@example.com',
  UserType: 'PREMIUM', // Only added for premium users
  SubscriptionExpiry: '2023-12-31',
};

// Query the sparse index
const queryParams = {
  TableName: 'Users',
  IndexName: 'PremiumUserIndex',
  KeyConditionExpression: 'UserType = :type AND SubscriptionExpiry > :today',
  ExpressionAttributeValues: {
    ':type': 'PREMIUM',
    ':today': '2023-06-15',
  },
};
```

3. **Write Sharding for Hot Keys**: Distribute writes across multiple partition keys.

```
// Function to get a random shard suffix
function getShardSuffix() {
  return Math.floor(Math.random() * 10); // 0-9 shards
}

// Write to a random shard
function createActivityLog(userId, activity) {
  const shardedUserId = `${userId}#${getShardSuffix()}`;

  const params = {
    TableName: 'UserActivity',
    Item: {
      ShardedUserId: shardedUserId,
      Timestamp: new Date().toISOString(),
      UserId: userId, // Keep original for filtering
      ActivityType: activity.type,
      Details: activity.details,
    },
  };

  return dynamoDb.put(params).promise();
}

// Read from all shards
async function getUserActivity(userId, limit = 20) {
  const activities = [];

  // Query each shard
  for (let shard = 0; shard < 10; shard++) {
    const shardedUserId = `${userId}#${shard}`;

    const params = {
```

```
      TableName: 'UserActivity',
      KeyConditionExpression: 'ShardedUserId = :userId',
      ExpressionAttributeValues: {
        ':userId': shardedUserId,
      },
      Limit: limit,
      ScanIndexForward: false, // Newest first
    };

    const result = await dynamoDb.query(params).promise();
    activities.push(...result.Items);
  }

  // Sort and limit combined results
  return activities
    .sort((a, b) => b.Timestamp.localeCompare(a.Timestamp))
    .slice(0, limit);
}
```

4. **Batch Operations**: Use batch reads and writes for efficiency.

```
// Batch get items
const params = {
  RequestItems: {
    Users: {
      Keys: [{ UserId: 'user1' }, { UserId: 'user2' }, { UserId: 'user3' }],
    },
    Products: {
      Keys: [{ ProductId: 'prod1' }, { ProductId: 'prod2' }],
    },
  },
};

const result = await dynamoDb.batchGet(params).promise();

// Batch write items
const batchParams = {
  RequestItems: {
    Orders: [
      { PutRequest: { Item: order1 } },
      { PutRequest: { Item: order2 } },
      { DeleteRequest: { Key: { OrderId: 'oldOrder' } } },
    ],
  },
};

await dynamoDb.batchWrite(batchParams).promise();
```

5. **Filtering Optimization**: Minimize the impact of filters in DynamoDB queries.

```
// Less efficient - scans entire result set before filtering
const lessEfficientParams = {
  TableName: 'Products',
  FilterExpression: 'Price < :maxPrice AND Category = :category',
  ExpressionAttributeValues: {
    ':maxPrice': 100,
    ':category': 'Electronics',
  },
};

// More efficient - use index for main condition
const moreEfficientParams = {
  TableName: 'Products',
  IndexName: 'CategoryPriceIndex', // Composite index on Category and Price
  KeyConditionExpression: 'Category = :category AND Price < :maxPrice',
  ExpressionAttributeValues: {
    ':category': 'Electronics',
    ':maxPrice': 100,
  },
};
```

Cassandra Optimization Techniques

1. **Data Model for Query Patterns**: Design tables specifically for each query pattern.

```
-- Instead of a single users table, create multiple tables optimized
-- for different access patterns

-- Find user by ID
CREATE TABLE users_by_id (
  user_id uuid PRIMARY KEY,
  name text,
  email text,
  created_at timestamp
);

-- Find user by email
CREATE TABLE users_by_email (
  email text PRIMARY KEY,
  user_id uuid,
  name text,
  created_at timestamp
);

-- Each query uses the table designed for that pattern
SELECT * FROM users_by_id WHERE user_id = 123e4567-e89b-12d3-a456-426614174000;
SELECT * FROM users_by_email WHERE email = 'user@example.com';
```

2. **Partition Key Design**: Distribute data evenly across the cluster.

```
-- Poor distribution (hot partition)
CREATE TABLE events (
  event_date date,
  event_id uuid,
  event_type text,
  details text,
  PRIMARY KEY (event_date, event_id)
);

-- Better distribution (add bucket for distribution)
CREATE TABLE events_bucketed (
  event_date date,
  bucket int,
  event_id uuid,
  event_type text,
  details text,
  PRIMARY KEY ((event_date, bucket), event_id)
);

-- Insert with random bucket for distribution
INSERT INTO events_bucketed (event_date, bucket, event_id, event_type, details)
VALUES ('2023-06-15', ABS(CAST(UUID() AS varint) % 10), UUID(), 'login',
'{"ip":"192.168.1.1"}');
```

3. **Materialized Views**: Pre-compute different arrangements of the same data.

```
-- Base table
CREATE TABLE videos (
  video_id uuid PRIMARY KEY,
  title text,
  upload_date timestamp,
  user_id uuid,
  category text,
  views counter
);

-- Materialized view for user's videos
CREATE MATERIALIZED VIEW videos_by_user AS
  SELECT video_id, title, upload_date, user_id, category, views
  FROM videos
  WHERE user_id IS NOT NULL AND video_id IS NOT NULL
  PRIMARY KEY (user_id, upload_date, video_id);

-- Materialized view for category browsing
CREATE MATERIALIZED VIEW videos_by_category AS
  SELECT video_id, title, upload_date, user_id, category, views
  FROM videos
  WHERE category IS NOT NULL AND video_id IS NOT NULL
  PRIMARY KEY (category, views, video_id);

-- Queries use the appropriate view
```

```
SELECT * FROM videos_by_user WHERE user_id = 123e4567-e89b-12d3-a456-426614174000;
SELECT * FROM videos_by_category WHERE category = 'Technology' LIMIT 10;
```

4. **Denormalization and Duplication**: Don't be afraid to duplicate data for query efficiency.

```
-- Original data model would require multiple queries
-- Orders table
CREATE TABLE orders (
  order_id uuid PRIMARY KEY,
  user_id uuid,
  total decimal,
  status text,
  order_date timestamp
);

-- Order items table
CREATE TABLE order_items (
  order_id uuid,
  item_id uuid,
  product_id uuid,
  quantity int,
  price decimal,
  PRIMARY KEY (order_id, item_id)
);

-- Denormalized model
-- Order details with embedded items
CREATE TABLE orders_with_items (
  order_id uuid,
  user_id uuid,
  total decimal,
  status text,
  order_date timestamp,
  items list<frozen<map<text, text>>>,
  PRIMARY KEY (order_id)
);

-- User orders with most important order data duplicated
CREATE TABLE user_orders (
  user_id uuid,
  order_date timestamp,
  order_id uuid,
  total decimal,
  status text,
  PRIMARY KEY (user_id, order_date, order_id)
);
```

5. **Consistency Level Tuning**: Adjust consistency levels based on operation importance.

```
// Critical write - use QUORUM consistency
PreparedStatement statement = session.prepare(
```

```
  "INSERT INTO user_accounts (user_id, balance) VALUES (?, ?)"
);
statement.setConsistencyLevel(ConsistencyLevel.QUORUM);
session.execute(statement.bind(userId, initialBalance));

// Less critical read - use LOCAL_ONE for performance
PreparedStatement query = session.prepare(
  "SELECT * FROM product_views WHERE product_id = ?"
);
query.setConsistencyLevel(ConsistencyLevel.LOCAL_ONE);
ResultSet results = session.execute(query.bind(productId));
```

E. Specialized Database Systems

As data requirements have evolved, specialized database systems have emerged to address specific use cases that traditional relational and NoSQL databases aren't optimized for. In this section, we'll explore graph databases, time-series databases, search engines, in-memory databases, and NewSQL databases.

5.1 Graph Databases

Graph databases excel at managing highly connected data and complex relationships. They're particularly suitable for social networks, recommendation engines, fraud detection, and knowledge graphs.

Core Concepts of Graph Databases

1. **Nodes (Vertices)**: Represent entities such as people, products, or accounts.
2. **Edges (Relationships)**: Connect nodes and represent relationships between entities.
3. **Properties**: Attributes that can be attached to both nodes and edges.
4. **Labels/Types**: Categorize nodes and edges to group similar entities.
5. **Traversal**: The process of navigating from node to node through relationships.

Comparing Graph Databases to Relational and NoSQL Databases

Feature	Relational	Document	Graph
Data Model	Tables, Rows, Columns	Documents, Collections	Nodes, Edges, Properties
Relationships	Foreign Keys	References or Embedded Documents	First-class Edges
Query Complexity for Relationships	Increases with JOINs	Requires Application Logic	Remains Constant
Schema	Rigid Schema	Schema-free/Flexible	Schema Optional
Use Cases	Structured Data	Semi-structured Data	Highly Connected Data

Neo4j: The Leading Graph Database

Neo4j is the most popular graph database, offering a property graph model and the Cypher query language.

Setting Up Neo4j

```
# Docker installation
docker run \
    --name neo4j \
    -p 7474:7474 -p 7687:7687 \
    -e NEO4J_AUTH=neo4j/password \
    -v $HOME/neo4j/data:/data \
    neo4j:latest
```

Basic Graph Modeling with Cypher

```
// Create nodes with labels and properties
CREATE (john:Person {name: 'John Doe', age: 35})
CREATE (mary:Person {name: 'Mary Smith', age: 32})
CREATE (acme:Company {name: 'ACME Corp', founded: 2010})

// Create relationships with properties
CREATE (john)-[:WORKS_AT {since: 2015, role: 'Developer'}]->(acme)
CREATE (mary)-[:WORKS_AT {since: 2018, role: 'Manager'}]->(acme)
CREATE (john)-[:FRIENDS_WITH {since: 2017}]->(mary)
```

Querying with Cypher

1. **Basic Node Queries**:

```
// Find all people
MATCH (p:Person)
RETURN p

// Find person by name
MATCH (p:Person {name: 'John Doe'})
RETURN p

// Find using conditions
MATCH (p:Person)
WHERE p.age > 30
RETURN p.name, p.age
```

2. **Relationship Queries**:

```
// Find direct relationships
MATCH (p:Person)-[r:WORKS_AT]->(c:Company)
RETURN p.name, r.role, c.name

// Find with relationship properties
```

```
MATCH (p:Person)-[r:WORKS_AT]->(c:Company)
WHERE r.since < 2017
RETURN p.name, r.since, c.name

// Find specific patterns
MATCH (p1:Person)-[:FRIENDS_WITH]->(p2:Person)-[:WORKS_AT]->(c:Company)
WHERE c.name = 'ACME Corp'
RETURN p1.name, p2.name
```

3. **Path Queries**:

```
// Find all paths between two nodes up to depth 4
MATCH path = shortestPath((a:Person {name: 'John Doe'})-[*..4]-(b:Person {name:
'Alice Johnson'}))
RETURN path

// Find all connections of a certain type within 3 steps
MATCH path = (p:Person {name: 'John Doe'})-[:FRIENDS_WITH*1..3]-(friend)
RETURN friend.name, length(path) as distance
```

4. **Aggregation Queries**:

```
// Count employees per company
MATCH (p:Person)-[:WORKS_AT]->(c:Company)
RETURN c.name, count(p) as employeeCount
ORDER BY employeeCount DESC

// Find the average age of employees at each company
MATCH (p:Person)-[:WORKS_AT]->(c:Company)
RETURN c.name, avg(p.age) as avgAge
ORDER BY avgAge DESC
```

Advanced Cypher Patterns

1. **Variable-Length Paths**:

```
// Find all friends and friends-of-friends
MATCH (p:Person {name: 'John Doe'})-[:FRIENDS_WITH*1..2]-(fof)
WHERE fof <> p  // Exclude starting point
RETURN DISTINCT fof.name,
       CASE WHEN (p)-[:FRIENDS_WITH]-(fof)
            THEN 'Direct Friend'
            ELSE 'Friend of Friend'
       END as relationship
```

2. **Pattern Comprehension**:

```
// Get person with list of companies they've worked at
MATCH (p:Person)
RETURN p.name,
       [(p)-[r:WORKS_AT]->(c) | {company: c.name, role: r.role}] as workHistory
```

3. **Recommendations**:

```
// Find friend-of-friend recommendations
MATCH (p:Person {name: 'John Doe'})-[:FRIENDS_WITH]->()-[:FRIENDS_WITH]->(fof)
WHERE NOT (p)-[:FRIENDS_WITH]-(fof) AND p <> fof
RETURN fof.name,
       count(*) as mutualFriends
ORDER BY mutualFriends DESC
```

Real-World Use Cases for Graph Databases

1. **Social Networks**:

```
// Find friends of friends who live in 'New York' that I might know
MATCH (me:Person {name: 'John Doe'})-[:FRIENDS_WITH]-(friend)-[:FRIENDS_WITH]-(foaf)
WHERE NOT (me)-[:FRIENDS_WITH]-(foaf)
  AND foaf <> me
  AND foaf.city = 'New York'
RETURN foaf.name,
       count(DISTINCT friend) as mutualFriends
ORDER BY mutualFriends DESC
```

2. **Fraud Detection**:

```
// Find suspicious patterns of accounts sharing devices and transferring money
MATCH path = (a1:Account)-[:LOGGED_IN_FROM]->(d:Device)<-[:LOGGED_IN_FROM]-
(a2:Account),
            (a1)-[t:TRANSFERRED]->(a2)
WHERE t.amount > 10000
  AND a1.owner <> a2.owner
  AND t.timestamp > datetime() - duration('P30D')  // Within last 30 days
RETURN path
```

3. **Knowledge Graphs**:

```
// Find connections between concepts across different domains
MATCH path = shortestPath(
  (c1:Concept {name: 'Machine Learning'})-[*..5]-(c2:Concept {name: 'Finance'})
)
RETURN path
```

4. **Recommendation Engines**:

```
// Product recommendations based on purchase patterns
MATCH (c:Customer {id: '123'})-[:PURCHASED]->(p1:Product)
MATCH (p1)<-[:PURCHASED]-(:Customer)-[:PURCHASED]->(reco:Product)
WHERE NOT (c)-[:PURCHASED]->(reco)
RETURN reco.name,
       count(*) as frequency
ORDER BY frequency DESC
LIMIT 5
```

Performance Considerations for Graph Databases

1. **Indexing in Neo4j**:

```
// Create index on node property
CREATE INDEX FOR (p:Person) ON (p.name)

// Create index on relationship property
CREATE INDEX FOR ()-[r:WORKS_AT]->() ON (r.since)

// Create composite index
CREATE INDEX FOR (p:Person) ON (p.city, p.age)

// Create full-text index
CREATE FULLTEXT INDEX personContent FOR (p:Person) ON EACH [p.bio, p.interests]
```

2. **Query Optimization**:

```
// Use parameters instead of literals
MATCH (p:Person)
WHERE p.name = $name
RETURN p

// Specify node labels
MATCH (p:Person) // Better than MATCH (p)
RETURN p

// Start with the most specific patterns
MATCH (p:Person {name: 'John Doe'})-[:FRIENDS_WITH]-(friend)  // Better starting
point
RETURN friend

// Use EXPLAIN and PROFILE for query analysis
EXPLAIN MATCH (p:Person)-[:FRIENDS_WITH]-(friend)
RETURN p.name, collect(friend.name)
```

3. **Data Modeling Best Practices**:

- Keep the model simple and focused on queries you need to support
- Use relationship properties instead of intermediate nodes when possible
- Consider duplicating some properties for performance (denormalization)
- Use domain-specific labels to partition the graph

Amazon Neptune: AWS's Graph Database

Amazon Neptune supports both property graph (like Neo4j) with Gremlin queries and RDF graphs with SPARQL.

Gremlin Queries in Neptune

```
// Connect to Neptune
Cluster cluster = Cluster.build()
    .addContactPoint("your-neptune-endpoint")
    .port(8182)
    .create();
GraphTraversalSource g = traversal().withRemote(
    DriverRemoteConnection.using(cluster, "g")
);

// Add vertices (nodes)
g.addV("person").property("name", "John").property("age", 35).next();
g.addV("person").property("name", "Mary").property("age", 32).next();
g.addV("company").property("name", "ACME").next();

// Add edges (relationships)
g.V().has("person", "name", "John")
    .addE("worksAt").property("since", 2015)
    .to(g.V().has("company", "name", "ACME")).next();

// Query examples
// Find all people
List<String> people = g.V().hasLabel("person")
    .values("name").toList();

// Find connections
List<Map<String, Object>> employeeData = g.V().has("company", "name", "ACME")
    .in("worksAt")
    .project("name", "age", "startYear")
    .by("name")
    .by("age")
    .by(inE("worksAt").values("since"))
    .toList();
```

5.2 Time-Series Databases

Time-series databases are optimized for tracking changes over time and handling high-volume write loads of timestamped data. They're commonly used for monitoring, IoT applications, financial trading, and sensor data analysis.

Core Concepts of Time-Series Databases

1. **Time Series**: A sequence of data points indexed in time order.
2. **Measurements/Metrics**: The quantity being recorded over time.
3. **Tags/Labels**: Metadata used to identify different series.
4. **Retention Policies**: Rules for how long to keep data at different resolutions.
5. **Downsampling**: The process of reducing data resolution over time.

InfluxDB: A Popular Time-Series Database

InfluxDB uses a SQL-like query language called InfluxQL and the newer Flux language.

Setting Up InfluxDB

```
# Docker installation
docker run -p 8086:8086 \
    -v influxdb:/var/lib/influxdb \
    influxdb:latest
```

Data Model in InfluxDB

```
measurement,tag1=value1,tag2=value2 field1=value1,field2=value2 timestamp
```

- **Measurement**: Similar to a table in SQL
- **Tags**: Indexed metadata (used in WHERE clauses)
- **Fields**: The actual values being stored (not indexed)
- **Timestamp**: When the measurement was taken

Writing Data to InfluxDB

```
# Using HTTP API
curl -i -XPOST 'http://localhost:8086/write?db=mydb' \
  --data-binary 'cpu_usage,host=server01,region=us-west cpu=0.64,memory=8.2
1465839830100000000'

# Using CLI
influx -database 'mydb' -execute "INSERT cpu_usage,host=server01,region=us-west
cpu=0.64,memory=8.2"
```

Querying with InfluxQL

1. **Basic Queries**:

```
-- Get recent CPU usage from all hosts
SELECT cpu FROM cpu_usage WHERE time > now() - 1h

-- Get average CPU usage grouped by host
SELECT mean(cpu) FROM cpu_usage
WHERE time > now() - 24h
GROUP BY host, time(1h)

-- Get the last value for each host
SELECT last(cpu) FROM cpu_usage GROUP BY host
```

2. **Continuous Queries (Automatic Downsampling)**:

```
-- Create a continuous query for downsampling
CREATE CONTINUOUS QUERY "cq_30m" ON "mydb"
BEGIN
  SELECT mean(cpu) AS cpu, mean(memory) AS memory
  INTO "cpu_usage_30m"
  FROM "cpu_usage"
  GROUP BY time(30m), host
END
```

3. **Advanced Analysis**:

```
-- Find rate of change
SELECT derivative(cpu, 1m) FROM cpu_usage
WHERE host='server01' AND time > now() - 6h

-- Find servers with CPU usage above 90% for more than 5 minutes
SELECT count(cpu) FROM cpu_usage
WHERE cpu > 90 AND time > now() - 1h
GROUP BY host, time(5m)
HAVING count(cpu) > 5
```

Querying with Flux (Modern InfluxDB)

```
// Basic query with Flux
from(bucket: "mydb/autogen")
  |> range(start: -1h)
  |> filter(fn: (r) => r._measurement == "cpu_usage")
  |> filter(fn: (r) => r._field == "cpu")
  |> mean()
  |> group(columns: ["host"])

// Find anomalies
from(bucket: "mydb/autogen")
  |> range(start: -1d)
```

```
  |> filter(fn: (r) => r._measurement == "cpu_usage")
  |> filter(fn: (r) => r._field == "cpu")
  |> group(columns: ["host"])
  |> difference()
  |> filter(fn: (r) => abs(r._value) > 0.3)  // Sudden changes > 30%
```

TimescaleDB: PostgreSQL for Time-Series

TimescaleDB is a PostgreSQL extension that optimizes it for time-series workloads while maintaining full SQL compatibility.

Setting Up TimescaleDB

```
# Docker installation
docker run -d --name timescaledb -p 5432:5432 \
  -e POSTGRES_PASSWORD=password \
  timescale/timescaledb:latest-pg14
```

Creating Time-Series Tables

```
-- Connect to PostgreSQL
\c postgres

-- Create extension
CREATE EXTENSION IF NOT EXISTS timescaledb;

-- Create regular table
CREATE TABLE cpu_usage (
  time TIMESTAMPTZ NOT NULL,
  host TEXT NOT NULL,
  region TEXT NOT NULL,
  cpu DOUBLE PRECISION NULL,
  memory DOUBLE PRECISION NULL
);

-- Convert to hypertable (time-partitioned)
SELECT create_hypertable('cpu_usage', 'time');

-- Create indexes
CREATE INDEX idx_cpu_usage_host ON cpu_usage(host, time DESC);
```

Querying with SQL (TimescaleDB)

```
-- Basic query
SELECT time, cpu
FROM cpu_usage
WHERE host = 'server01'
```

```
  AND time > NOW() - INTERVAL '1 hour'
ORDER BY time DESC;

-- Aggregate data
SELECT
  time_bucket('15 minutes', time) AS bucket,
  host,
  AVG(cpu) AS avg_cpu
FROM cpu_usage
WHERE time > NOW() - INTERVAL '1 day'
GROUP BY bucket, host
ORDER BY bucket DESC, host;

-- Find the 95th percentile
SELECT
  host,
  percentile_cont(0.95) WITHIN GROUP (ORDER BY cpu) AS p95_cpu
FROM cpu_usage
WHERE time > NOW() - INTERVAL '1 day'
GROUP BY host;
```

TimescaleDB-Specific Functions

```
-- Time-weighted average
SELECT time_weight('linear', time, cpu)
FROM cpu_usage
WHERE host = 'server01'
  AND time BETWEEN '2023-06-01' AND '2023-06-02';

-- First/last value per period
SELECT time_bucket('1 hour', time) AS hour,
       first(cpu, time) AS first_cpu,
       last(cpu, time) AS last_cpu
FROM cpu_usage
WHERE time > NOW() - INTERVAL '1 day'
GROUP BY hour
ORDER BY hour;

-- Gaps and islands detection
SELECT host, time_bucket('1 hour', time) AS hour,
       has_gaps(time, INTERVAL '5 minutes') AS has_data_gaps
FROM cpu_usage
GROUP BY host, hour;
```

Real-World Use Cases for Time-Series Databases

1. **Infrastructure Monitoring**:

```
-- Find hosts with potential memory leaks
SELECT
```

```
  host,
  first(memory, time) AS start_memory,
  last(memory, time) AS end_memory,
  (last(memory, time) - first(memory, time)) AS memory_growth
FROM cpu_usage
WHERE time > NOW() - INTERVAL '7 days'
GROUP BY host
HAVING (last(memory, time) - first(memory, time)) > 1.0
ORDER BY memory_growth DESC;
```

2. **IoT Device Tracking**:

```
-- Track temperature anomalies from IoT sensors
SELECT
  sensor_id,
  time_bucket('1 hour', time) AS hour,
  AVG(temperature) AS avg_temp,
  MAX(temperature) AS max_temp,
  MIN(temperature) AS min_temp,
  MAX(temperature) - MIN(temperature) AS temp_range
FROM sensor_readings
WHERE time > NOW() - INTERVAL '1 day'
GROUP BY sensor_id, hour
HAVING MAX(temperature) - MIN(temperature) > 10
ORDER BY temp_range DESC;
```

3. **Financial Market Data**:

```
-- Calculate moving averages for stock prices
WITH price_data AS (
  SELECT
    symbol,
    time_bucket('1 day', time) AS day,
    last(price, time) AS closing_price
  FROM stock_prices
  WHERE time > NOW() - INTERVAL '90 days'
  GROUP BY symbol, day
)
SELECT
  symbol,
  day,
  closing_price,
  AVG(closing_price) OVER (
    PARTITION BY symbol
    ORDER BY day
    ROWS BETWEEN 19 PRECEDING AND CURRENT ROW
  ) AS moving_avg_20d
FROM price_data
ORDER BY symbol, day;
```

Performance Considerations for Time-Series Databases

1. **Schema Design**:

- Use tags/labels for data you'll query by
- Keep cardinality of tags in mind (avoid high-cardinality tags)
- Choose appropriate field types to minimize storage

2. **Retention and Downsampling**:

```
-- InfluxDB retention policy
CREATE RETENTION POLICY "one_day" ON "mydb" DURATION 1d REPLICATION 1

-- TimescaleDB automated data retenion
SELECT add_retention_policy('cpu_usage', INTERVAL '30 days');

-- TimescaleDB continuous aggregates (similar to continuous queries)
CREATE MATERIALIZED VIEW cpu_hourly
WITH (timescaledb.continuous) AS
SELECT
  time_bucket('1 hour', time) AS bucket,
  host,
  AVG(cpu) AS avg_cpu,
  MAX(cpu) AS max_cpu
FROM cpu_usage
GROUP BY bucket, host;
```

3. **Query Optimization**:

- Filter by time first, then by tags
- Use time-bucket functions to pre-aggregate data
- Leverage specialized time-series functions
- Consider continuous aggregations for common queries

5.3 Search Engines (Elasticsearch)

Search engines are specialized databases optimized for full-text search, faceted search, and analyzing unstructured data. Elasticsearch is the most popular search engine database, often used in conjunction with other databases to provide search functionality.

Core Concepts of Elasticsearch

1. **Index**: Similar to a database in traditional DBMS.
2. **Document**: JSON objects stored within an index.
3. **Mapping**: Schema definition for documents.
4. **Shards and Replicas**: How data is distributed and replicated.
5. **Inverted Index**: Data structure that maps content to locations.

Setting Up Elasticsearch

```
# Docker installation
docker run -d --name elasticsearch -p 9200:9200 -p 9300:9300 \
  -e "discovery.type=single-node" \
  elasticsearch:7.15.0
```

Basic Operations in Elasticsearch

1. **Creating an Index**:

```
# Create index with settings and mappings
curl -X PUT "localhost:9200/products" -H "Content-Type: application/json" -d'
{
  "settings": {
    "number_of_shards": 1,
    "number_of_replicas": 0
  },
  "mappings": {
    "properties": {
      "name": { "type": "text" },
      "description": { "type": "text" },
      "category": { "type": "keyword" },
      "price": { "type": "float" },
      "created": { "type": "date" },
      "tags": { "type": "keyword" },
      "in_stock": { "type": "boolean" },
      "location": { "type": "geo_point" }
    }
  }
}'
```

2. **Indexing Documents**:

```
# Index a document
curl -X POST "localhost:9200/products/_doc" -H "Content-Type: application/json" -d'
{
  "name": "Smartphone XYZ",
  "description": "High-performance smartphone with advanced camera features and long
battery life",
  "category": "Electronics",
  "price": 599.99,
  "created": "2023-01-15",
  "tags": ["smartphone", "camera", "5G"],
  "in_stock": true,
  "location": {
    "lat": 40.7128,
    "lon": -74.0060
  }
}'

# Index with specific ID
```

```
curl -X PUT "localhost:9200/products/_doc/1001" -H "Content-Type: application/json"
-d'
{
  "name": "Laptop ABC",
  "description": "Lightweight laptop with powerful processor and high-resolution
display",
  "category": "Electronics",
  "price": 1299.99,
  "created": "2023-02-20",
  "tags": ["laptop", "ultrabook", "16GB RAM"],
  "in_stock": true,
  "location": {
    "lat": 37.7749,
    "lon": -122.4194
  }
}'
```

3. **Searching Documents**:

```
# Simple search
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "match": {
      "description": "powerful processor"
    }
  }
}'

# Multi-field search
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "multi_match": {
      "query": "high performance",
      "fields": ["name", "description"],
      "type": "best_fields"
    }
  }
}'

# Boolean query
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "bool": {
      "must": [
        { "match": { "category": "Electronics" } }
      ],
      "should": [
```

```
        { "match": { "description": "powerful" } },
        { "match": { "description": "lightweight" } }
      ],
      "must_not": [
        { "range": { "price": { "gt": 1500 } } }
      ],
      "filter": [
        { "term": { "in_stock": true } }
      ]
    }
  }
}'
```

Advanced Search Features

1. **Aggregations**:

```
# Aggregation for price statistics and category counts
curl -X GET "localhost:9200/products/_search?size=0" -H "Content-Type:
application/json" -d'
{
  "aggs": {
    "price_stats": {
      "stats": { "field": "price" }
    },
    "categories": {
      "terms": { "field": "category" },
      "aggs": {
        "avg_price": {
          "avg": { "field": "price" }
        }
      }
    }
  }
}'

# Histogram aggregation
curl -X GET "localhost:9200/products/_search?size=0" -H "Content-Type:
application/json" -d'
{
  "aggs": {
    "price_histogram": {
      "histogram": {
        "field": "price",
        "interval": 100
      }
    }
  }
}'
```

2. **Geospatial Queries**:

```
# Find products within distance of a point
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "geo_distance": {
      "distance": "50km",
      "location": {
        "lat": 40.7128,
        "lon": -74.0060
      }
    }
  }
}'

# Find products within a bounding box
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "geo_bounding_box": {
      "location": {
        "top_left": {
          "lat": 40.73,
          "lon": -74.1
        },
        "bottom_right": {
          "lat": 40.01,
          "lon": -73.9
        }
      }
    }
  }
}'
```

3. **Full-Text Search Features**:

```
# Fuzzy matching
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "fuzzy": {
      "name": {
        "value": "phne",
        "fuzziness": "AUTO"
      }
    }
  }
}'

# Highlighting
```

```
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "match": {
      "description": "powerful processor"
    }
  },
  "highlight": {
    "fields": {
      "description": {}
    }
  }
}'

# Phrase matching with slop (word proximity)
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "query": {
    "match_phrase": {
      "description": {
        "query": "high resolution",
        "slop": 1
      }
    }
  }
}'
```

Elasticsearch from Node.js

```
// Using the official Elasticsearch client
const { Client } = require('@elastic/elasticsearch');
const client = new Client({ node: 'http://localhost:9200' });

// Index a document
async function indexDocument() {
  await client.index({
    index: 'products',
    body: {
      name: 'Wireless Headphones',
      description:
        'Noise-cancelling wireless headphones with 20-hour battery life',
      category: 'Electronics',
      price: 199.99,
      created: new Date(),
      tags: ['headphones', 'wireless', 'noise-cancelling'],
      in_stock: true,
    },
  });
}

// Search with query
```

```
async function searchProducts(keyword) {
  const result = await client.search({
    index: 'products',
    body: {
      query: {
        multi_match: {
          query: keyword,
          fields: ['name^2', 'description'],
          fuzziness: 'AUTO',
        },
      },
      highlight: {
        fields: {
          name: {},
          description: {},
        },
      },
      aggs: {
        categories: {
          terms: { field: 'category' },
        },
        price_range: {
          range: {
            field: 'price',
            ranges: [{ to: 100 }, { from: 100, to: 500 }, { from: 500 }],
          },
        },
      },
    },
  });

  return {
    hits: result.body.hits.hits,
    aggregations: result.body.aggregations,
  };
}
```

Real-World Use Cases for Elasticsearch

1. **Product Search for E-commerce**:

```
async function searchProducts(options) {
  const {
    keyword,
    category,
    minPrice,
    maxPrice,
    sort,
    page = 1,
    pageSize = 20,
  } = options;

  // Build query
```

```
    const query = {
      bool: {
        must: [],
      },
    };

    // Add keyword search if provided
    if (keyword) {
      query.bool.must.push({
        multi_match: {
          query: keyword,
          fields: ['name^3', 'description', 'tags^2'],
          type: 'best_fields',
          fuzziness: 'AUTO',
        },
      });
    }

    // Add filters
    if (category) {
      query.bool.filter = query.bool.filter || [];
      query.bool.filter.push({ term: { category } });
    }

    // Price range filter
    if (minPrice !== undefined || maxPrice !== undefined) {
      const range = { price: {} };
      if (minPrice !== undefined) range.price.gte = minPrice;
      if (maxPrice !== undefined) range.price.lte = maxPrice;

      query.bool.filter = query.bool.filter || [];
      query.bool.filter.push({ range });
    }

    // Only in-stock products
    query.bool.filter = query.bool.filter || [];
    query.bool.filter.push({ term: { in_stock: true } });

    // Determine sort order
    let sortField;
    switch (sort) {
      case 'price_asc':
        sortField = [{ price: 'asc' }];
        break;
      case 'price_desc':
        sortField = [{ price: 'desc' }];
        break;
      case 'newest':
        sortField = [{ created: 'desc' }];
        break;
      default:
        sortField = [{ _score: 'desc' }];
    }

    // Execute search
```

```
    const result = await client.search({
      index: 'products',
      body: {
        query,
        sort: sortField,
        from: (page - 1) * pageSize,
        size: pageSize,
        highlight: {
          fields: {
            name: {},
            description: {},
          },
        },
        aggs: {
          categories: {
            terms: { field: 'category' },
          },
          price_ranges: {
            range: {
              field: 'price',
              ranges: [
                { to: 100, key: 'Under $100' },
                { from: 100, to: 200, key: '$100 to $200' },
                { from: 200, to: 500, key: '$200 to $500' },
                { from: 500, key: '$500 and above' },
              ],
            },
          },
          tags: {
            terms: { field: 'tags', size: 10 },
          },
        },
      },
    });

    // Format results
    return {
      products: result.body.hits.hits.map((hit) => ({
        id: hit._id,
        ...hit._source,
        score: hit._score,
        highlights: hit.highlight,
      })),
      total: result.body.hits.total.value,
      facets: {
        categories: result.body.aggregations.categories.buckets,
        priceRanges: result.body.aggregations.price_ranges.buckets,
        tags: result.body.aggregations.tags.buckets,
      },
      page,
      pageSize,
      totalPages: Math.ceil(result.body.hits.total.value / pageSize),
    };
  }
```

2. **Log Analysis and Monitoring**:

```
async function analyzeLogs(options) {
  const {
    startTime = 'now-24h',
    endTime = 'now',
    level,
    service,
    keyword,
  } = options;

  // Base query
  const query = {
    bool: {
      must: [{ range: { timestamp: { gte: startTime, lte: endTime } } }],
    },
  };

  // Add filters
  if (level) {
    query.bool.must.push({ term: { level } });
  }

  if (service) {
    query.bool.must.push({ term: { service } });
  }

  if (keyword) {
    query.bool.must.push({ match: { message: keyword } });
  }

  // Execute search
  const result = await client.search({
    index: 'logs-*',
    body: {
      query,
      size: 0, // We're just interested in aggregations
      aggs: {
        error_count_over_time: {
          date_histogram: {
            field: 'timestamp',
            calendar_interval: '1h',
          },
          aggs: {
            by_level: {
              terms: { field: 'level' },
            },
          },
        },
        services: {
          terms: { field: 'service' },
          aggs: {
            error_count: {
              filter: { term: { level: 'error' } },
```

```
            },
            top_errors: {
              top_hits: {
                size: 5,
                sort: [{ timestamp: 'desc' }],
                _source: ['timestamp', 'message', 'stack_trace'],
              },
            },
          },
        },
        error_types: {
          terms: { field: 'error_type' },
          aggs: {
            trend: {
              date_histogram: {
                field: 'timestamp',
                calendar_interval: '1h',
              },
            },
          },
        },
      },
    },
  });

  return {
    timeSeries: result.body.aggregations.error_count_over_time.buckets,
    serviceBreakdown: result.body.aggregations.services.buckets,
    errorTypes: result.body.aggregations.error_types.buckets,
  };
}
```

3. **Content Search with Recommendations**:

```
async function searchContentWithRecommendations(options) {
  const { keyword, userId, contentType, page = 1, pageSize = 10 } = options;

  // Search query
  const searchQuery = {
    multi_match: {
      query: keyword,
      fields: ['title^2', 'content', 'tags^1.5'],
      type: 'best_fields',
    },
  };

  // Get user preferences for personalization
  const userPreferences = await getUserPreferences(userId);

  // Personalized search with function score
  const query = {
    function_score: {
      query: searchQuery,
```

```
        functions: [
          // Boost recently published content
          {
            gauss: {
              published_date: {
                origin: 'now',
                scale: '30d',
                decay: 0.5,
              },
            },
            weight: 1,
          },
          // Boost content matching user preferences
          {
            filter: { terms: { category: userPreferences.categories } },
            weight: 1.2,
          },
          // Boost popular content
          {
            field_value_factor: {
              field: 'popularity_score',
              factor: 0.5,
              modifier: 'log1p',
            },
          },
        ],
        score_mode: 'sum',
        boost_mode: 'multiply',
      },
    };

    // Add content type filter if specified
    if (contentType) {
      query.function_score.query = {
        bool: {
          must: [searchQuery],
          filter: [{ term: { type: contentType } }],
        },
      };
    }

    // Execute search
    const result = await client.search({
      index: 'content',
      body: {
        query,
        from: (page - 1) * pageSize,
        size: pageSize,
        highlight: {
          fields: {
            title: {},
            content: { fragment_size: 150, number_of_fragments: 3 },
          },
        },
        suggest: {
```

```
          text: keyword,
          term_suggestion: {
            term: {
              field: 'title',
              suggest_mode: 'always',
            },
          },
        },
      },
    });

    // Get the search results
    const searchResults = result.body.hits.hits.map((hit) => ({
      id: hit._id,
      ...hit._source,
      score: hit._score,
      highlights: hit.highlight,
    }));

    // Get content recommendations based on user history
    const recommendations = await getContentRecommendations(
      userId,
      searchResults.map((r) => r.id)
    );

    return {
      results: searchResults,
      total: result.body.hits.total.value,
      suggestions: result.body.suggest?.term_suggestion?.[0]?.options || [],
      recommendations,
    };
  }

  // Helper function to get recommendations (would be another Elasticsearch query)
  async function getContentRecommendations(userId, excludeIds) {
    const userHistory = await getUserViewHistory(userId);

    // Use "more like this" query based on user history
    const result = await client.search({
      index: 'content',
      body: {
        query: {
          bool: {
            must: {
              more_like_this: {
                fields: ['title', 'content', 'tags', 'category'],
                like: userHistory.map((item) => ({
                  _index: 'content',
                  _id: item.contentId,
                })),
                min_term_freq: 1,
                min_doc_freq: 1,
                max_query_terms: 25,
              },
            },
```

```
            must_not: {
              ids: {
                values: excludeIds,
              },
            },
          },
        },
        size: 5,
      },
    });

    return result.body.hits.hits.map((hit) => ({
      id: hit._id,
      ...hit._source,
      score: hit._score,
    }));
  }
```

Performance Considerations for Elasticsearch

1. **Mapping Optimization**:

```
# Optimize mappings for different field types
curl -X PUT "localhost:9200/optimized_products" -H "Content-Type: application/json"
-d'
{
  "mappings": {
    "properties": {
      "id": { "type": "keyword" },
      "name": {
        "type": "text",
        "fields": {
          "keyword": { "type": "keyword" },
          "completion": { "type": "completion" }
        },
        "analyzer": "english"
      },
      "description": {
        "type": "text",
        "analyzer": "english"
      },
      "sku": { "type": "keyword" },
      "price": { "type": "float" },
      "category": { "type": "keyword" },
      "tags": { "type": "keyword" },
      "created": { "type": "date" },
      "modified": { "type": "date" },
      "in_stock": { "type": "boolean" },
      "stock_count": { "type": "integer" },
      "attributes": {
        "type": "nested",
        "properties": {
          "name": { "type": "keyword" },
```

```
          "value": { "type": "keyword" }
        }
      }
    }
  }
}'
```

2. **Index Settings Optimization**:

```
# Configure index settings for better performance
curl -X PUT "localhost:9200/performance_tuned_index" -H "Content-Type:
application/json" -d'
{
  "settings": {
    "number_of_shards": 3,
    "number_of_replicas": 1,
    "refresh_interval": "30s",
    "index": {
      "codec": "best_compression",
      "search.slowlog.threshold.query.warn": "1s",
      "search.slowlog.threshold.fetch.warn": "800ms"
    },
    "analysis": {
      "analyzer": {
        "custom_analyzer": {
          "type": "custom",
          "tokenizer": "standard",
          "filter": ["lowercase", "english_stop", "snowball"]
        }
      },
      "filter": {
        "english_stop": {
          "type": "stop",
          "stopwords": "_english_"
        },
        "snowball": {
          "type": "snowball",
          "language": "English"
        }
      }
    }
  }
}'
```

3. **Query Optimization**:

- Use filters instead of queries when exact matching
- Consider using `_source` filtering to return only needed fields
- Use pagination properly to avoid deep paging issues
- Use aggregation bucketing wisely to avoid memory issues

```
# Optimized query using filter context
curl -X GET "localhost:9200/products/_search" -H "Content-Type: application/json" -
d'
{
  "size": 20,
  "_source": ["name", "price", "category", "tags"],
  "query": {
    "bool": {
      "must": {
        "match": {
          "description": {
            "query": "wireless headphones",
            "operator": "and",
            "fuzziness": "AUTO"
          }
        }
      },
      "filter": [
        { "term": { "category": "Electronics" } },
        { "range": { "price": { "lte": 300 } } },
        { "term": { "in_stock": true } }
      ]
    }
  },
  "sort": [
    { "_score": "desc" },
    { "price": "asc" }
  ]
}'
```

4. **Bulk Operations**:

```
// Bulk indexing for better performance
async function bulkIndexProducts(products) {
  const operations = products.flatMap((product) => [
    { index: { _index: 'products', _id: product.id } },
    product,
  ]);

  const bulkResponse = await client.bulk({ body: operations });

  return {
    took: bulkResponse.body.took,
    errors: bulkResponse.body.errors,
    items: bulkResponse.body.items,
  };
}
```

5. **Monitoring and Management**:

- Use tools like Elasticsearch Curator for index management

- Monitor cluster health and performance
- Implement appropriate backup strategies
- Consider shard allocation strategies for large clusters

5.4 In-Memory Databases

In-memory databases store data primarily in RAM rather than on disk, providing extremely fast data access at the cost of persistence. They're used for caching, session storage, real-time analytics, and other use cases requiring very low latency.

Core Concepts of In-Memory Databases

1. **RAM Storage**: Data is stored primarily in memory rather than on disk.
2. **Persistence Options**: Methods to provide durability despite in-memory storage.
3. **Data Structures**: Specialized data structures optimized for in-memory access.
4. **Eviction Policies**: Rules for removing data when memory limits are reached.
5. **Distributed Architecture**: Scaling in-memory databases across machines.

Redis: Popular In-Memory Database

Redis is the most widely used in-memory database, offering rich data structures and versatile capabilities.

Setting Up Redis

```
# Docker installation
docker run -p 6379:6379 --name redis-server -d redis
```

Core Data Structures

1. **Strings**:

```
# Basic key-value operations
> SET user:1000:name "John Doe"
OK
> GET user:1000:name
"John Doe"

# Numeric operations
> SET counter 1
OK
> INCR counter
(integer) 2
> INCRBY counter 5
(integer) 7

# String ranges
> SET message "Hello, world!"
OK
```

```
> GETRANGE message 0 4
"Hello"
```

2. **Lists**:

```
# Queue operations
> LPUSH tasks "send-email"
(integer) 1
> LPUSH tasks "process-payment"
(integer) 2
> RPUSH tasks "generate-report"
(integer) 3
> LRANGE tasks 0 -1
1) "process-payment"
2) "send-email"
3) "generate-report"

# Pop operations (for work queues)
> RPOP tasks
"generate-report"
> BLPOP tasks 10  # Blocking pop with timeout
1) "tasks"
2) "process-payment"
```

3. **Sets**:

```
# Add and remove members
> SADD tags:product:1001 "electronics" "laptop" "sale"
(integer) 3
> SREM tags:product:1001 "sale"
(integer) 1
> SMEMBERS tags:product:1001
1) "laptop"
2) "electronics"

# Set operations
> SADD user:1000:permissions "read" "write"
(integer) 2
> SADD user:1001:permissions "read" "admin"
(integer) 2
> SINTER user:1000:permissions user:1001:permissions
1) "read"
> SUNION user:1000:permissions user:1001:permissions
1) "read"
2) "write"
3) "admin"
```

4. **Sorted Sets**:

```
# Add members with scores
> ZADD leaderboard 1000 "user:1000"
(integer) 1
> ZADD leaderboard 2500 "user:1001"
(integer) 1
> ZADD leaderboard 1800 "user:1002"
(integer) 1

# Get range by score
> ZRANGE leaderboard 0 -1 WITHSCORES
1) "user:1000"
2) "1000"
3) "user:1002"
4) "1800"
5) "user:1001"
6) "2500"

# Get rank
> ZREVRANK leaderboard "user:1001"
(integer) 0  # First place (zero-based)
```

5. **Hashes**:

```
# Set multiple fields
> HSET user:1000 name "John Doe" email "john@example.com" age 35
(integer) 3

# Get specific fields or all fields
> HGET user:1000 name
"John Doe"
> HGETALL user:1000
1) "name"
2) "John Doe"
3) "email"
4) "john@example.com"
5) "age"
6) "35"

# Increment numeric fields
> HINCRBY user:1000 age 1
(integer) 36
```

6. **HyperLogLog**:

```
# Approximate counting of unique items
> PFADD visitors:2023-06-15 "user:1000" "user:1001"
(integer) 1
> PFADD visitors:2023-06-15 "user:1002" "user:1000"
(integer) 1
> PFCOUNT visitors:2023-06-15
```

```
(integer) 3  # Only counts unique IDs

# Combine HyperLogLogs
> PFADD visitors:2023-06-16 "user:1001" "user:1003"
(integer) 1
> PFMERGE visitors:june visitors:2023-06-15 visitors:2023-06-16
OK
> PFCOUNT visitors:june
(integer) 4  # Unique visitors across both days
```

Redis Persistence Options

```
# RDB persistence configuration in redis.conf
save 900 1      # Save if at least 1 key changes in 900 seconds
save 300 10     # Save if at least 10 keys change in 300 seconds
save 60 10000   # Save if at least 10000 keys change in 60 seconds

# AOF persistence configuration in redis.conf
appendonly yes
appendfsync everysec  # Options: always, everysec, no
```

Redis Pub/Sub Messaging

```
# In subscriber client
> SUBSCRIBE notifications
Reading messages... (press Ctrl-C to quit)
1) "subscribe"
2) "notifications"
3) (integer) 1

# In publisher client
> PUBLISH notifications "Hello, subscribers!"
(integer) 1  # Number of clients that received the message

# Back in subscriber client, the message appears:
1) "message"
2) "notifications"
3) "Hello, subscribers!"
```

Redis Transactions

```
# Atomic transaction with MULTI/EXEC
> MULTI
OK
> SET balance:1000 500
QUEUED
> DECRBY balance:1000 100
```

```
QUEUED
> INCRBY balance:1001 100
QUEUED
> EXEC
1) OK
2) (integer) 400
3) (integer) 100

# Transaction with conditions
> WATCH balance:1000  # Watch for changes
OK
> MULTI
OK
> DECRBY balance:1000 100
QUEUED
> EXEC
(nil)  # Returns nil if balance:1000 changed between WATCH and EXEC
```

Redis from Node.js

```
const Redis = require('ioredis');
const redis = new Redis();

// Basic operations
async function basicOperations() {
  // Strings
  await redis.set('user:1000:name', 'John Doe');
  const name = await redis.get('user:1000:name');

  // Increment counter
  await redis.incr('visitor_count');

  // Lists
  await redis.lpush('tasks', 'task1', 'task2');
  const nextTask = await redis.rpop('tasks');

  // Hashes
  await redis.hset('user:1000', 'name', 'John', 'email', 'john@example.com');
  const email = await redis.hget('user:1000', 'email');

  // Sets
  await redis.sadd('user:1000:roles', 'admin', 'editor');
  const isAdmin = await redis.sismember('user:1000:roles', 'admin');
}

// Implement a simple rate limiter
async function isRateLimited(userId, maxRequests = 10, windowSeconds = 60) {
  const key = `ratelimit:${userId}`;

  // Increment counter
  const count = await redis.incr(key);

  // Set expiry on first request
```

```
  if (count === 1) {
    await redis.expire(key, windowSeconds);
  }

  // Check if rate limited
  return count > maxRequests;
}

// Implement a distributed lock
async function acquireLock(lockName, timeout = 10) {
  // Set lock with NX (only if it doesn't exist) and expiry
  const result = await redis.set(
    `lock:${lockName}`,
    process.pid.toString(),
    'EX',
    timeout,
    'NX'
  );

  return result === 'OK';
}

async function releaseLock(lockName) {
  // Only release if we own the lock
  const script = `
    if redis.call("get", KEYS[1]) == ARGV[1] then
      return redis.call("del", KEYS[1])
    else
      return 0
    end
  `;

  return redis.eval(script, 1, `lock:${lockName}`, process.pid.toString());
}
```

Memcached: Focused Cache Solution

Memcached is a simpler in-memory database focused purely on caching.

Setting Up Memcached

```
# Docker installation
docker run -p 11211:11211 --name memcached -d memcached
```

Basic Memcached Operations

```
# Using telnet to interact
$ telnet localhost 11211

# Set a key with expiration (in seconds)
```

```
set user:1000 0 3600 10
John Doe
STORED

# Get a key
get user:1000
VALUE user:1000 0 10
John Doe
END

# Delete a key
delete user:1000
DELETED

# Increment a counter
set counter 0 0 1
0
STORED
incr counter 1
1
```

Memcached from Node.js

```
const Memcached = require('memcached');
const memcached = new Memcached('localhost:11211');

// Store a value (with 1 hour TTL)
memcached.set(
  'user:1000',
  { name: 'John Doe', email: 'john@example.com' },
  3600,
  (err) => {
    if (err) console.error(err);
  }
);

// Retrieve a value
memcached.get('user:1000', (err, data) => {
  if (err) console.error(err);
  console.log(data);
});

// Delete a key
memcached.del('user:1000', (err) => {
  if (err) console.error(err);
});

// Cache function results
function getUserData(userId, callback) {
  const cacheKey = `user_data:${userId}`;

  memcached.get(cacheKey, (err, data) => {
    if (err) return callback(err);
```

```
    if (data) {
      // Cache hit
      return callback(null, data);
    }

    // Cache miss - fetch from database
    database.getUser(userId, (err, userData) => {
      if (err) return callback(err);

      // Store in cache (30 minute TTL)
      memcached.set(cacheKey, userData, 1800, (err) => {
        if (err) console.error('Cache store error:', err);
      });

      callback(null, userData);
    });
  });
}
```

Real-World Use Cases for In-Memory Databases

1. **Caching Layer**:

```
// Implementing a caching layer with Redis
class CacheLayer {
  constructor(redisClient, defaultTTL = 3600) {
    this.redis = redisClient;
    this.defaultTTL = defaultTTL;
  }

  async get(key) {
    return this.redis.get(`cache:${key}`);
  }

  async set(key, value, ttl = this.defaultTTL) {
    return this.redis.set(
      `cache:${key}`,
      typeof value === 'string' ? value : JSON.stringify(value),
      'EX',
      ttl
    );
  }

  async delete(key) {
    return this.redis.del(`cache:${key}`);
  }

  async mget(keys) {
    const cacheKeys = keys.map((key) => `cache:${key}`);
    const values = await this.redis.mget(cacheKeys);

    // Convert results to object
```

```
    const result = {};
    keys.forEach((key, index) => {
      result[key] = values[index];
    });

    return result;
  }

  // Cache function results
  async cached(fn, cacheKey, ttl = this.defaultTTL) {
    // Try to get from cache first
    const cached = await this.get(cacheKey);
    if (cached) {
      return JSON.parse(cached);
    }

    // Execute function
    const result = await fn();

    // Store in cache
    await this.set(cacheKey, result, ttl);

    return result;
  }
}
```

2. **Session Store**:

```
// Express session with Redis
const express = require('express');
const session = require('express-session');
const RedisStore = require('connect-redis').default;
const Redis = require('ioredis');

const redisClient = new Redis();
const app = express();

app.use(
  session({
    store: new RedisStore({ client: redisClient }),
    secret: 'your-secret-key',
    resave: false,
    saveUninitialized: false,
    cookie: { secure: process.env.NODE_ENV === 'production', maxAge: 86400000 }, //
1 day
  })
);

app.get('/login', (req, res) => {
  // Set user in session
  req.session.user = { id: 1000, name: 'John Doe' };
  res.send('Logged in');
});
```

```
app.get('/profile', (req, res) => {
  // Access user from session
  if (!req.session.user) {
    return res.redirect('/login');
  }

  res.send(`Hello, ${req.session.user.name}`);
});

app.get('/logout', (req, res) => {
  // Destroy session
  req.session.destroy();
  res.send('Logged out');
});
```

3. **Real-Time Leaderboard**:

```
// Implementing a leaderboard with Redis sorted sets
class Leaderboard {
  constructor(redisClient, leaderboardName, ttl = null) {
    this.redis = redisClient;
    this.key = `leaderboard:${leaderboardName}`;
    this.ttl = ttl;
  }

  async addScore(userId, score) {
    await this.redis.zadd(this.key, score, userId);

    // Set expiry if ttl provided
    if (this.ttl && (await this.redis.ttl(this.key)) < 0) {
      await this.redis.expire(this.key, this.ttl);
    }
  }

  async incrementScore(userId, increment) {
    return this.redis.zincrby(this.key, increment, userId);
  }

  async getRank(userId) {
    // Use zrevrank for descending rank (higher score = better rank)
    const rank = await this.redis.zrevrank(this.key, userId);
    return rank !== null ? rank + 1 : null; // 1-based rank
  }

  async getScore(userId) {
    return this.redis.zscore(this.key, userId);
  }

  async getTopScores(count = 10) {
    // Get top users with scores
    const results = await this.redis.zrevrange(
      this.key,
```

```
      0,
      count - 1,
      'WITHSCORES'
    );

    // Format results as array of objects
    const leaderboard = [];
    for (let i = 0; i < results.length; i += 2) {
      leaderboard.push({
        userId: results[i],
        score: parseFloat(results[i + 1]),
        rank: i / 2 + 1,
      });
    }

    return leaderboard;
  }

  async getUsersAroundRank(userId, range = 2) {
    // Get user's rank
    const rank = await this.getRank(userId);
    if (!rank) return [];

    // Calculate range
    const start = Math.max(0, rank - range - 1);
    const end = rank + range - 1;

    // Get users
    const results = await this.redis.zrevrange(
      this.key,
      start,
      end,
      'WITHSCORES'
    );

    // Format results
    const leaderboard = [];
    for (let i = 0; i < results.length; i += 2) {
      leaderboard.push({
        userId: results[i],
        score: parseFloat(results[i + 1]),
        rank: start + i / 2 + 1,
      });
    }

    return leaderboard;
  }
}
```

4. **Rate Limiter**:

```
// Implementing a rate limiter with Redis
class RateLimiter {
```

```
  constructor(redisClient) {
    this.redis = redisClient;
  }

  // Fixed window rate limiter
  async fixedWindow(key, maxRequests, windowSeconds) {
    const currentCount = await this.redis.incr(`ratelimit:${key}`);

    // Set expiry on first request
    if (currentCount === 1) {
      await this.redis.expire(`ratelimit:${key}`, windowSeconds);
    }

    return {
      allowed: currentCount <= maxRequests,
      current: currentCount,
      remaining: Math.max(0, maxRequests - currentCount),
      ttl: await this.redis.ttl(`ratelimit:${key}`),
    };
  }

  // Sliding window rate limiter
  async slidingWindow(key, maxRequests, windowSeconds) {
    const now = Date.now();
    const windowStart = now - windowSeconds * 1000;

    // Remove old entries
    await this.redis.zremrangebyscore(`sliding:${key}`, 0, windowStart);

    // Add current request
    await this.redis.zadd(
      `sliding:${key}`,
      now,
      `${now}-${Math.random().toString(36)}`
    );

    // Set key expiry
    await this.redis.expire(`sliding:${key}`, windowSeconds);

    // Count requests in current window
    const currentCount = await this.redis.zcard(`sliding:${key}`);

    return {
      allowed: currentCount <= maxRequests,
      current: currentCount,
      remaining: Math.max(0, maxRequests - currentCount),
      ttl: windowSeconds,
    };
  }

  // Token bucket rate limiter
  async tokenBucket(key, maxTokens, refillRate, tokensPerRequest = 1) {
    const bucketKey = `bucket:${key}`;
    const now = Date.now();
```

```
    // Get or create bucket
    let bucket = await this.redis.hgetall(bucketKey);

    if (!Object.keys(bucket).length) {
      // Initialize bucket
      bucket = {
        tokens: maxTokens,
        lastRefill: now,
      };
    } else {
      // Convert string values to numbers
      bucket.tokens = parseFloat(bucket.tokens);
      bucket.lastRefill = parseInt(bucket.lastRefill);

      // Refill tokens based on time elapsed
      const elapsedSeconds = (now - bucket.lastRefill) / 1000;
      const tokensToAdd = elapsedSeconds * refillRate;

      bucket.tokens = Math.min(maxTokens, bucket.tokens + tokensToAdd);
      bucket.lastRefill = now;
    }

    // Try to consume tokens
    const allowed = bucket.tokens >= tokensPerRequest;

    if (allowed) {
      bucket.tokens -= tokensPerRequest;
    }

    // Save bucket state
    await this.redis.hmset(bucketKey, bucket);
    await this.redis.expire(bucketKey, Math.ceil(maxTokens / refillRate) * 2);

    return {
      allowed,
      remaining: Math.floor(bucket.tokens),
      resetAfter: (maxTokens - bucket.tokens) / refillRate,
    };
  }
}
```

Performance Considerations for In-Memory Databases

1. **Memory Management**:

```
# Redis memory configuration in redis.conf
maxmemory 1gb
maxmemory-policy allkeys-lru  # Options: noeviction, allkeys-lru, volatile-lru,
allkeys-random, volatile-random, volatile-ttl
```

2. **Connection Pooling**:

```
// Connection pooling with ioredis
const Redis = require('ioredis');

// Create a connection pool
const redisPool = new Redis.Cluster(
  [
    {
      host: 'redis-node1',
      port: 6379,
    },
    {
      host: 'redis-node2',
      port: 6379,
    },
    {
      host: 'redis-node3',
      port: 6379,
    },
  ],
  {
    // Pool options
    maxConnections: 100,
    // Retry strategy
    retryStrategy(times) {
      const delay = Math.min(times * 50, 2000);
      return delay;
    },
  }
);
```

3. **Pipelining and Transactions**:

```
// Pipelining in Redis
async function userBatchUpdate(users) {
  const pipeline = redis.pipeline();

  for (const user of users) {
    // Queue multiple operations
    pipeline.hset(`user:${user.id}`, user);
    pipeline.expire(`user:${user.id}`, 3600);

    if (user.score) {
      pipeline.zadd('user_scores', user.score, user.id);
    }
  }

  // Execute all commands in a single round-trip
  return pipeline.exec();
}
```

4. **Data Compression**:

```
// Compress large values with Node.js zlib
const zlib = require('zlib');
const util = require('util');
const compressAsync = util.promisify(zlib.deflate);
const decompressAsync = util.promisify(zlib.inflate);

async function setCompressed(redis, key, value, ttl = 3600) {
  // Compress value
  const compressed = await compressAsync(Buffer.from(JSON.stringify(value)));

  // Store with compression flag
  await redis.set(`${key}:compressed`, compressed, 'EX', ttl);
}

async function getCompressed(redis, key) {
  // Get compressed data
  const compressed = await redis.getBuffer(`${key}:compressed`);
  if (!compressed) return null;

  // Decompress
  const decompressed = await decompressAsync(compressed);

  // Parse JSON
  return JSON.parse(decompressed.toString());
}
```

5. **Persistence Tuning**:

```
# Optimized RDB persistence in redis.conf
save 900 1
save 300 10
save 60 10000
stop-writes-on-bgsave-error yes
rdbcompression yes
rdbchecksum yes

# Optimized AOF persistence in redis.conf
appendonly yes
appendfsync everysec
no-appendfsync-on-rewrite yes
auto-aof-rewrite-percentage 100
auto-aof-rewrite-min-size 64mb
```

5.5 NewSQL Databases

NewSQL databases aim to provide the ACID guarantees of traditional relational databases while achieving the horizontal scalability of NoSQL systems. They're suitable for applications requiring both strong consistency and scalability.

Core Concepts of NewSQL Databases

1. **Distributed SQL**: SQL interface with distributed storage and processing.
2. **Horizontal Scalability**: Ability to scale by adding nodes without sharding at the application level.
3. **ACID Guarantees**: Full transactional support across the distributed system.
4. **Automatic Sharding**: Data distribution without manual intervention.
5. **Distributed Query Processing**: Queries spanning multiple nodes and partitions.

CockroachDB: Resilient Distributed SQL

CockroachDB is inspired by Google's Spanner database, offering distributed SQL with strong consistency.

Setting Up CockroachDB

```
# Docker installation
docker run -d --name=roach1 -p 26257:26257 -p 8080:8080 cockroachdb/cockroach:latest
start-single-node --insecure
```

Basic CockroachDB Operations

```
-- Connect to CockroachDB
-- Using the cockroach SQL shell
$ cockroach sql --insecure --host=localhost

-- Create a database
CREATE DATABASE example;
USE example;

-- Create a table with a primary key
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name STRING NOT NULL,
  email STRING UNIQUE,
  created_at TIMESTAMPTZ DEFAULT now()
);

-- Insert data
INSERT INTO users (name, email) VALUES
  ('John Doe', 'john@example.com'),
  ('Jane Smith', 'jane@example.com');

-- Query data
SELECT * FROM users WHERE name LIKE 'J%';
```

Distributed Features

```
-- Create a table with explicit sharding
CREATE TABLE orders (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
```

```
  user_id UUID NOT NULL REFERENCES users(id),
  amount DECIMAL(19,4) NOT NULL,
  status STRING NOT NULL,
  created_at TIMESTAMPTZ DEFAULT now(),
  CONSTRAINT check_positive_amount CHECK (amount > 0)
) PARTITION BY RANGE (created_at) (
    PARTITION orders_2023_q1 VALUES FROM ('2023-01-01') TO ('2023-04-01'),
    PARTITION orders_2023_q2 VALUES FROM ('2023-04-01') TO ('2023-07-01'),
    PARTITION orders_2023_q3 VALUES FROM ('2023-07-01') TO ('2023-10-01'),
    PARTITION orders_2023_q4 VALUES FROM ('2023-10-01') TO ('2024-01-01')
);

-- View partitioning information
SHOW PARTITIONS FROM TABLE orders;

-- Multi-region configuration (Enterprise feature)
ALTER DATABASE example SET PRIMARY REGION "us-east";
ALTER DATABASE example ADD REGION "us-west";
ALTER DATABASE example ADD REGION "eu-west";

-- Configure table for global access pattern
ALTER TABLE users SET GLOBAL;

-- Configure table with regional access patterns
ALTER TABLE orders SET REGIONAL BY ROW;
```

Transactions in CockroachDB

```
-- Multi-statement transaction
BEGIN;

-- Update order status
UPDATE orders SET status = 'shipped' WHERE id = 'some-uuid';

-- Add shipment record
INSERT INTO shipments (order_id, carrier, tracking_number)
VALUES ('some-uuid', 'FedEx', '123456789');

COMMIT;
```

Using CockroachDB from Node.js

```
const { Pool } = require('pg'); // CockroachDB supports the PostgreSQL protocol

const pool = new Pool({
  host: 'localhost',
  port: 26257,
  user: 'root',
  password: '',
  database: 'example',
```

```
  ssl: false,
});

async function createUser(name, email) {
  const client = await pool.connect();

  try {
    await client.query('BEGIN');

    const result = await client.query(
      'INSERT INTO users (name, email) VALUES ($1, $2) RETURNING id',
      [name, email]
    );

    await client.query('COMMIT');
    return result.rows[0].id;
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}

async function getUserWithOrders(userId) {
  const client = await pool.connect();

  try {
    // Get user
    const userResult = await client.query('SELECT * FROM users WHERE id = $1', [
      userId,
    ]);

    if (userResult.rows.length === 0) {
      return null;
    }

    const user = userResult.rows[0];

    // Get user's orders
    const ordersResult = await client.query(
      'SELECT * FROM orders WHERE user_id = $1 ORDER BY created_at DESC',
      [userId]
    );

    // Combine results
    user.orders = ordersResult.rows;

    return user;
  } finally {
    client.release();
  }
}
```

Google Spanner and Cloud Spanner

Google Spanner is the original NewSQL database, with Cloud Spanner being its managed service offering.

Cloud Spanner Schema

```
-- Create a schema in Spanner
CREATE TABLE users (
  user_id STRING(36) NOT NULL,
  name STRING(100) NOT NULL,
  email STRING(100) NOT NULL,
  created_at TIMESTAMP NOT NULL OPTIONS (allow_commit_timestamp = true),
) PRIMARY KEY (user_id);

CREATE TABLE orders (
  order_id STRING(36) NOT NULL,
  user_id STRING(36) NOT NULL,
  amount NUMERIC NOT NULL,
  status STRING(20) NOT NULL,
  created_at TIMESTAMP NOT NULL OPTIONS (allow_commit_timestamp = true),
  CONSTRAINT FK_UserOrder FOREIGN KEY (user_id) REFERENCES users (user_id)
) PRIMARY KEY (order_id),
  INTERLEAVE IN PARENT users ON DELETE CASCADE;

-- Create an index for querying orders by user and date
CREATE INDEX OrdersByUserAndDate ON orders(user_id, created_at DESC);
```

Using Spanner from Node.js

```
const { Spanner } = require('@google-cloud/spanner');

// Initialize Spanner with your project and instance
const spanner = new Spanner({ projectId: 'your-project-id' });
const instance = spanner.instance('your-instance-id');
const database = instance.database('your-database-id');

// Create a user and an order in a transaction
async function createUserWithOrder(userData, orderData) {
  const transaction = await database.runTransaction();

  try {
    // Generate IDs
    const userId = generateUuid();
    const orderId = generateUuid();

    // Insert user
    await transaction.insert('users', {
      user_id: userId,
      name: userData.name,
      email: userData.email,
      created_at: Spanner.timestamp(new Date()),
```

```
      });

      // Insert order
      await transaction.insert('orders', {
        order_id: orderId,
        user_id: userId,
        amount: orderData.amount,
        status: 'created',
        created_at: Spanner.timestamp(new Date()),
      });

      // Commit transaction
      await transaction.commit();

      return { userId, orderId };
    } catch (error) {
      await transaction.rollback();
      throw error;
    }
  }

  // Query with a strongly consistent read
  async function getUserWithOrders(userId) {
    const [user] = await database.table('users').read({
      columns: ['user_id', 'name', 'email', 'created_at'],
      keys: [userId],
      limit: 1,
    });

    if (!user.length) {
      return null;
    }

    const [orders] = await database.run({
      sql: `
        SELECT * FROM orders
        WHERE user_id = @userId
        ORDER BY created_at DESC
      `,
      params: { userId },
    });

    user[0].orders = orders;
    return user[0];
  }
```

TiDB: MySQL Compatible NewSQL

TiDB is an open-source NewSQL database that's compatible with MySQL protocol and syntax.

Setting Up TiDB

```
# Docker installation (simplified version)
docker run -d -p 4000:4000 -p 10080:10080 pingcap/tidb:latest
```

Basic TiDB Usage

```
-- Connect to TiDB using MySQL client
mysql -h 127.0.0.1 -P 4000 -u root

-- Create a database
CREATE DATABASE example;
USE example;

-- Create a table
CREATE TABLE users (
  id BIGINT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(100) NOT NULL,
  email VARCHAR(100) UNIQUE KEY,
  created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Insert data
INSERT INTO users (name, email) VALUES
  ('John Doe', 'john@example.com'),
  ('Jane Smith', 'jane@example.com');

-- Query data
SELECT * FROM users;
```

TiDB Distributed Features

```
-- Create a partitioned table in TiDB
CREATE TABLE orders (
  id BIGINT AUTO_INCREMENT,
  user_id BIGINT NOT NULL,
  amount DECIMAL(10,2) NOT NULL,
  status VARCHAR(20) NOT NULL,
  order_date DATE NOT NULL,
  PRIMARY KEY (id, order_date)
) PARTITION BY RANGE (YEAR(order_date)) (
  PARTITION p2020 VALUES LESS THAN (2021),
  PARTITION p2021 VALUES LESS THAN (2022),
  PARTITION p2022 VALUES LESS THAN (2023),
  PARTITION p2023 VALUES LESS THAN (2024),
  PARTITION future VALUES LESS THAN MAXVALUE
);

-- Information about partitions
SHOW CREATE TABLE orders;
SELECT * FROM information_schema.partitions WHERE table_name = 'orders';
```

```
-- Create a table with hash sharding
CREATE TABLE user_events (
  user_id BIGINT NOT NULL,
  event_type VARCHAR(50) NOT NULL,
  event_time TIMESTAMP NOT NULL,
  data JSON,
  PRIMARY KEY (user_id, event_type, event_time)
);
```

Working with TiDB from Node.js

```
const mysql = require('mysql2/promise');

async function main() {
  // Connect to TiDB
  const connection = await mysql.createConnection({
    host: 'localhost',
    port: 4000,
    user: 'root',
    password: '',
    database: 'example',
  });

  try {
    // Start transaction
    await connection.beginTransaction();

    // Create order
    const [orderResult] = await connection.execute(
      'INSERT INTO orders (user_id, amount, status, order_date) VALUES (?, ?, ?,
?)',
      [1001, 199.99, 'created', new Date()]
    );

    const orderId = orderResult.insertId;

    // Add order items
    await connection.execute(
      'INSERT INTO order_items (order_id, product_id, quantity, price) VALUES (?, ?,
?, ?)',
      [orderId, 5001, 2, 99.99]
    );

    // Commit transaction
    await connection.commit();

    return orderId;
  } catch (error) {
    // Rollback on error
    await connection.rollback();
    throw error;
  } finally {
```

```
    await connection.end();
  }
}
```

Real-World Use Cases for NewSQL Databases

1. **Financial Systems**:

```
// Banking transaction system with CockroachDB
async function transferFunds(fromAccountId, toAccountId, amount) {
  const client = await pool.connect();

  try {
    // Start transaction with serializable isolation level
    await client.query('BEGIN ISOLATION LEVEL SERIALIZABLE');

    // Check sufficient funds
    const balanceResult = await client.query(
      'SELECT balance FROM accounts WHERE id = $1 FOR UPDATE',
      [fromAccountId]
    );

    if (balanceResult.rows.length === 0) {
      throw new Error(`Account ${fromAccountId} not found`);
    }

    const currentBalance = parseFloat(balanceResult.rows[0].balance);

    if (currentBalance < amount) {
      throw new Error('Insufficient funds');
    }

    // Check destination account exists
    const toAccountResult = await client.query(
      'SELECT id FROM accounts WHERE id = $1',
      [toAccountId]
    );

    if (toAccountResult.rows.length === 0) {
      throw new Error(`Destination account ${toAccountId} not found`);
    }

    // Deduct from source account
    await client.query(
      'UPDATE accounts SET balance = balance - $1 WHERE id = $2',
      [amount, fromAccountId]
    );

    // Add to destination account
    await client.query(
      'UPDATE accounts SET balance = balance + $1 WHERE id = $2',
      [amount, toAccountId]
    );
```

```
    // Record transaction
    await client.query(
      `INSERT INTO transactions
       (from_account_id, to_account_id, amount, type, status, created_at)
       VALUES ($1, $2, $3, 'transfer', 'completed', NOW())`,
      [fromAccountId, toAccountId, amount]
    );

    // Commit transaction
    await client.query('COMMIT');

    return { success: true };
  } catch (error) {
    await client.query('ROLLBACK');
    throw error;
  } finally {
    client.release();
  }
}
```

2. **E-commerce Platform**:

```
// TiDB implementation for handling flash sales with high concurrency
async function processOrder(userId, productId, quantity) {
  const connection = await pool.getConnection();

  try {
    await connection.beginTransaction();

    // Check product availability with pessimistic locking
    const [productRows] = await connection.query(
      'SELECT price, stock FROM products WHERE id = ? FOR UPDATE',
      [productId]
    );

    if (productRows.length === 0) {
      throw new Error('Product not found');
    }

    const product = productRows[0];

    if (product.stock < quantity) {
      throw new Error('Insufficient stock');
    }

    // Calculate total amount
    const totalAmount = product.price * quantity;

    // Check user exists
    const [userRows] = await connection.query(
      'SELECT id FROM users WHERE id = ?',
      [userId]
```

```
    );

    if (userRows.length === 0) {
      throw new Error('User not found');
    }

    // Create order
    const [orderResult] = await connection.query(
      `INSERT INTO orders
       (user_id, total_amount, status, order_date)
       VALUES (?, ?, 'created', NOW())`,
      [userId, totalAmount]
    );

    const orderId = orderResult.insertId;

    // Add order item
    await connection.query(
      `INSERT INTO order_items
       (order_id, product_id, quantity, price)
       VALUES (?, ?, ?, ?)`,
      [orderId, productId, quantity, product.price]
    );

    // Update product stock
    await connection.query(
      'UPDATE products SET stock = stock - ? WHERE id = ?',
      [quantity, productId]
    );

    // Add inventory movement record
    await connection.query(
      `INSERT INTO inventory_movements
       (product_id, quantity, movement_type, reference_id, reference_type)
       VALUES (?, ?, 'order', ?, 'order')`,
      [productId, -quantity, orderId]
    );

    await connection.commit();

    return { orderId, totalAmount };
  } catch (error) {
    await connection.rollback();
    throw error;
  } finally {
    connection.release();
  }
}
```

3. **Global SaaS Application**:

```
// Multi-region application with Cloud Spanner
async function createTenant(tenantData, regionPreference) {
```

```
  const transaction = await database.runTransaction();

  try {
    // Generate IDs
    const tenantId = generateUuid();

    // Insert tenant record
    await transaction.insert('tenants', {
      tenant_id: tenantId,
      name: tenantData.name,
      plan: tenantData.plan,
      region_preference: regionPreference,
      status: 'active',
      created_at: Spanner.timestamp(new Date()),
    });

    // Create default admin user
    await transaction.insert('users', {
      user_id: generateUuid(),
      tenant_id: tenantId,
      email: tenantData.adminEmail,
      role: 'admin',
      created_at: Spanner.timestamp(new Date()),
    });

    // Initialize tenant configuration
    await transaction.insert('tenant_configs', {
      tenant_id: tenantId,
      config_key: 'default_settings',
      config_value: JSON.stringify(getDefaultSettings(tenantData.plan)),
      created_at: Spanner.timestamp(new Date()),
    });

    // Create tenant storage allocation
    await transaction.insert('storage_allocations', {
      tenant_id: tenantId,
      storage_gb: getPlanStorage(tenantData.plan),
      created_at: Spanner.timestamp(new Date()),
    });

    // Commit transaction
    await transaction.commit();

    // Queue async provisioning job
    await queueTenantProvisioning(tenantId, regionPreference);

    return { tenantId };
  } catch (error) {
    await transaction.rollback();
    throw error;
  }
}

// Read tenant data with strong consistency
async function getTenantDetails(tenantId) {
```

```
  const [tenantRows] = await database.run({
    sql: `
      SELECT t.*, sa.storage_gb, sa.used_storage_gb
      FROM tenants t
      JOIN storage_allocations sa ON t.tenant_id = sa.tenant_id
      WHERE t.tenant_id = @tenantId
    `,
    params: { tenantId },
  });

  if (tenantRows.length === 0) {
    return null;
  }

  // Get tenant configuration
  const [configRows] = await database.run({
    sql: `
      SELECT config_key, config_value
      FROM tenant_configs
      WHERE tenant_id = @tenantId
    `,
    params: { tenantId },
  });

  // Get user count
  const [countResult] = await database.run({
    sql: `
      SELECT COUNT(*) as user_count
      FROM users
      WHERE tenant_id = @tenantId
    `,
    params: { tenantId },
  });

  return {
    ...tenantRows[0],
    config: configRows.reduce((acc, row) => {
      acc[row.config_key] = JSON.parse(row.config_value);
      return acc;
    }, {}),
    userCount: countResult[0].user_count,
  };
}
```

Performance Considerations for NewSQL Databases

1. **Partitioning Strategies**:

```
-- CockroachDB time-based partitioning
CREATE TABLE events (
  id UUID NOT NULL DEFAULT gen_random_uuid(),
  event_type STRING NOT NULL,
  user_id UUID NOT NULL,
```

```
  data JSONB,
  timestamp TIMESTAMPTZ NOT NULL DEFAULT now(),
  PRIMARY KEY (timestamp, id)
) PARTITION BY RANGE (timestamp) (
  PARTITION events_2023_q1 VALUES FROM ('2023-01-01') TO ('2023-04-01'),
  PARTITION events_2023_q2 VALUES FROM ('2023-04-01') TO ('2023-07-01'),
  PARTITION events_2023_q3 VALUES FROM ('2023-07-01') TO ('2023-10-01'),
  PARTITION events_2023_q4 VALUES FROM ('2023-10-01') TO ('2024-01-01')
);

-- TiDB hash partitioning
CREATE TABLE user_activities (
  user_id BIGINT NOT NULL,
  activity_id BIGINT NOT NULL,
  activity_type VARCHAR(50) NOT NULL,
  created_at TIMESTAMP NOT NULL,
  data JSON,
  PRIMARY KEY (user_id, activity_id)
) PARTITION BY HASH(user_id) PARTITIONS 8;
```

2. **Index Design**:

```
-- Optimized indexes for CockroachDB
CREATE TABLE products (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  name STRING NOT NULL,
  category STRING NOT NULL,
  price DECIMAL(10,2) NOT NULL,
  stock INT NOT NULL,
  created_at TIMESTAMPTZ NOT NULL DEFAULT now()
);

-- Index for category filtering and price sorting
CREATE INDEX idx_products_category_price ON products (category, price);

-- Index for stock level reporting
CREATE INDEX idx_products_stock ON products (stock) STORING (name, category);

-- Covering index that includes all needed fields
CREATE INDEX idx_products_category_created ON products (category, created_at)
STORING (name, price);
```

3. **Query Optimization**:

```
-- CockroachDB: Use EXPLAIN to analyze query plans
EXPLAIN ANALYZE SELECT * FROM orders WHERE user_id = 'some-uuid' AND created_at >
'2023-01-01';

-- TiDB: Use EXPLAIN to view execution plan
EXPLAIN SELECT p.*, c.name as category_name
FROM products p
```

```
JOIN categories c ON p.category_id = c.id
WHERE p.price > 100
ORDER BY p.created_at DESC
LIMIT 20;
```

4. **Transaction Design**:

```
// CockroachDB: Handling transaction conflicts
async function updateInventory(productId, quantity) {
  const maxRetries = 5;
  let retries = 0;

  while (retries < maxRetries) {
    const client = await pool.connect();

    try {
      await client.query('BEGIN');

      // Get current stock with FOR UPDATE
      const result = await client.query(
        'SELECT stock FROM products WHERE id = $1 FOR UPDATE',
        [productId]
      );

      if (result.rows.length === 0) {
        await client.query('ROLLBACK');
        throw new Error('Product not found');
      }

      const currentStock = result.rows[0].stock;

      if (currentStock < quantity) {
        await client.query('ROLLBACK');
        throw new Error('Insufficient stock');
      }

      // Update stock
      await client.query(
        'UPDATE products SET stock = stock - $1 WHERE id = $2',
        [quantity, productId]
      );

      await client.query('COMMIT');
      return { success: true, remaining: currentStock - quantity };
    } catch (error) {
      await client.query('ROLLBACK');

      // Check if it's a transaction retry error
      if (error.code === '40001') {
        retries++;
        // Exponential backoff
        await new Promise((r) => setTimeout(r, Math.pow(2, retries) * 50));
      } else {
```

```
        throw error;
      }
    } finally {
      client.release();
    }
  }

  throw new Error('Transaction failed after maximum retries');
}
```

5. **Scaling Considerations**:

```
// Spanner scaling strategy for batch operations
async function batchProcessRecords(records) {
  // Split into smaller batches to avoid transaction size limits
  const batchSize = 500;
  const batches = [];

  for (let i = 0; i < records.length; i += batchSize) {
    batches.push(records.slice(i, i + batchSize));
  }

  const results = [];

  // Process batches with multiple concurrent transactions
  const batchPromises = batches.map(async (batch) => {
    const transaction = await database.runTransaction();

    try {
      for (const record of batch) {
        // Process record
        transaction.insert('processed_records', {
          id: record.id,
          data: record.data,
          processed_at: Spanner.timestamp(new Date()),
        });
      }

      // Commit batch
      await transaction.commit();
      return batch.length;
    } catch (error) {
      await transaction.rollback();
      throw error;
    }
  });

  // Wait for all batches to complete
  const batchResults = await Promise.allSettled(batchPromises);

  // Count successes and failures
  const processed = batchResults
    .filter((r) => r.status === 'fulfilled')
```

```
    .reduce((sum, r) => sum + r.value, 0);

  const errors = batchResults
    .filter((r) => r.status === 'rejected')
    .map((r) => r.reason.message);

  return { processed, errors };
}
```

F. Advanced Topics

This section covers advanced concepts in database management, focusing on distributed databases, scaling strategies, data warehousing, and cloud database services.

6.1 Distributed Database Concepts

Distributed databases spread data across multiple nodes or geographic locations, offering improved scalability, availability, and fault tolerance.

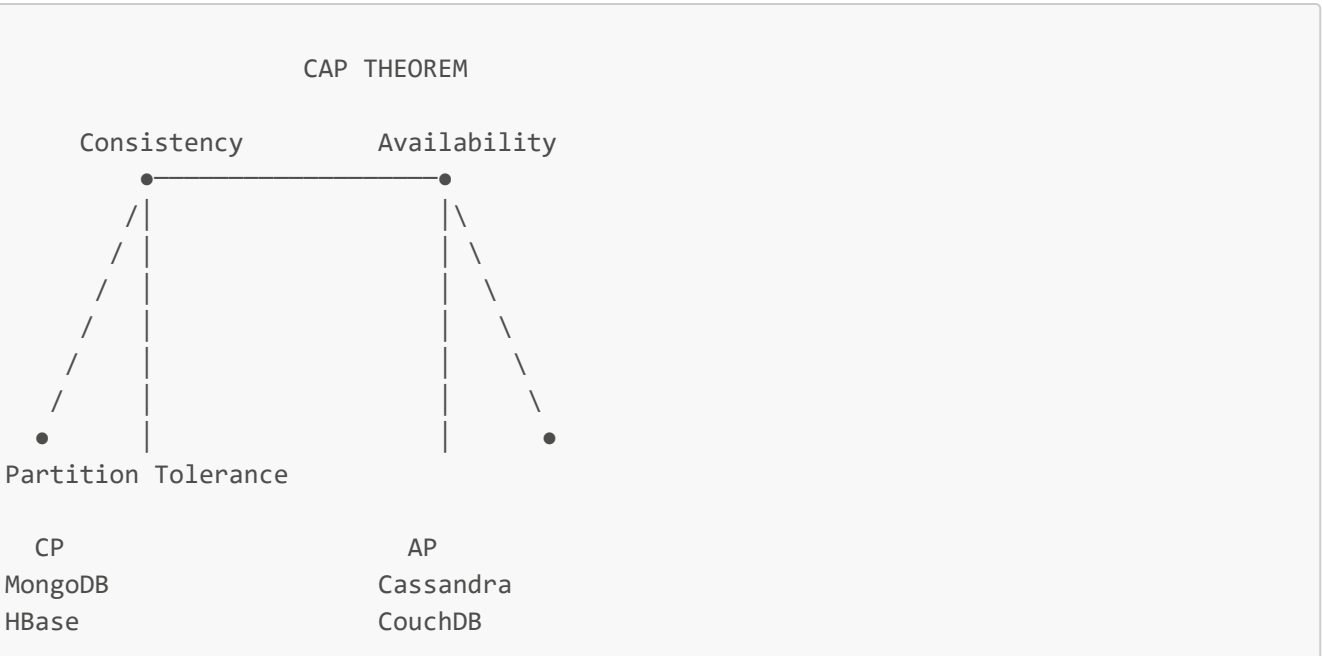
Core Concepts of Distributed Databases

1. **Data Distribution**: How data is spread across nodes.
2. **Consistency Models**: Guarantees about data consistency in distributed systems.
3. **CAP Theorem**: The tradeoff between Consistency, Availability, and Partition Tolerance.
4. **Distributed Transactions**: Managing transactions across multiple nodes.
5. **Consensus Algorithms**: How nodes agree on a shared state.

CAP Theorem and Its Implications

The CAP theorem states that distributed systems can provide at most two of these three guarantees:

- **Consistency**: All nodes see the same data at the same time.
- **Availability**: Every request receives a response, without guarantee of the data being the latest.
- **Partition Tolerance**: The system continues to operate despite network partitions.



```
Redis Cluster           Riak

            CA
        Traditional RDBMS
        (Not partition tolerant)
```

Practical Implications of CAP Choices

```
// AP System Example (Cassandra)
const cassandra = require('cassandra-driver');

const client = new cassandra.Client({
  contactPoints: ['node1', 'node2', 'node3'],
  localDataCenter: 'datacenter1',
  keyspace: 'mykeyspace',
});

// Write with eventual consistency
async function writeWithEventualConsistency(key, value) {
  const query = 'INSERT INTO key_value (key, value) VALUES (?, ?)';

  // Using LOCAL_ONE consistency level for better availability
  const options = { consistency: cassandra.types.consistencies.LOCAL_ONE };

  await client.execute(query, [key, value], options);
}

// Read with eventual consistency
async function readWithEventualConsistency(key) {
  const query = 'SELECT value FROM key_value WHERE key = ?';

  // Using LOCAL_ONE consistency level
  const options = { consistency: cassandra.types.consistencies.LOCAL_ONE };

  const result = await client.execute(query, [key], options);
  return result.rows[0]?.value;
}
```

```
// CP System Example (MongoDB)
const { MongoClient } = require('mongodb');

const client = new MongoClient('mongodb://node1,node2,node3/?replicaSet=rs0');

// Write with strong consistency
async function writeWithStrongConsistency(key, value) {
  await client.connect();

  // Using "majority" write concern for consistency
  const options = { writeConcern: { w: 'majority' } };
```

```
  await client
    .db('mydb')
    .collection('key_value')
    .insertOne({ key, value }, options);
}

// Read with strong consistency
async function readWithStrongConsistency(key) {
  await client.connect();

  // Using "majority" read concern for consistency
  const options = { readConcern: { level: 'majority' } };

  const result = await client
    .db('mydb')
    .collection('key_value')
    .findOne({ key }, options);

  return result?.value;
}
```

Consistency Models in Distributed Systems

1. **Strong Consistency**: All reads reflect the most recent write.
2. **Eventual Consistency**: Given enough time without updates, all replicas will converge.
3. **Causal Consistency**: Operations that are causally related will be seen in the same order by all nodes.
4. **Session Consistency**: A client's reads will reflect its own writes within a session.

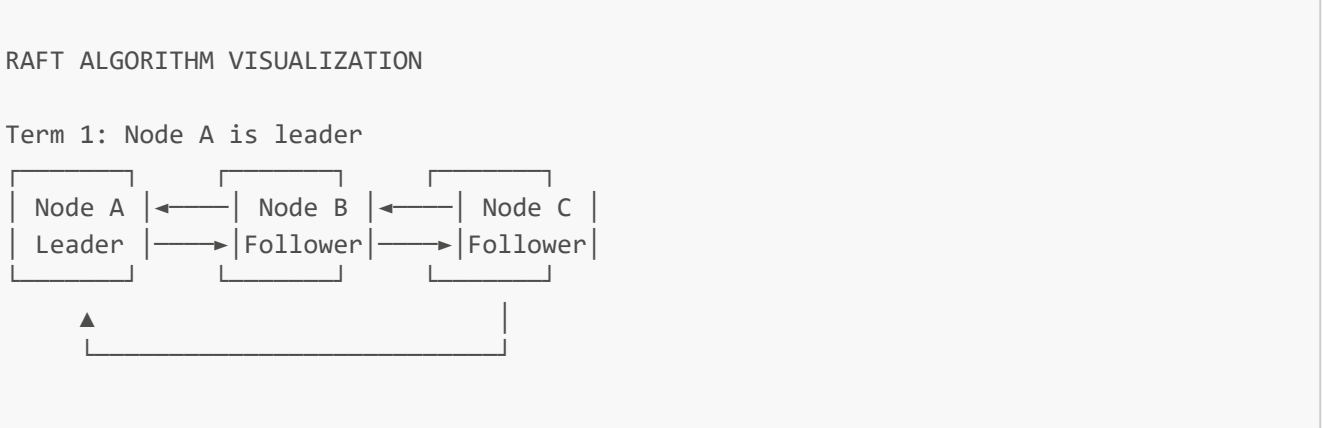
```
// Example of different consistency levels in Cassandra
async function demonstrateConsistencyLevels() {
  // Write with strong consistency (ALL)
  const strongWriteOptions = { consistency: cassandra.types.consistencies.ALL };
  await client.execute(
    'INSERT INTO messages (id, content) VALUES (?, ?)',
    [uuidv4(), 'Important message'],
    strongWriteOptions
  );

  // Read with strong consistency (ALL)
  const strongReadOptions = { consistency: cassandra.types.consistencies.ALL };
  const strongResult = await client.execute(
    'SELECT * FROM messages WHERE id = ?',
    [messageId],
    strongReadOptions
  );

  // Write with medium consistency (QUORUM)
  const quorumWriteOptions = {
    consistency: cassandra.types.consistencies.QUORUM,
  };
```

```
  await client.execute(
    'INSERT INTO messages (id, content) VALUES (?, ?)',
    [uuidv4(), 'Regular message'],
    quorumWriteOptions
  );

  // Read with medium consistency (QUORUM)
  const quorumReadOptions = {
    consistency: cassandra.types.consistencies.QUORUM,
  };
  const quorumResult = await client.execute(
    'SELECT * FROM messages WHERE id = ?',
    [messageId],
    quorumReadOptions
  );

  // Write with relaxed consistency (ONE)
  const relaxedWriteOptions = {
    consistency: cassandra.types.consistencies.ONE,
  };
  await client.execute(
    'INSERT INTO messages (id, content) VALUES (?, ?)',
    [uuidv4(), 'Low priority message'],
    relaxedWriteOptions
  );

  // Read with relaxed consistency (ONE)
  const relaxedReadOptions = { consistency: cassandra.types.consistencies.ONE };
  const relaxedResult = await client.execute(
    'SELECT * FROM messages WHERE id = ?',
    [messageId],
    relaxedReadOptions
  );
}
```

Consensus Algorithms

Consensus algorithms allow distributed systems to agree on a single value or state.

1. **Paxos**: A family of protocols for solving consensus in unreliable networks.
2. **Raft**: A more understandable consensus algorithm, used in systems like etcd and CockroachDB.



```
1. Leader sends heartbeats to followers
2. Followers respond to heartbeats
3. If leader fails, election timeout occurs

Term 2: Leader fails, election starts
┌─────────┐     ┌─────────┐     ┌─────────┐
│ Node A  │     │ Node B  │◄────┤ Node C  │
│  Down   │     │Candidate├────►│Follower│
└─────────┘     └─────────┘     └─────────┘

Term 3: New leader elected
┌─────────┐     ┌─────────┐     ┌─────────┐
│ Node A  │     │ Node B  │◄────┤ Node C  │
│  Down   │     │ Leader  ├────►│Follower│
└─────────┘     └─────────┘     └─────────┘
```

3. **ZAB (ZooKeeper Atomic Broadcast)**: The consensus algorithm used in Apache ZooKeeper.

Distributed Transactions

Distributed transactions ensure that operations across multiple nodes either all succeed or all fail.

Two-Phase Commit (2PC)

```
TWO-PHASE COMMIT PROTOCOL

Phase 1: Prepare
Coordinator                 Participants
     │                           │
     ├──────── Prepare ─────────►│
     │                           │
     │◄──── Vote (Yes/No) ──────┤
     │                           │

Phase 2: Commit/Abort
     │                           │
     ├─────Commit/Abort─────────►│
     │                           │
     │◄────Acknowledgment────────┤
     │                           │
```

```
// Two-Phase Commit simulation in Node.js
class TwoPhaseCommitCoordinator {
  constructor(participants) {
    this.participants = participants;
  }

  async executeTransaction(operations) {
    // Phase 1: Prepare
    try {
```

```
      const prepareResults = await Promise.all(
        this.participants.map((p) => p.prepare(operations))
      );

      // Check if all participants voted YES
      const allPrepared = prepareResults.every(
        (result) => result.status === 'PREPARED'
      );

      if (allPrepared) {
        // Phase 2a: Commit
        await Promise.all(this.participants.map((p) => p.commit(operations)));
        return { status: 'COMMITTED' };
      } else {
        // Phase 2b: Abort
        await Promise.all(this.participants.map((p) => p.abort(operations)));
        return { status: 'ABORTED', reason: 'Some participants voted NO' };
      }
    } catch (error) {
      // Error occurred, abort
      try {
        await Promise.all(this.participants.map((p) => p.abort(operations)));
      } catch (abortError) {
        console.error('Error during abort:', abortError);
      }

      return { status: 'ABORTED', reason: error.message };
    }
  }
}

// Participant implementation
class Participant {
  constructor(name, db) {
    this.name = name;
    this.db = db;
  }

  async prepare(operations) {
    try {
      // Check if operations can be executed
      await this.db.beginTransaction();

      // Validate operations
      for (const op of operations) {
        await this.validateOperation(op);
      }

      // Mark as prepared in a log
      await this.db.execute('INSERT INTO transaction_log (status) VALUES (?)', [
        'PREPARED',
      ]);

      return { status: 'PREPARED' };
    } catch (error) {
```

```
        await this.db.rollback();
        return { status: 'REJECTED', reason: error.message };
      }
    }

    async commit(operations) {
      try {
        // Execute operations
        for (const op of operations) {
          await this.executeOperation(op);
        }

        // Update log
        await this.db.execute(
          'UPDATE transaction_log SET status = ? WHERE id = ?',
          ['COMMITTED', this.currentTransactionId]
        );

        await this.db.commit();
        return { status: 'COMMITTED' };
      } catch (error) {
        await this.db.rollback();
        throw error;
      }
    }

    async abort(operations) {
      await this.db.rollback();

      // Update log
      try {
        await this.db.execute(
          'UPDATE transaction_log SET status = ? WHERE id = ?',
          ['ABORTED', this.currentTransactionId]
        );
      } catch (error) {
        console.error('Failed to update transaction log:', error);
      }

      return { status: 'ABORTED' };
    }
  }
```

Saga Pattern

The Saga pattern is an alternative to two-phase commit, using a sequence of local transactions with compensating transactions for rollback.

```
// Saga pattern implementation
class SagaCoordinator {
  constructor() {
    this.steps = [];
  }
```

```
  addStep(execute, compensate) {
    this.steps.push({ execute, compensate });
    return this;
  }

  async execute() {
    const completedSteps = [];
    let result = null;

    try {
      // Execute each step in sequence
      for (const step of this.steps) {
        result = await step.execute();
        completedSteps.push(step);
      }

      return { status: 'COMPLETED', result };
    } catch (error) {
      // Failure occurred, execute compensating transactions in reverse order
      for (const step of completedSteps.reverse()) {
        try {
          await step.compensate(result);
        } catch (compensateError) {
          console.error('Compensation error:', compensateError);
          // Continue with other compensations
        }
      }

      return { status: 'FAILED', error: error.message };
    }
  }
}

// Example usage: Order processing saga
async function processOrder(orderId, userId, items) {
  const saga = new SagaCoordinator();

  // Step 1: Verify inventory
  saga.addStep(
    async () => {
      const inventoryResult = await inventoryService.checkAvailability(items);
      if (!inventoryResult.available) {
        throw new Error(
          `Items not available: ${inventoryResult.unavailableItems.join(', ')}`
        );
      }
      return inventoryResult;
    },
    async () => {
      // No compensation needed for inventory check
    }
  );

  // Step 2: Reserve inventory
```

```
  saga.addStep(
    async () => {
      return await inventoryService.reserveItems(orderId, items);
    },
    async () => {
      await inventoryService.releaseItems(orderId, items);
    }
  );

  // Step 3: Process payment
  saga.addStep(
    async () => {
      const amount = items.reduce(
        (sum, item) => sum + item.price * item.quantity,
        0
      );
      return await paymentService.processPayment(userId, orderId, amount);
    },
    async (paymentResult) => {
      if (paymentResult && paymentResult.transactionId) {
        await paymentService.refundPayment(paymentResult.transactionId);
      }
    }
  );

  // Step 4: Update order status
  saga.addStep(
    async () => {
      return await orderService.updateStatus(orderId, 'CONFIRMED');
    },
    async () => {
      await orderService.updateStatus(orderId, 'CANCELLED');
    }
  );

  // Execute the saga
  return await saga.execute();
}
```

Sharding and Partitioning Strategies

Sharding divides data across multiple servers, improving performance and scalability.

1. **Range-Based Sharding**: Partitioning data based on ranges of a key.

```
// Range-based sharding implementation for user data
class RangeShardManager {
  constructor() {
    this.shardConfig = [
      { min: 'A', max: 'F', server: 'shard1.example.com' },
      { min: 'G', max: 'M', server: 'shard2.example.com' },
      { min: 'N', max: 'T', server: 'shard3.example.com' },
      { min: 'U', max: 'Z', server: 'shard4.example.com' },
```

```
    ];

    // Connection pool for each shard
    this.connections = new Map();
  }

  // Get shard for a username (sharding by first letter)
  getShardForUsername(username) {
    if (!username || typeof username !== 'string') {
      throw new Error('Invalid username');
    }

    const firstChar = username.charAt(0).toUpperCase();

    const shard = this.shardConfig.find(
      (s) => firstChar >= s.min && firstChar <= s.max
    );

    if (!shard) {
      throw new Error(`No shard found for username: ${username}`);
    }

    return shard;
  }

  // Get connection to the appropriate shard
  async getConnection(username) {
    const shard = this.getShardForUsername(username);

    if (!this.connections.has(shard.server)) {
      // Create new connection
      const connection = await createDbConnection(shard.server);
      this.connections.set(shard.server, connection);
    }

    return this.connections.get(shard.server);
  }

  // Execute a query on the appropriate shard
  async executeQuery(username, query, params) {
    const connection = await this.getConnection(username);
    return connection.execute(query, params);
  }

  // Close all connections
  async close() {
    const closePromises = Array.from(this.connections.values()).map((conn) =>
      conn.close()
    );

    await Promise.all(closePromises);
    this.connections.clear();
  }
}
```

2. **Hash-Based Sharding**: Using a hash function to distribute data across shards.

```
// Hash-based sharding for user data
class HashShardManager {
  constructor(shardCount = 8) {
    this.shardCount = shardCount;
    this.shardServers = [];

    // Initialize shard servers
    for (let i = 0; i < shardCount; i++) {
      this.shardServers.push(`shard${i}.example.com`);
    }

    this.connections = new Map();
  }

  // Get shard for a user ID
  getShardForUserId(userId) {
    // Convert string ID to number if needed
    const numericId =
      typeof userId === 'string'
        ? parseInt(userId.replace(/[^0-9]/g, ''), 10) || 0
        : userId;

    // Simple hash function: modulo
    const shardIndex = numericId % this.shardCount;
    return {
      index: shardIndex,
      server: this.shardServers[shardIndex],
    };
  }

  // Get connection to the appropriate shard
  async getConnection(userId) {
    const shard = this.getShardForUserId(userId);

    if (!this.connections.has(shard.server)) {
      // Create new connection
      const connection = await createDbConnection(shard.server);
      this.connections.set(shard.server, connection);
    }

    return this.connections.get(shard.server);
  }

  // Execute a query on a specific shard
  async executeQuery(userId, query, params) {
    const connection = await this.getConnection(userId);
    return connection.execute(query, params);
  }

  // Execute a query across all shards (for aggregation)
  async executeQueryAcrossShards(query, params) {
    const connectionPromises = this.shardServers.map((server) =>
```

```
        this.connections.has(server)
          ? this.connections.get(server)
          : createDbConnection(server)
      );

      const connections = await Promise.all(connectionPromises);

      // Execute query on all shards
      const results = await Promise.all(
        connections.map((conn) => conn.execute(query, params))
      );

      // Merge results (implementation depends on query type)
      return this.mergeResults(results);
    }

    // Helper method to merge results from multiple shards
    mergeResults(results) {
      // Implementation depends on query type (e.g., SUM, COUNT, etc.)
      // Simple example for row combination:
      return results.flatMap((result) => result.rows || []);
    }
  }
```

3. **Directory-Based Sharding**: Using a lookup service to track data location.

```
// Directory-based sharding implementation
class DirectoryShardManager {
  constructor(directoryServiceUrl) {
    this.directoryServiceUrl = directoryServiceUrl;
    this.connections = new Map();
    this.cache = new Map(); // Local cache of shard locations
    this.cacheTTL = 60000; // 1 minute TTL for cache entries
  }

  // Get shard for a entity
  async getShardForKey(entityType, entityId) {
    const cacheKey = `${entityType}:${entityId}`;

    // Check cache first
    if (this.cache.has(cacheKey)) {
      const cachedValue = this.cache.get(cacheKey);
      if (cachedValue.expiry > Date.now()) {
        return cachedValue.shard;
      }
      // Cache expired, remove entry
      this.cache.delete(cacheKey);
    }

    // Query directory service
    const response = await fetch(
      `${this.directoryServiceUrl}/lookup?type=${entityType}&id=${entityId}`
    );
```

```
    if (!response.ok) {
      throw new Error(`Directory service error: ${response.statusText}`);
    }

    const data = await response.json();

    // Cache result
    this.cache.set(cacheKey, {
      shard: data.shard,
      expiry: Date.now() + this.cacheTTL,
    });

    return data.shard;
  }

  // Get connection to the appropriate shard
  async getConnection(entityType, entityId) {
    const shard = await this.getShardForKey(entityType, entityId);

    if (!this.connections.has(shard.server)) {
      // Create new connection
      const connection = await createDbConnection(shard.server);
      this.connections.set(shard.server, connection);
    }

    return this.connections.get(shard.server);
  }

  // Execute a query on the appropriate shard
  async executeQuery(entityType, entityId, query, params) {
    const connection = await this.getConnection(entityType, entityId);
    return connection.execute(query, params);
  }

  // Register a new entity in the directory
  async registerEntity(entityType, entityId, shardId) {
    const response = await fetch(`${this.directoryServiceUrl}/register`, {
      method: 'POST',
      headers: {
        'Content-Type': 'application/json',
      },
      body: JSON.stringify({
        type: entityType,
        id: entityId,
        shardId: shardId,
      }),
    });

    if (!response.ok) {
      throw new Error(`Failed to register entity: ${response.statusText}`);
    }

    return response.json();
```

```
    }
  }
```

Replication Approaches

Replication creates and maintains copies of data across multiple nodes.

1. **Synchronous Replication**: Primary node waits for acknowledgment from replicas before confirming writes.

```
// Synchronous replication implementation
class SynchronousReplicationManager {
  constructor(primaryNodeUrl, replicaUrls) {
    this.primaryNodeUrl = primaryNodeUrl;
    this.replicaUrls = replicaUrls;
    this.allNodes = [primaryNodeUrl, ...replicaUrls];
    this.connections = new Map();
  }

  // Initialize connections
  async initialize() {
    for (const url of this.allNodes) {
      const connection = await createDbConnection(url);
      this.connections.set(url, connection);
    }
  }

  // Execute write query with synchronous replication
  async executeWrite(query, params) {
    // Execute on primary first
    const primaryConnection = this.connections.get(this.primaryNodeUrl);
    const primaryResult = await primaryConnection.execute(query, params);

    // Get transaction ID or timestamp from primary operation
    const transactionId = primaryResult.transactionId;

    // Synchronously replicate to all replicas
    const replicaPromises = this.replicaUrls.map((url) => {
      const connection = this.connections.get(url);
      return connection.execute(query, { ...params, transactionId });
    });

    // Wait for all replicas to confirm
    await Promise.all(replicaPromises);

    return primaryResult;
  }

  // Execute read query (from primary only)
  async executeReadFromPrimary(query, params) {
    const connection = this.connections.get(this.primaryNodeUrl);
    return connection.execute(query, params);
  }
```

```
  // Execute read query (from any replica, for load balancing)
  async executeReadFromReplica(query, params) {
    // Simple round-robin load balancing
    const replicaIndex = Math.floor(Math.random() * this.replicaUrls.length);
    const replicaUrl = this.replicaUrls[replicaIndex];

    const connection = this.connections.get(replicaUrl);
    return connection.execute(query, params);
  }

  // Close all connections
  async close() {
    const closePromises = Array.from(this.connections.values()).map((conn) =>
      conn.close()
    );

    await Promise.all(closePromises);
    this.connections.clear();
  }
}
```

2. **Asynchronous Replication**: Primary node confirms writes immediately, replicas update in the background.

```
// Asynchronous replication implementation
class AsynchronousReplicationManager {
  constructor(primaryNodeUrl, replicaUrls) {
    this.primaryNodeUrl = primaryNodeUrl;
    this.replicaUrls = replicaUrls;
    this.connections = new Map();
    this.replicationQueue = [];
    this.isProcessingQueue = false;
  }

  // Initialize connections
  async initialize() {
    // Connect to primary
    const primaryConnection = await createDbConnection(this.primaryNodeUrl);
    this.connections.set(this.primaryNodeUrl, primaryConnection);

    // Connect to replicas
    for (const url of this.replicaUrls) {
      const connection = await createDbConnection(url);
      this.connections.set(url, connection);
    }

    // Start processing queue
    this.startQueueProcessor();
  }

  // Execute write query with asynchronous replication
  async executeWrite(query, params) {
    // Execute on primary first
    const primaryConnection = this.connections.get(this.primaryNodeUrl);
```

```
    const primaryResult = await primaryConnection.execute(query, params);

    // Add to replication queue
    this.replicationQueue.push({
      query,
      params,
      transactionId: primaryResult.transactionId,
      timestamp: Date.now(),
    });

    return primaryResult;
  }

  // Process replication queue
  async startQueueProcessor() {
    if (this.isProcessingQueue) return;
    this.isProcessingQueue = true;

    const processQueue = async () => {
      try {
        if (this.replicationQueue.length === 0) {
          // No items to process, wait a bit
          await new Promise((resolve) => setTimeout(resolve, 100));
          return;
        }

        // Get next item from queue
        const item = this.replicationQueue.shift();

        // Replicate to all replicas
        const replicaPromises = this.replicaUrls.map((url) => {
          const connection = this.connections.get(url);
          return connection
            .execute(item.query, {
              ...item.params,
              transactionId: item.transactionId,
            })
            .catch((error) => {
              console.error(`Replication error to ${url}:`, error);
              // Re-add to queue for retry
              this.replicationQueue.push(item);
            });
        });

        await Promise.allSettled(replicaPromises);
      } catch (error) {
        console.error('Error processing replication queue:', error);
      }

      // Schedule next processing
      setTimeout(processQueue, 10);
    };

    // Start processing
    processQueue();
```

```
  }

  // Execute read query (from primary for consistency)
  async executeReadFromPrimary(query, params) {
    const connection = this.connections.get(this.primaryNodeUrl);
    return connection.execute(query, params);
  }

  // Execute read query (from any replica, with potential staleness)
  async executeReadFromReplica(query, params) {
    // Simple round-robin load balancing
    const replicaIndex = Math.floor(Math.random() * this.replicaUrls.length);
    const replicaUrl = this.replicaUrls[replicaIndex];

    const connection = this.connections.get(replicaUrl);
    return connection.execute(query, params);
  }
}
```

3. **Semi-Synchronous Replication**: Primary waits for acknowledgment from a subset of replicas before confirming.

```
// Semi-synchronous replication implementation
class SemiSynchronousReplicationManager {
  constructor(primaryNodeUrl, replicaUrls, minAcks = 1) {
    this.primaryNodeUrl = primaryNodeUrl;
    this.replicaUrls = replicaUrls;
    this.minAcks = Math.min(minAcks, replicaUrls.length);
    this.connections = new Map();
    this.replicationQueue = [];
  }

  // Initialize connections
  async initialize() {
    // Connect to primary
    const primaryConnection = await createDbConnection(this.primaryNodeUrl);
    this.connections.set(this.primaryNodeUrl, primaryConnection);

    // Connect to replicas
    for (const url of this.replicaUrls) {
      const connection = await createDbConnection(url);
      this.connections.set(url, connection);
    }
  }

  // Execute write query with semi-synchronous replication
  async executeWrite(query, params) {
    // Execute on primary first
    const primaryConnection = this.connections.get(this.primaryNodeUrl);
    const primaryResult = await primaryConnection.execute(query, params);

    // Get transaction ID from primary operation
    const transactionId = primaryResult.transactionId;
```

```
    // Create promises for all replicas
    const replicaPromises = this.replicaUrls.map((url) => {
      const connection = this.connections.get(url);
      return connection
        .execute(query, { ...params, transactionId })
        .then(() => true)
        .catch(() => false);
    });

    // Wait for at least minAcks replicas to confirm
    const results = await Promise.allSettled(replicaPromises);
    const successCount = results.filter(
      (r) => r.status === 'fulfilled' && r.value === true
    ).length;

    if (successCount < this.minAcks) {
      throw new Error(
        `Semi-synchronous replication failed: only ${successCount} replicas
acknowledged (required: ${this.minAcks})`
      );
    }

    // Queue async replication for remaining replicas
    const failedReplicas = results
      .map((r, i) => ({ result: r, index: i }))
      .filter(
        (item) => item.result.status !== 'fulfilled' || !item.result.value
      )
      .map((item) => this.replicaUrls[item.index]);

    if (failedReplicas.length > 0) {
      // Add to background replication queue
      this.replicationQueue.push({
        query,
        params: { ...params, transactionId },
        replicaUrls: failedReplicas,
        timestamp: Date.now(),
      });

      // Process queue asynchronously
      setTimeout(() => this.processReplicationQueue(), 0);
    }

    return primaryResult;
  }

  // Process replication queue in background
  async processReplicationQueue() {
    if (this.replicationQueue.length === 0) return;

    const item = this.replicationQueue.shift();

    // Try to replicate to failed replicas
    for (const url of item.replicaUrls) {
```

```
        try {
          const connection = this.connections.get(url);
          await connection.execute(item.query, item.params);
        } catch (error) {
          console.error(`Background replication error to ${url}:`, error);
          // Could implement more sophisticated retry logic here
        }
      }

      // Process next item if any
      if (this.replicationQueue.length > 0) {
        setTimeout(() => this.processReplicationQueue(), 10);
      }
    }
  }
```

Data Consistency Strategies

Strategies for maintaining data consistency in distributed systems.

1. **Quorum-Based Consistency**: Ensuring a majority of nodes agree on data operations.

```
// Quorum-based consistency manager
class QuorumConsistencyManager {
  constructor(nodes, readQuorum, writeQuorum) {
    this.nodes = nodes;
    this.readQuorum = readQuorum;
    this.writeQuorum = writeQuorum;
    this.connections = new Map();

    // Validate quorum settings
    if (this.readQuorum + this.writeQuorum <= nodes.length) {
      console.warn(
        'Warning: Read and write quorums do not guarantee consistency!'
      );
    }
  }

  // Initialize connections
  async initialize() {
    for (const node of this.nodes) {
      const connection = await createDbConnection(node);
      this.connections.set(node, connection);
    }
  }

  // Execute read with quorum
  async read(key) {
    // Select random subset of nodes to query
    const shuffledNodes = [...this.nodes].sort(() => 0.5 - Math.random());
    const nodesToQuery = shuffledNodes.slice(0, this.readQuorum);

    // Query selected nodes
```

```
      const promises = nodesToQuery.map(async (node) => {
        try {
          const connection = this.connections.get(node);
          const result = await connection.execute(
            'SELECT value, timestamp FROM key_value WHERE key = ?',
            [key]
          );

          if (result.rows.length === 0) {
            return { node, value: null, timestamp: 0 };
          }

          return {
            node,
            value: result.rows[0].value,
            timestamp: result.rows[0].timestamp,
          };
        } catch (error) {
          console.error(`Error reading from ${node}:`, error);
          return { node, error };
        }
      });

      const results = await Promise.all(promises);

      // Count successful responses
      const successfulResponses = results.filter((r) => !r.error);

      if (successfulResponses.length < this.readQuorum) {
        throw new Error(
          `Failed to achieve read quorum
(${successfulResponses.length}/${this.readQuorum})`
        );
      }

      // Find the value with the latest timestamp
      const latestResponse = successfulResponses.reduce(
        (latest, current) => {
          return current.timestamp > latest.timestamp ? current : latest;
        },
        { timestamp: 0 }
      );

      return latestResponse.value;
    }

    // Execute write with quorum
    async write(key, value) {
      const timestamp = Date.now();

      // Select random subset of nodes to write to
      const shuffledNodes = [...this.nodes].sort(() => 0.5 - Math.random());
      const nodesToWrite = shuffledNodes.slice(0, this.writeQuorum);

      // Write to selected nodes
```

```
    const promises = nodesToWrite.map(async (node) => {
      try {
        const connection = this.connections.get(node);
        await connection.execute(
          'INSERT INTO key_value (key, value, timestamp) VALUES (?, ?, ?) ' +
            'ON DUPLICATE KEY UPDATE value = IF(timestamp < ?, ?, value), timestamp
= IF(timestamp < ?, ?, timestamp)',
          [key, value, timestamp, timestamp, value, timestamp, timestamp]
        );
        return { node, success: true };
      } catch (error) {
        console.error(`Error writing to ${node}:`, error);
        return { node, success: false, error };
      }
    });

    const results = await Promise.all(promises);

    // Count successful writes
    const successfulWrites = results.filter((r) => r.success).length;

    if (successfulWrites < this.writeQuorum) {
      throw new Error(
        `Failed to achieve write quorum (${successfulWrites}/${this.writeQuorum})`
      );
    }

    // Schedule background anti-entropy to sync remaining nodes
    setTimeout(() => this.repairInconsistency(key, value, timestamp), 0);

    return { success: true, timestamp };
  }

  // Background process to synchronize data
  async repairInconsistency(key, value, timestamp) {
    for (const node of this.nodes) {
      try {
        const connection = this.connections.get(node);
        await connection.execute(
          'INSERT INTO key_value (key, value, timestamp) VALUES (?, ?, ?) ' +
            'ON DUPLICATE KEY UPDATE value = IF(timestamp < ?, ?, value), timestamp
= IF(timestamp < ?, ?, timestamp)',
          [key, value, timestamp, timestamp, value, timestamp, timestamp]
        );
      } catch (error) {
        console.error(`Error repairing inconsistency on ${node}:`, error);
        // Could implement more sophisticated retry logic here
      }
    }
  }
}
```

2. **Versioned Data (Vector Clocks)**: Using vector clocks to track causality between distributed updates.

```
// Vector clock implementation
class VectorClock {
  constructor(nodeIds) {
    this.clock = {};

    // Initialize clock for all nodes
    for (const nodeId of nodeIds) {
      this.clock[nodeId] = 0;
    }
  }

  // Increment counter for a node
  increment(nodeId) {
    if (!(nodeId in this.clock)) {
      this.clock[nodeId] = 0;
    }

    this.clock[nodeId]++;
    return this.clone();
  }

  // Merge with another vector clock
  merge(otherClock) {
    const result = this.clone();

    // Take maximum value for each node
    for (const nodeId in otherClock.clock) {
      if (!(nodeId in result.clock)) {
        result.clock[nodeId] = 0;
      }

      result.clock[nodeId] = Math.max(
        result.clock[nodeId],
        otherClock.clock[nodeId]
      );
    }

    return result;
  }

  // Compare vector clocks
  compare(otherClock) {
    let isGreater = false;
    let isLess = false;

    // Check all nodes in this clock
    for (const nodeId in this.clock) {
      if (!(nodeId in otherClock.clock)) {
        isGreater = true;
        continue;
      }

      if (this.clock[nodeId] > otherClock.clock[nodeId]) {
        isGreater = true;
```

```
      } else if (this.clock[nodeId] < otherClock.clock[nodeId]) {
        isLess = true;
      }
    }

    // Check for nodes in other clock but not in this one
    for (const nodeId in otherClock.clock) {
      if (!(nodeId in this.clock) && otherClock.clock[nodeId] > 0) {
        isLess = true;
      }
    }

    if (isGreater && isLess) {
      return 'CONFLICT'; // Concurrent updates
    } else if (isGreater) {
      return 'GREATER'; // This is newer
    } else if (isLess) {
      return 'LESS'; // Other is newer
    } else {
      return 'EQUAL'; // Same version
    }
  }

  // Create a copy of this vector clock
  clone() {
    const copy = new VectorClock([]);
    copy.clock = { ...this.clock };
    return copy;
  }

  // Convert to string representation
  toString() {
    return JSON.stringify(this.clock);
  }

  // Create from string representation
  static fromString(str) {
    const parsed = JSON.parse(str);
    const clock = new VectorClock([]);
    clock.clock = parsed;
    return clock;
  }
}

// Database with vector clock versioning
class VectorClockDatabase {
  constructor(nodeId, nodes) {
    this.nodeId = nodeId;
    this.nodes = nodes;
    this.store = new Map();
  }

  // Get value for key
  async get(key) {
    const entry = this.store.get(key);
```

```
    if (!entry) {
      return { value: null, exists: false };
    }

    return {
      value: entry.value,
      vectorClock: entry.vectorClock,
      exists: true,
    };
  }

  // Put value with vector clock
  async put(key, value, clientVectorClock = null) {
    // Get current value and its vector clock
    const current = await this.get(key);

    // Initialize a new vector clock if no client clock provided
    let newVectorClock;

    if (!clientVectorClock) {
      if (current.exists) {
        // Increment existing clock
        newVectorClock = current.vectorClock.clone().increment(this.nodeId);
      } else {
        // Create new clock
        newVectorClock = new VectorClock(this.nodes).increment(this.nodeId);
      }
    } else {
      if (current.exists) {
        // Compare clocks to detect conflicts
        const comparison = clientVectorClock.compare(current.vectorClock);

        if (comparison === 'LESS') {
          throw new Error('Stale write attempt with outdated vector clock');
        } else if (comparison === 'CONFLICT') {
          // Conflicting versions, need reconciliation
          // In a real system, this might trigger conflict resolution
          console.warn('Vector clock conflict detected, forcing merge');
        }
      }

      // Increment client clock
      newVectorClock = clientVectorClock.clone().increment(this.nodeId);
    }

    // Store value with new vector clock
    this.store.set(key, {
      value,
      vectorClock: newVectorClock,
    });

    return newVectorClock;
  }
```

```
  // Merge remote data (for anti-entropy)
  async merge(key, remoteValue, remoteVectorClock) {
    const current = await this.get(key);

    if (!current.exists) {
      // We don't have this value, accept remote version
      this.store.set(key, {
        value: remoteValue,
        vectorClock: remoteVectorClock,
      });
      return remoteVectorClock;
    }

    // Compare vector clocks
    const comparison = current.vectorClock.compare(remoteVectorClock);

    if (comparison === 'LESS') {
      // Remote version is newer, accept it
      this.store.set(key, {
        value: remoteValue,
        vectorClock: remoteVectorClock,
      });
      return remoteVectorClock;
    } else if (comparison === 'GREATER') {
      // Our version is newer, keep it
      return current.vectorClock;
    } else if (comparison === 'CONFLICT') {
      // Conflicting versions, need reconciliation
      // This is application-specific, but here's a simple approach
      const mergedValue = this.resolveConflict(current.value, remoteValue);

      // Create merged vector clock
      const mergedClock = current.vectorClock.merge(remoteVectorClock);

      // Store merged version
      this.store.set(key, {
        value: mergedValue,
        vectorClock: mergedClock,
      });

      return mergedClock;
    }

    // Clocks are equal, no change needed
    return current.vectorClock;
  }

  // Application-specific conflict resolution
  resolveConflict(localValue, remoteValue) {
    // Example strategy: merge JSON objects
    if (typeof localValue === 'object' && typeof remoteValue === 'object') {
      return { ...localValue, ...remoteValue };
    }

    // Default strategy: use most recent timestamp if available
```

```
    if (localValue?.timestamp && remoteValue?.timestamp) {
      return localValue.timestamp > remoteValue.timestamp
        ? localValue
        : remoteValue;
    }

    // Last resort: keep local value
    return localValue;
  }
}
```

6.2 Database Migration and Version Control

Managing changes to database schemas and data over time is crucial for application evolution.

Database Schema Migration

1. **Migration Framework**: A system for tracking and applying schema changes.

```
// Database migration manager
class MigrationManager {
  constructor(db, migrationTableName = 'schema_migrations') {
    this.db = db;
    this.migrationTableName = migrationTableName;
  }

  // Initialize migration table
  async initialize() {
    // Create migration table if it doesn't exist
    await this.db.execute(`
      CREATE TABLE IF NOT EXISTS ${this.migrationTableName} (
        version VARCHAR(50) PRIMARY KEY,
        applied_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        description TEXT
      )
    `);
  }

  // Get applied migrations
  async getAppliedMigrations() {
    const result = await this.db.execute(
      `SELECT version FROM ${this.migrationTableName} ORDER BY version`
    );

    return result.rows.map((row) => row.version);
  }

  // Apply a single migration
  async applyMigration(migration) {
    // Start transaction
    await this.db.execute('BEGIN');

    try {
```

```
        console.log(
          `Applying migration ${migration.version}: ${migration.description}`
        );

        // Execute up migration
        if (typeof migration.up === 'function') {
          await migration.up(this.db);
        } else {
          await this.db.execute(migration.up);
        }

        // Record migration
        await this.db.execute(
          `INSERT INTO ${this.migrationTableName} (version, description) VALUES (?,
?)`,
          [migration.version, migration.description]
        );

        // Commit transaction
        await this.db.execute('COMMIT');
        console.log(`Migration ${migration.version} applied successfully.`);
      } catch (error) {
        // Rollback on error
        await this.db.execute('ROLLBACK');
        console.error(`Migration ${migration.version} failed:`, error);
        throw error;
      }
    }

    // Revert a single migration
    async revertMigration(migration) {
      // Start transaction
      await this.db.execute('BEGIN');

      try {
        console.log(
          `Reverting migration ${migration.version}: ${migration.description}`
        );

        // Execute down migration
        if (typeof migration.down === 'function') {
          await migration.down(this.db);
        } else {
          await this.db.execute(migration.down);
        }

        // Remove migration record
        await this.db.execute(
          `DELETE FROM ${this.migrationTableName} WHERE version = ?`,
          [migration.version]
        );

        // Commit transaction
        await this.db.execute('COMMIT');
        console.log(`Migration ${migration.version} reverted successfully.`);
```

```
      } catch (error) {
        // Rollback on error
        await this.db.execute('ROLLBACK');
        console.error(`Reverting migration ${migration.version} failed:`, error);
        throw error;
      }
    }

    // Migrate database to target version
    async migrate(migrations, targetVersion = null) {
      await this.initialize();

      // Sort migrations by version
      migrations.sort((a, b) => a.version.localeCompare(b.version));

      // Get applied migrations
      const appliedVersions = await this.getAppliedMigrations();

      // Determine migrations to apply
      let migrationsToApply = migrations.filter(
        (m) => !appliedVersions.includes(m.version)
      );

      // Limit to target version if specified
      if (targetVersion !== null) {
        migrationsToApply = migrationsToApply.filter(
          (m) => m.version <= targetVersion
        );
      }

      // Apply migrations in order
      for (const migration of migrationsToApply) {
        await this.applyMigration(migration);
      }

      return migrationsToApply.length;
    }

    // Rollback to specific version
    async rollback(migrations, targetVersion) {
      await this.initialize();

      // Sort migrations by version in reverse order
      migrations.sort((a, b) => b.version.localeCompare(a.version));

      // Get applied migrations
      const appliedVersions = await this.getAppliedMigrations();

      // Determine migrations to revert
      const migrationsToRevert = migrations.filter(
        (m) => appliedVersions.includes(m.version) && m.version > targetVersion
      );

      // Revert migrations in reverse order
      for (const migration of migrationsToRevert) {
```

```
      await this.revertMigration(migration);
    }

    return migrationsToRevert.length;
  }
}
```

2. **Migration Files**: Structured files containing schema changes and rollback commands.

```
// Example migration file
const migration_20230615123456 = {
  version: '20230615123456',
  description: 'Create users table',

  // Up migration
  up: async (db) => {
    await db.execute(`
      CREATE TABLE users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(50) NOT NULL UNIQUE,
        email VARCHAR(100) NOT NULL UNIQUE,
        password_hash VARCHAR(100) NOT NULL,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
        INDEX idx_users_email (email)
      )
    `);
  },

  // Down migration
  down: async (db) => {
    await db.execute('DROP TABLE users');
  },
};

// Another migration file
const migration_20230616123456 = {
  version: '20230616123456',
  description: 'Add user profile fields',

  // Up migration
  up: async (db) => {
    await db.execute(`
      ALTER TABLE users
      ADD COLUMN first_name VARCHAR(50),
      ADD COLUMN last_name VARCHAR(50),
      ADD COLUMN bio TEXT,
      ADD COLUMN profile_image_url VARCHAR(255)
    `);
  },

  // Down migration
  down: async (db) => {
```

```
    await db.execute(`
      ALTER TABLE users
      DROP COLUMN first_name,
      DROP COLUMN last_name,
      DROP COLUMN bio,
      DROP COLUMN profile_image_url
    `);
  },
};

// Example migration runner
async function runMigrations() {
  const db = await createDatabaseConnection();
  const migrationManager = new MigrationManager(db);

  const migrations = [
    migration_20230615123456,
    migration_20230616123456,
    // Add more migrations here
  ];

  try {
    const appliedCount = await migrationManager.migrate(migrations);
    console.log(`Applied ${appliedCount} migrations successfully.`);
  } catch (error) {
    console.error('Migration failed:', error);
  } finally {
    await db.close();
  }
}
```

3. **Zero-Downtime Migrations**: Techniques for updating schemas without service interruption.

```
// Example of a zero-downtime migration approach
const zeroDowntimeMigration = {
  version: '20230617123456',
  description: 'Add email verification status with zero downtime',

  // Multi-stage migration
  up: async (db) => {
    // Stage 1: Add new column (doesn't affect existing queries)
    await db.execute(`
      ALTER TABLE users
      ADD COLUMN email_verified BOOLEAN
    `);

    // Stage 2: Fill with default values (fast operation with reasonable default)
    await db.execute(`
      UPDATE users
      SET email_verified = FALSE
      WHERE email_verified IS NULL
    `);
```

```
    // Stage 3: Update application to use the new column (done outside this
migration)
    // This would be done by deploying updated application code that handles the new
field
    // ...

    // Stage 4: Add constraints after application is updated
    // This would be done in a subsequent migration after application deployment
    // ...
  },

  down: async (db) => {
    // Revert in reverse order
    await db.execute(`
      ALTER TABLE users
      DROP COLUMN email_verified
    `);
  },
};

// Example of breaking a large migration into smaller chunks
const chunkMigration = {
  version: '20230618123456',
  description: 'Index historical data in chunks',

  up: async (db) => {
    // Get total rows
    const countResult = await db.execute(
      'SELECT COUNT(*) as total FROM large_table'
    );
    const totalRows = countResult.rows[0].total;

    // Process in chunks of 10,000 rows
    const chunkSize = 10000;
    const totalChunks = Math.ceil(totalRows / chunkSize);

    for (let chunk = 0; chunk < totalChunks; chunk++) {
      const offset = chunk * chunkSize;

      console.log(
        `Processing chunk ${chunk + 1}/${totalChunks} (offset ${offset})`
      );

      // Process chunk with limit and offset
      await db.execute(`
        UPDATE large_table
        SET processed = TRUE
        WHERE processed = FALSE
        LIMIT ${chunkSize}
      `);

      // Optional: Add a small delay to reduce database load
      await new Promise((resolve) => setTimeout(resolve, 100));
    }
```

```
    // Add index after processing
    await db.execute(
      'CREATE INDEX idx_large_table_processed ON large_table(processed)'
    );
  },

  down: async (db) => {
    await db.execute('DROP INDEX idx_large_table_processed ON large_table');
    await db.execute('UPDATE large_table SET processed = FALSE');
  },
};
```

Database Versioning Patterns

1. **Semantic Versioning**: Using semantic versioning for database schemas.

```
// Database versioning with semantic versioning
class DatabaseVersionManager {
  constructor(db) {
    this.db = db;
  }

  // Initialize version table
  async initialize() {
    await this.db.execute(`
      CREATE TABLE IF NOT EXISTS db_version (
        id INT PRIMARY KEY DEFAULT 1,
        major INT NOT NULL,
        minor INT NOT NULL,
        patch INT NOT NULL,
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        CHECK (id = 1) -- Ensure only one row exists
      )
    `);

    // Insert initial version if table is empty
    const versionCheck = await this.db.execute(
      'SELECT COUNT(*) as count FROM db_version'
    );
    if (versionCheck.rows[0].count === 0) {
      await this.db.execute(
        'INSERT INTO db_version (major, minor, patch) VALUES (1, 0, 0)'
      );
    }
  }

  // Get current schema version
  async getCurrentVersion() {
    await this.initialize();

    const result = await this.db.execute(
      'SELECT major, minor, patch FROM db_version WHERE id = 1'
    );
```

```
    const version = result.rows[0];
    return `${version.major}.${version.minor}.${version.patch}`;
  }

  // Update schema version
  async updateVersion(major, minor, patch) {
    await this.initialize();

    await this.db.execute(
      'UPDATE db_version SET major = ?, minor = ?, patch = ?, updated_at =
CURRENT_TIMESTAMP WHERE id = 1',
      [major, minor, patch]
    );

    return `${major}.${minor}.${patch}`;
  }

  // Increment major version (breaking changes)
  async incrementMajor() {
    const currentVersion = await this.getCurrentVersion();
    const [major, minor, patch] = currentVersion.split('.').map(Number);

    return this.updateVersion(major + 1, 0, 0);
  }

  // Increment minor version (new features, non-breaking)
  async incrementMinor() {
    const currentVersion = await this.getCurrentVersion();
    const [major, minor, patch] = currentVersion.split('.').map(Number);

    return this.updateVersion(major, minor + 1, 0);
  }

  // Increment patch version (bug fixes)
  async incrementPatch() {
    const currentVersion = await this.getCurrentVersion();
    const [major, minor, patch] = currentVersion.split('.').map(Number);

    return this.updateVersion(major, minor, patch + 1);
  }
}
```

2. **Change Tracking**: Recording all schema changes for audit and rollback.

```
// Schema change tracking system
class SchemaChangeTracker {
  constructor(db) {
    this.db = db;
  }

  // Initialize tracking
  async initialize() {
```

```
    await this.db.execute(`
      CREATE TABLE IF NOT EXISTS schema_changes (
        id INT AUTO_INCREMENT PRIMARY KEY,
        change_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        user_id VARCHAR(50) NOT NULL,
        description TEXT NOT NULL,
        sql_commands TEXT NOT NULL,
        rollback_commands TEXT,
        environment VARCHAR(20) NOT NULL,
        applied BOOLEAN DEFAULT TRUE,
        version VARCHAR(50)
      )
    `);
  }

  // Record a schema change
  async recordChange(options) {
    const {
      userId,
      description,
      sqlCommands,
      rollbackCommands,
      environment,
      version,
    } = options;

    await this.initialize();

    // Insert the change record
    const result = await this.db.execute(
      `INSERT INTO schema_changes
       (user_id, description, sql_commands, rollback_commands, environment, version)
       VALUES (?, ?, ?, ?, ?, ?)`,
      [userId, description, sqlCommands, rollbackCommands, environment, version]
    );

    return result.insertId;
  }

  // Get change history
  async getChangeHistory(options = {}) {
    const {
      limit = 100,
      offset = 0,
      environment = null,
      startDate = null,
      endDate = null,
      version = null,
    } = options;

    let query = 'SELECT * FROM schema_changes WHERE 1=1';
    const params = [];

    if (environment) {
      query += ' AND environment = ?';
```

```
      params.push(environment);
    }

    if (startDate) {
      query += ' AND change_date >= ?';
      params.push(startDate);
    }

    if (endDate) {
      query += ' AND change_date <= ?';
      params.push(endDate);
    }

    if (version) {
      query += ' AND version = ?';
      params.push(version);
    }

    query += ' ORDER BY change_date DESC LIMIT ? OFFSET ?';
    params.push(limit, offset);

    const result = await this.db.execute(query, params);
    return result.rows;
  }

  // Get rollback script for a specific change
  async getRollbackScript(changeId) {
    const result = await this.db.execute(
      'SELECT rollback_commands FROM schema_changes WHERE id = ?',
      [changeId]
    );

    if (result.rows.length === 0) {
      throw new Error(`Change ID ${changeId} not found`);
    }

    return result.rows[0].rollback_commands;
  }
}
```

3. **Feature Flags for Database Changes**: Using feature flags to gradually roll out schema changes.

```
// Database feature flag system
class DatabaseFeatureFlags {
  constructor(db) {
    this.db = db;
    this.cache = new Map();
    this.cacheTTL = 60000; // 1 minute cache TTL
  }

  // Initialize feature flags table
  async initialize() {
    await this.db.execute(`
```

```
      CREATE TABLE IF NOT EXISTS feature_flags (
        flag_name VARCHAR(100) PRIMARY KEY,
        enabled BOOLEAN NOT NULL DEFAULT FALSE,
        description TEXT,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP
      )
    `);
  }

  // Create a new feature flag
  async createFlag(flagName, description, enabled = false) {
    await this.initialize();

    await this.db.execute(
      `INSERT INTO feature_flags (flag_name, description, enabled)
       VALUES (?, ?, ?)
       ON DUPLICATE KEY UPDATE
       description = VALUES(description)`,
      [flagName, description, enabled]
    );

    // Update cache
    this.cache.set(flagName, {
      enabled,
      timestamp: Date.now(),
    });
  }

  // Check if a feature flag is enabled
  async isEnabled(flagName) {
    // Check cache first
    if (this.cache.has(flagName)) {
      const cachedValue = this.cache.get(flagName);
      if (Date.now() - cachedValue.timestamp < this.cacheTTL) {
        return cachedValue.enabled;
      }

      // Cache expired, remove it
      this.cache.delete(flagName);
    }

    await this.initialize();

    // Query database
    const result = await this.db.execute(
      'SELECT enabled FROM feature_flags WHERE flag_name = ?',
      [flagName]
    );

    let enabled = false;

    if (result.rows.length > 0) {
      enabled = result.rows[0].enabled;
```

```
      // Update cache
      this.cache.set(flagName, {
        enabled,
        timestamp: Date.now(),
      });
    }

    return enabled;
  }

  // Enable a feature flag
  async enableFlag(flagName) {
    await this.initialize();

    const result = await this.db.execute(
      'UPDATE feature_flags SET enabled = TRUE WHERE flag_name = ?',
      [flagName]
    );

    if (result.affectedRows === 0) {
      throw new Error(`Feature flag '${flagName}' not found`);
    }

    // Update cache
    this.cache.set(flagName, {
      enabled: true,
      timestamp: Date.now(),
    });
  }

  // Disable a feature flag
  async disableFlag(flagName) {
    await this.initialize();

    const result = await this.db.execute(
      'UPDATE feature_flags SET enabled = FALSE WHERE flag_name = ?',
      [flagName]
    );

    if (result.affectedRows === 0) {
      throw new Error(`Feature flag '${flagName}' not found`);
    }

    // Update cache
    this.cache.set(flagName, {
      enabled: false,
      timestamp: Date.now(),
    });
  }

  // Get all feature flags
  async getAllFlags() {
    await this.initialize();

    const result = await this.db.execute(
```

```
      'SELECT * FROM feature_flags ORDER BY flag_name'
    );

    return result.rows;
  }
}

// Example: using feature flags for gradual schema changes
async function userQueryWithFeatureFlag(db, featureFlags, userId) {
  // Check if new schema is enabled
  const useNewSchema = await featureFlags.isEnabled('new_user_schema');

  let query;

  if (useNewSchema) {
    // Use new schema with additional fields
    query = `
      SELECT id, username, email, first_name, last_name, bio, created_at
      FROM users
      WHERE id = ?
    `;
  } else {
    // Use old schema
    query = `
      SELECT id, username, email, created_at
      FROM users
      WHERE id = ?
    `;
  }

  const result = await db.execute(query, [userId]);
  return result.rows[0];
}
```

6.3 Data Warehousing Concepts

Data warehousing involves storing large volumes of structured data optimized for analysis and reporting.

Data Warehouse Architecture

1. **Traditional Architecture**: ETL (Extract, Transform, Load) process feeding a centralized warehouse.

```
// Simple ETL process implementation
class ETLProcessor {
  constructor(sourceDb, warehouseDb) {
    this.sourceDb = sourceDb;
    this.warehouseDb = warehouseDb;
  }

  // Run extraction process
  async extract(sourceTable, query, params = []) {
    console.log(`Extracting data from ${sourceTable}...`);
```

```
    let extractedData;

    if (query) {
      // Use custom query
      const result = await this.sourceDb.execute(query, params);
      extractedData = result.rows;
    } else {
      // Extract all data from table
      const result = await this.sourceDb.execute(
        `SELECT * FROM ${sourceTable}`
      );
      extractedData = result.rows;
    }

    console.log(`Extracted ${extractedData.length} rows from ${sourceTable}`);
    return extractedData;
  }

  // Run transformation process
  async transform(data, transformationFn) {
    console.log('Transforming data...');

    if (typeof transformationFn !== 'function') {
      // No transformation, return data as is
      return data;
    }

    // Apply transformation function to each record
    const transformedData = [];

    for (const record of data) {
      const transformed = await transformationFn(record);
      if (transformed) {
        transformedData.push(transformed);
      }
    }

    console.log(`Transformed ${transformedData.length} records`);
    return transformedData;
  }

  // Load data into warehouse
  async load(targetTable, data, options = {}) {
    const {
      batchSize = 1000,
      truncateFirst = false,
      updateOnDuplicate = false,
    } = options;

    console.log(`Loading data into ${targetTable}...`);

    if (data.length === 0) {
      console.log('No data to load.');
      return 0;
    }
```

```
    // Get column names from first record
    const columns = Object.keys(data[0]);

    // Truncate target table if requested
    if (truncateFirst) {
      await this.warehouseDb.execute(`TRUNCATE TABLE ${targetTable}`);
      console.log(`Truncated table ${targetTable}`);
    }

    // Process in batches
    let totalLoaded = 0;

    for (let i = 0; i < data.length; i += batchSize) {
      const batch = data.slice(i, i + batchSize);

      // Create placeholders for prepared statement
      const placeholders = batch
        .map(() => `(${columns.map(() => '?').join(', ')})`)
        .join(', ');

      // Flatten values for prepared statement
      const values = batch.flatMap((record) =>
        columns.map((col) => record[col])
      );

      let query = `INSERT INTO ${targetTable} (${columns.join(
        ', '
      )}) VALUES ${placeholders}`;

      // Add ON DUPLICATE KEY UPDATE clause if requested
      if (updateOnDuplicate) {
        const updates = columns
          .map((col) => `${col} = VALUES(${col})`)
          .join(', ');

        query += ` ON DUPLICATE KEY UPDATE ${updates}`;
      }

      // Execute insert
      const result = await this.warehouseDb.execute(query, values);
      totalLoaded += result.affectedRows;

      console.log(`Loaded batch of ${batch.length} records`);
    }

    console.log(`Total loaded: ${totalLoaded} records into ${targetTable}`);
    return totalLoaded;
  }

  // Run complete ETL process
  async runETL(sourceTable, targetTable, options = {}) {
    const {
      extractQuery = null,
      extractParams = [],
```

```
      transformFn = null,
      loadOptions = {},
    } = options;

    console.log(`Starting ETL process: ${sourceTable} -> ${targetTable}`);

    try {
      // Extract
      const extractedData = await this.extract(
        sourceTable,
        extractQuery,
        extractParams
      );

      // Transform
      const transformedData = await this.transform(extractedData, transformFn);

      // Load
      const loadedCount = await this.load(
        targetTable,
        transformedData,
        loadOptions
      );

      console.log(
        `ETL process completed successfully. Loaded ${loadedCount} records.`
      );
      return {
        success: true,
        extractedCount: extractedData.length,
        transformedCount: transformedData.length,
        loadedCount,
      };
    } catch (error) {
      console.error('ETL process failed:', error);
      return {
        success: false,
        error: error.message,
      };
    }
  }
}
```

2. **Modern Data Lake Architecture**: A more flexible approach using data lakes and ELT (Extract, Load, Transform).

```
// Data Lake ELT implementation
class DataLakeProcessor {
  constructor(sourceDb, dataLake, queryEngine) {
    this.sourceDb = sourceDb;        // Source database connection
    this.dataLake = dataLake;        // Data lake storage (e.g., S3, HDFS)
    this.queryEngine = queryEngine;  // Query engine (e.g., Athena, Presto)
  }
```

```
    // Extract data and load directly to data lake (raw zone)
    async extractAndLoad(sourceTable, targetPath, options = {}) {
      const {
        format = 'parquet',
        partitionBy = null,
        extractQuery = `SELECT * FROM ${sourceTable}`,
        extractParams = []
      } = options;

      console.log(`Extracting data from ${sourceTable} to ${targetPath}...`);

      try {
        // Extract data from source
        const result = await this.sourceDb.execute(extractQuery, extractParams);
        const data = result.rows;

        console.log(`Extracted ${data.length} rows from ${sourceTable}`);

        // Determine path with partitioning if needed
        let finalPath = targetPath;

        if (partitionBy && data.length > 0) {
          // Apply partitioning logic
          const partitions = this.createPartitions(data, partitionBy);

          // Upload each partition
          for (const [partitionPath, partitionData] of Object.entries(partitions)) {
            const partitionFilePath = `${targetPath}/${partitionPath}`;
            await this.uploadToDataLake(partitionData, partitionFilePath, format);
          }
        } else {
          // Upload all data to target path
          await this.uploadToDataLake(data, finalPath, format);
        }

        console.log(`Successfully loaded data to ${targetPath}`);

        return {
          success: true,
          count: data.length,
          path: targetPath
        };
      } catch (error) {
        console.error(`Extract and load failed:`, error);
        return {
          success: false,
          error: error.message
        };
      }
    }

    // Helper to create partitions based on partition keys
    createPartitions(data, partitionBy) {
      const partitions = {};
```

```
    // Group data by partition keys
    for (const record of data) {
      let partitionPath = '';

      if (typeof partitionBy === 'string') {
        // Single partition key
        const value = record[partitionBy];
        partitionPath = `${partitionBy}=${value}`;
      } else if (Array.isArray(partitionBy)) {
        // Multiple partition keys
        partitionPath = partitionBy
          .map(key => `${key}=${record[key]}`)
          .join('/');
      } else {
        throw new Error('Invalid partitionBy value');
      }

      if (!partitions[partitionPath]) {
        partitions[partitionPath] = [];
      }

      partitions[partitionPath].push(record);
    }

    return partitions;
  }

  // Upload data to data lake
  async uploadToDataLake(data, path, format) {
    console.log(`Uploading ${data.length} records to ${path} in ${format} format`);

    // Format conversion logic
    let formattedData;

    switch (format.toLowerCase()) {
      case 'json':
        formattedData = JSON.stringify(data);
        break;
      case 'csv':
        formattedData = this.convertToCSV(data);
        break;
      case 'parquet':
        formattedData = await this.convertToParquet(data);
        break;
      default:
        throw new Error(`Unsupported format: ${format}`);
    }

    // Upload to data lake (implementation depends on the storage system)
    await this.dataLake.uploadFile(path, formattedData, { contentType: format });

    return {
      path,
      count: data.length,
```

```
        format
      };
    }

    // Transform data (create views/tables in refined zone)
    async transform(sqlTransformation, options = {}) {
      const {
        destinationTable,
        description = '',
        materialize = true
      } = options;

      console.log(`Running transformation to create ${destinationTable}`);

      try {
        if (materialize) {
          // Create materialized table
          await this.queryEngine.execute(`
            CREATE OR REPLACE TABLE ${destinationTable} AS
            ${sqlTransformation}
          `);
        } else {
          // Create view
          await this.queryEngine.execute(`
            CREATE OR REPLACE VIEW ${destinationTable} AS
            ${sqlTransformation}
          `);
        }

        // Record transformation metadata
        await this.queryEngine.execute(`
          INSERT INTO data_lake_metadata (
            table_name,
            table_type,
            description,
            transformation_sql,
            created_at
          ) VALUES (?, ?, ?, ?, CURRENT_TIMESTAMP)
          ON DUPLICATE KEY UPDATE
          description = VALUES(description),
          transformation_sql = VALUES(transformation_sql),
          updated_at = CURRENT_TIMESTAMP
        `, [
          destinationTable,
          materialize ? 'TABLE' : 'VIEW',
          description,
          sqlTransformation
        ]);

        console.log(`Successfully created ${materialize ? 'table' : 'view'}:
${destinationTable}`);

        return {
          success: true,
          tableName: destinationTable,
```

```
          type: materialize ? 'TABLE' : 'VIEW'
        };
      } catch (error) {
        console.error(`Transformation failed:`, error);
        return {
          success: false,
          error: error.message
        };
      }
    }

    // Helper to convert data to CSV
    convertToCSV(data) {
    if (data.length === 0) return '';

    const headers = Object.keys(data[0]);
    const headerRow = headers.join(',');

    const rows = data.map(row => {
      return headers.map(field => {
        const value = row[field];

        // Handle different types for CSV
        if (value === null || value === undefined) {
          return '';
        } else if (typeof value === 'string') {
          // Escape quotes and wrap in quotes
          return `"${value.replace(/"/g, '""')}"`;
        } else if (value instanceof Date) {
          return `"${value.toISOString()}"`;
        } else {
          return String(value);
        }
      }).join(',');
    });

    return [headerRow, ...rows].join('\n');
  }

  // Helper to convert data to Parquet (simplified)
  async convertToParquet(data) {
    // In reality, you would use a library like parquetjs, but here's a simplified
  example
    console.log('Converting data to Parquet format');

    // Mock implementation that would normally use the parquetjs library
    // This is just a placeholder for the concept
    const mockParquetConverter = {
      convertToParquet: async (data) => {
        // Pretend we're converting to parquet
        return Buffer.from(JSON.stringify(data));
      }
    };

    return await mockParquetConverter.convertToParquet(data);
```

```
}

// Run entire ELT pipeline
async runELTPipeline(sourceTable, targetDatasetName, transformations) {
  // Step 1: Extract and Load to raw zone
  const rawZonePath = `raw/${sourceTable}/${new Date().toISOString().split('T')
[0]}`;

  const extractResult = await this.extractAndLoad(
    sourceTable,
    rawZonePath,
    {
      format: 'parquet',
      partitionBy: 'created_date'
    }
  );

  if (!extractResult.success) {
    throw new Error(`Extract and load failed: ${extractResult.error}`);
  }

  // Step 2: Create raw zone table/view
  await this.queryEngine.execute(`
    CREATE EXTERNAL TABLE IF NOT EXISTS raw_${sourceTable} (
      -- schema here
    )
    PARTITIONED BY (created_date STRING)
    STORED AS PARQUET
    LOCATION '${rawZonePath}'
  `);

  await this.queryEngine.execute(`MSCK REPAIR TABLE raw_${sourceTable}`);

  // Step 3: Run transformations to create refined zone
  const transformationResults = [];

  for (const transform of transformations) {
    const result = await this.transform(
      transform.sql,
      {
        destinationTable: transform.tableName,
        description: transform.description,
        materialize: transform.materialize
      }
    );

    transformationResults.push(result);

    if (!result.success) {
      console.error(`Transformation failed: ${result.error}`);
    }
  }

  return {
    extractResult,
```

```
    transformationResults
  };
}
```

Star and Snowflake Schemas

1. **Star Schema**: A dimensional model with a central fact table surrounded by dimension tables.

```
-- Example of a Star Schema for an E-commerce Data Warehouse

-- Dimension table: dim_customer
CREATE TABLE dim_customer (
  customer_id INT PRIMARY KEY,
  customer_name VARCHAR(100),
  email VARCHAR(100),
  phone VARCHAR(20),
  address TEXT,
  city VARCHAR(50),
  state VARCHAR(50),
  zip_code VARCHAR(20),
  country VARCHAR(50),
  customer_since DATE,
  is_active BOOLEAN,
  segment VARCHAR(20)
);

-- Dimension table: dim_product
CREATE TABLE dim_product (
  product_id INT PRIMARY KEY,
  product_name VARCHAR(100),
  description TEXT,
  category VARCHAR(50),
  subcategory VARCHAR(50),
  brand VARCHAR(50),
  supplier VARCHAR(100),
  cost DECIMAL(10,2),
  base_price DECIMAL(10,2),
  is_active BOOLEAN
);

-- Dimension table: dim_store
CREATE TABLE dim_store (
  store_id INT PRIMARY KEY,
  store_name VARCHAR(100),
  address TEXT,
  city VARCHAR(50),
  state VARCHAR(50),
  zip_code VARCHAR(20),
  country VARCHAR(50),
  phone VARCHAR(20),
  manager_name VARCHAR(100),
  open_date DATE,
  store_type VARCHAR(50),
```

```
  store_size INT
);

-- Dimension table: dim_date
CREATE TABLE dim_date (
  date_id INT PRIMARY KEY,  -- Could be a surrogate key or date in integer format
  full_date DATE,
  day_of_week INT,
  day_name VARCHAR(10),
  day_of_month INT,
  day_of_year INT,
  week_of_year INT,
  month INT,
  month_name VARCHAR(10),
  quarter INT,
  year INT,
  is_weekend BOOLEAN,
  is_holiday BOOLEAN,
  holiday_name VARCHAR(50)
);

-- Fact table: fact_sales
CREATE TABLE fact_sales (
  sale_id INT PRIMARY KEY,
  customer_id INT,
  product_id INT,
  store_id INT,
  date_id INT,
  quantity INT,
  unit_price DECIMAL(10,2),
  discount_amount DECIMAL(10,2),
  sales_amount DECIMAL(10,2),  -- unit_price * quantity - discount
  cost_amount DECIMAL(10,2),
  profit_amount DECIMAL(10,2), -- sales_amount - cost_amount

  -- Foreign keys
  FOREIGN KEY (customer_id) REFERENCES dim_customer(customer_id),
  FOREIGN KEY (product_id) REFERENCES dim_product(product_id),
  FOREIGN KEY (store_id) REFERENCES dim_store(store_id),
  FOREIGN KEY (date_id) REFERENCES dim_date(date_id)
);

-- Example query: Monthly sales by product category
SELECT
  d.year,
  d.month_name,
  p.category,
  SUM(f.sales_amount) as total_sales,
  SUM(f.profit_amount) as total_profit
FROM fact_sales f
JOIN dim_date d ON f.date_id = d.date_id
JOIN dim_product p ON f.product_id = p.product_id
WHERE d.year = 2023
GROUP BY d.year, d.month_name, p.category
ORDER BY d.month, p.category;
```

2. **Snowflake Schema**: A dimensional model where dimension tables are normalized into multiple related tables.

```
-- Example of a Snowflake Schema for an E-commerce Data Warehouse

-- Dimension tables for normalized Customer dimension
CREATE TABLE dim_geography (
  geography_id INT PRIMARY KEY,
  city VARCHAR(50),
  state VARCHAR(50),
  zip_code VARCHAR(20),
  country VARCHAR(50),
  region VARCHAR(50)
);

CREATE TABLE dim_customer_segment (
  segment_id INT PRIMARY KEY,
  segment_name VARCHAR(50),
  segment_description TEXT
);

CREATE TABLE dim_customer (
  customer_id INT PRIMARY KEY,
  customer_name VARCHAR(100),
  email VARCHAR(100),
  phone VARCHAR(20),
  address TEXT,
  geography_id INT,
  segment_id INT,
  customer_since DATE,
  is_active BOOLEAN,

  FOREIGN KEY (geography_id) REFERENCES dim_geography(geography_id),
  FOREIGN KEY (segment_id) REFERENCES dim_customer_segment(segment_id)
);

-- Dimension tables for normalized Product dimension
CREATE TABLE dim_supplier (
  supplier_id INT PRIMARY KEY,
  supplier_name VARCHAR(100),
  contact_name VARCHAR(100),
  phone VARCHAR(20),
  email VARCHAR(100),
  address TEXT
);

CREATE TABLE dim_category (
  category_id INT PRIMARY KEY,
  category_name VARCHAR(50),
  parent_category_id INT,

  FOREIGN KEY (parent_category_id) REFERENCES dim_category(category_id)
);
```

```sql
CREATE TABLE dim_brand (
  brand_id INT PRIMARY KEY,
  brand_name VARCHAR(50),
  manufacturer VARCHAR(100)
);

CREATE TABLE dim_product (
  product_id INT PRIMARY KEY,
  product_name VARCHAR(100),
  description TEXT,
  category_id INT,
  brand_id INT,
  supplier_id INT,
  cost DECIMAL(10,2),
  base_price DECIMAL(10,2),
  is_active BOOLEAN,

  FOREIGN KEY (category_id) REFERENCES dim_category(category_id),
  FOREIGN KEY (brand_id) REFERENCES dim_brand(brand_id),
  FOREIGN KEY (supplier_id) REFERENCES dim_supplier(supplier_id)
);

-- The rest of the snowflake schema follows the same pattern
-- Now the fact table references the primary dimension tables

-- Fact table
CREATE TABLE fact_sales (
  sale_id INT PRIMARY KEY,
  customer_id INT,
  product_id INT,
  store_id INT,
  date_id INT,
  quantity INT,
  unit_price DECIMAL(10,2),
  discount_amount DECIMAL(10,2),
  sales_amount DECIMAL(10,2),
  cost_amount DECIMAL(10,2),
  profit_amount DECIMAL(10,2),

  FOREIGN KEY (customer_id) REFERENCES dim_customer(customer_id),
  FOREIGN KEY (product_id) REFERENCES dim_product(product_id),
  FOREIGN KEY (store_id) REFERENCES dim_store(store_id),
  FOREIGN KEY (date_id) REFERENCES dim_date(date_id)
);

-- Example query: Sales by category and brand
SELECT
  c.category_name,
  b.brand_name,
  SUM(f.sales_amount) as total_sales,
  SUM(f.profit_amount) as total_profit
FROM fact_sales f
JOIN dim_product p ON f.product_id = p.product_id
JOIN dim_category c ON p.category_id = c.category_id
```

```
JOIN dim_brand b ON p.brand_id = b.brand_id
JOIN dim_date d ON f.date_id = d.date_id
WHERE d.year = 2023
GROUP BY c.category_name, b.brand_name
ORDER BY total_sales DESC;
```

Data Modeling Concepts

1. **Dimensional Modeling**: Design approach focusing on facts (measures) and dimensions (context).

```
// Dimensional modeling helper for generating star schema
class DimensionalModelGenerator {
  constructor(db) {
    this.db = db;
  }

  // Create a date dimension table
  async createDateDimension(startDate, endDate, tableName = 'dim_date') {
    console.log(`Creating date dimension from ${startDate} to ${endDate}`);

    // Create table structure
    await this.db.execute(`
      CREATE TABLE IF NOT EXISTS ${tableName} (
        date_id INT PRIMARY KEY,
        full_date DATE UNIQUE,
        day_of_week INT,
        day_name VARCHAR(10),
        day_of_month INT,
        day_of_year INT,
        week_of_year INT,
        month INT,
        month_name VARCHAR(10),
        quarter INT,
        year INT,
        is_weekend BOOLEAN,
        is_holiday BOOLEAN,
        holiday_name VARCHAR(50)
      )
    `);

    // Convert string dates to Date objects if needed
    const start =
      typeof startDate === 'string' ? new Date(startDate) : startDate;
    const end = typeof endDate === 'string' ? new Date(endDate) : endDate;

    // Generate date records
    let currentDate = new Date(start);
    const batchSize = 1000;
    let batch = [];

    while (currentDate <= end) {
      const date = new Date(currentDate);
```

```
        // Calculate date properties
        const year = date.getFullYear();
        const month = date.getMonth() + 1;
        const day = date.getDate();
        const dayOfWeek = date.getDay();

        // Generate date_id in format YYYYMMDD
        const dateId = year * 10000 + month * 100 + day;

        // Create date record
        const dateRecord = {
          date_id: dateId,
          full_date: date.toISOString().split('T')[0],
          day_of_week: dayOfWeek,
          day_name: [
            'Sunday',
            'Monday',
            'Tuesday',
            'Wednesday',
            'Thursday',
            'Friday',
            'Saturday',
          ][dayOfWeek],
          day_of_month: day,
          day_of_year: this.getDayOfYear(date),
          week_of_year: this.getWeekOfYear(date),
          month: month,
          month_name: [
            'January',
            'February',
            'March',
            'April',
            'May',
            'June',
            'July',
            'August',
            'September',
            'October',
            'November',
            'December',
          ][month - 1],
          quarter: Math.ceil(month / 3),
          year: year,
          is_weekend: dayOfWeek === 0 || dayOfWeek === 6,
          is_holiday: false, // Would need holiday calculation logic
          holiday_name: null,
        };

        batch.push(dateRecord);

        // Process batch if it reaches batch size
        if (batch.length >= batchSize) {
          await this.insertDateBatch(tableName, batch);
          batch = [];
        }
```

```
    // Move to next day
    currentDate.setDate(currentDate.getDate() + 1);
  }

  // Insert any remaining records
  if (batch.length > 0) {
    await this.insertDateBatch(tableName, batch);
  }

  console.log(`Date dimension created successfully in table ${tableName}`);
}

// Helper to insert a batch of date records
async insertDateBatch(tableName, records) {
  if (records.length === 0) return;

  // Generate placeholder values
  const fields = Object.keys(records[0]);
  const placeholders = records
    .map(() => `(${fields.map(() => '?').join(', ')})`)
    .join(', ');

  // Flatten all values for the prepared statement
  const values = records.flatMap((record) =>
    fields.map((field) => record[field])
  );

  // Execute batch insert
  await this.db.execute(
    `
    INSERT INTO ${tableName} (${fields.join(', ')})
    VALUES ${placeholders}
    ON DUPLICATE KEY UPDATE
      full_date = VALUES(full_date)
  `,
    values
  );

  console.log(`Inserted ${records.length} date records`);
}

// Helper to get day of year (1-366)
getDayOfYear(date) {
  const start = new Date(date.getFullYear(), 0, 0);
  const diff = date - start;
  const oneDay = 1000 * 60 * 60 * 24;
  return Math.floor(diff / oneDay);
}

// Helper to get week of year (1-53)
getWeekOfYear(date) {
  const firstDayOfYear = new Date(date.getFullYear(), 0, 1);
  const pastDaysOfYear = (date - firstDayOfYear) / 86400000;
  return Math.ceil((pastDaysOfYear + firstDayOfYear.getDay() + 1) / 7);
```

```
  }

  // Create a slowly changing dimension (SCD) Type 2 table
  async createSCD2Dimension(sourceTable, targetTable, dimensionConfig) {
    const {
      keyField,
      effectiveDateField = 'effective_date',
      expiryDateField = 'expiry_date',
      currentFlagField = 'is_current',
      versionField = 'version',
      trackFields,
      surrogatePrimaryKey = `${targetTable}_id`,
    } = dimensionConfig;

    console.log(`Creating SCD Type 2 dimension table ${targetTable}`);

    // Get schema from source
    const schemaResult = await this.db.execute(`DESCRIBE ${sourceTable}`);
    const sourceSchema = schemaResult.rows;

    // Create target table with SCD2 fields
    let createTableSQL = `
      CREATE TABLE IF NOT EXISTS ${targetTable} (
        ${surrogatePrimaryKey} INT AUTO_INCREMENT PRIMARY KEY,
        ${sourceSchema
          .map((field) => `${field.Field} ${field.Type}`)
          .join(',\n        ')},
        ${effectiveDateField} DATETIME NOT NULL,
        ${expiryDateField} DATETIME,
        ${currentFlagField} BOOLEAN NOT NULL DEFAULT TRUE,
        ${versionField} INT NOT NULL DEFAULT 1
      )
    `;

    await this.db.execute(createTableSQL);

    // Create unique index on business key and version
    await this.db.execute(`
      CREATE UNIQUE INDEX idx_${targetTable}_version
      ON ${targetTable} (${keyField}, ${versionField})
    `);

    console.log(
      `Created SCD Type 2 dimension table structure for ${targetTable}`
    );

    return {
      tableName: targetTable,
      keyField,
      trackFields,
      surrogateKey: surrogatePrimaryKey,
      effectiveDateField,
      expiryDateField,
      currentFlagField,
      versionField,
```

```
    };
  }

  // Update SCD Type 2 dimension with new source data
  async updateSCD2Dimension(sourceData, dimensionConfig) {
    const {
      tableName,
      keyField,
      trackFields,
      surrogateKey,
      effectiveDateField,
      expiryDateField,
      currentFlagField,
      versionField,
    } = dimensionConfig;

    console.log(
      `Updating SCD Type 2 dimension ${tableName} with ${sourceData.length} records`
    );

    for (const sourceRecord of sourceData) {
      // Check if record already exists
      const existingResult = await this.db.execute(
        `
        SELECT * FROM ${tableName}
        WHERE ${keyField} = ? AND ${currentFlagField} = TRUE
      `,
        [sourceRecord[keyField]]
      );

      const now = new Date();
      const existingRecord = existingResult.rows[0];

      if (!existingRecord) {
        // Insert as new record
        const fields = Object.keys(sourceRecord);
        fields.push(effectiveDateField);

        const placeholders = fields.map(() => '?').join(', ');
        const values = [...Object.values(sourceRecord), now];

        await this.db.execute(
          `
          INSERT INTO ${tableName} (${fields.join(', ')})
          VALUES (${placeholders})
        `,
          values
        );

        console.log(
          `Inserted new record for ${keyField}=${sourceRecord[keyField]}`
        );
      } else {
        // Check if tracked fields have changed
        const hasChanges = trackFields.some(
```

```
          (field) => sourceRecord[field] !== existingRecord[field]
        );

        if (hasChanges) {
          // Update current record - mark as expired
          await this.db.execute(
            `
            UPDATE ${tableName}
            SET
              ${expiryDateField} = ?,
              ${currentFlagField} = FALSE
            WHERE ${surrogateKey} = ?
          `,
            [now, existingRecord[surrogateKey]]
          );

          // Insert new version
          const fields = Object.keys(sourceRecord);
          fields.push(effectiveDateField);
          fields.push(versionField);

          const placeholders = fields.map(() => '?').join(', ');
          const values = [
            ...Object.values(sourceRecord),
            now,
            existingRecord[versionField] + 1,
          ];

          await this.db.execute(
            `
            INSERT INTO ${tableName} (${fields.join(', ')})
            VALUES (${placeholders})
          `,
            values
          );

          console.log(
            `Updated record for ${keyField}=${
              sourceRecord[keyField]
            } to version ${existingRecord[versionField] + 1}`
          );
        } else {
          console.log(`No changes for ${keyField}=${sourceRecord[keyField]}`);
        }
      }
    }

    console.log(`SCD Type 2 dimension ${tableName} updated successfully`);
  }

  // Generate a fact table from source data
  async createFactTable(factTableConfig) {
    const {
      tableName,
      sourceTable,
```

```
        dimensions,
        measures,
        incrementalField = null,
      } = factTableConfig;

      console.log(`Creating fact table ${tableName}`);

      // Create table structure
      let createTableSQL = `
        CREATE TABLE IF NOT EXISTS ${tableName} (
          ${tableName}_id INT AUTO_INCREMENT PRIMARY KEY,
          ${dimensions.map((dim) => `${dim.factField} INT`).join(',\n        ')},
          ${measures
            .map(
              (measure) => `${measure.field} ${measure.type || 'DECIMAL(16,4)'}`
            )
            .join(',\n        ')},
          etl_date DATETIME DEFAULT CURRENT_TIMESTAMP,

          ${dimensions
            .map(
              (dim) =>
                `FOREIGN KEY (${dim.factField}) REFERENCES ${dim.dimensionTable}
(${dim.dimensionKey})`
            )
            .join(',\n        ')}
        )
      `;

      await this.db.execute(createTableSQL);

      console.log(`Created fact table structure for ${tableName}`);

      // Create index on dimension keys
      for (const dim of dimensions) {
        await this.db.execute(`
          CREATE INDEX idx_${tableName}_${dim.factField}
          ON ${tableName} (${dim.factField})
        `);
      }

      console.log(`Created indexes for fact table ${tableName}`);

      return {
        tableName,
        dimensions,
        measures,
        incrementalField,
      };
    }

    // Populate fact table from source data
    async populateFactTable(factConfig, options = {}) {
      const { tableName, sourceTable, dimensions, measures, incrementalField } =
        factConfig;
```

```
    const { incrementalValue = null, batchSize = 1000 } = options;

    console.log(`Populating fact table ${tableName}`);

    // Build source query with dimension lookups
    let sourceQuery = `
      SELECT
        ${dimensions
          .map((dim) => {
            if (dim.lookupSQL) {
              return dim.lookupSQL;
            } else {
              return `(SELECT ${dim.dimensionKey} FROM ${dim.dimensionTable}
                    WHERE ${dim.dimensionTable}.${dim.lookupField} =
${sourceTable}.${dim.sourceField}) AS ${dim.factField}`;
            }
          })
          .join(',\n        ')},
        ${measures
          .map((measure) => {
            if (measure.expression) {
              return `${measure.expression} AS ${measure.field}`;
            } else {
              return `${sourceTable}.${measure.field}`;
            }
          })
          .join(',\n        ')}
      FROM ${sourceTable}
    `;

    // Add incremental filter if applicable
    if (incrementalField && incrementalValue) {
      sourceQuery += `
        WHERE ${sourceTable}.${incrementalField} > ?
      `;
    }

    // Execute query to get source data
    const sourceResult = await this.db.execute(
      sourceQuery,
      incrementalField && incrementalValue ? [incrementalValue] : []
    );

    const sourceData = sourceResult.rows;
    console.log(`Retrieved ${sourceData.length} source records for fact table`);

    // Insert data in batches
    for (let i = 0; i < sourceData.length; i += batchSize) {
      const batch = sourceData.slice(i, i + batchSize);

      // Skip records with null dimension keys (failed lookups)
      const validRecords = batch.filter((record) =>
        dimensions.every((dim) => record[dim.factField] !== null)
      );
```

```
      if (validRecords.length === 0) {
        console.log(`No valid records in batch ${i / batchSize + 1}`);
        continue;
      }

      // Generate field list and placeholders
      const fields = [
        ...dimensions.map((dim) => dim.factField),
        ...measures.map((measure) => measure.field),
      ];

      const placeholders = validRecords
        .map(() => `(${fields.map(() => '?').join(', ')})`)
        .join(', ');

      // Flatten values for prepared statement
      const values = validRecords.flatMap((record) =>
        fields.map((field) => record[field])
      );

      // Insert records
      await this.db.execute(
        `
        INSERT INTO ${tableName} (${fields.join(', ')})
        VALUES ${placeholders}
      `,
        values
      );

      console.log(
        `Inserted batch ${i / batchSize + 1} with ${
          validRecords.length
        } records`
      );
    }

    console.log(`Fact table ${tableName} populated successfully`);
  }
}
```

2. **Fact Table Types**: Transaction, periodic snapshot, and accumulating snapshot fact tables.

```
// Example of implementing different fact table types
class FactTableManager {
  constructor(db, dimensionalModel) {
    this.db = db;
    this.dimensionalModel = dimensionalModel;
  }

  // Create transaction fact table
  async createTransactionFactTable(tableName, config) {
    const { dimensions, measures, sourceTable, sourceQuery, incrementalField } =
```

```
    config;

  console.log(`Creating transaction fact table: ${tableName}`);

  // Create table with structure suited for transaction facts
  let createTableSQL = `
    CREATE TABLE IF NOT EXISTS ${tableName} (
      ${tableName}_id INT AUTO_INCREMENT PRIMARY KEY,
      ${dimensions.map((dim) => `${dim.name}_key INT`).join(',\n        ')},
      transaction_date_key INT,
      transaction_time_key INT,  -- Optional, for intraday transactions
      transaction_id VARCHAR(50),  -- Business key from source system
      ${measures
        .map((m) => `${m.name} ${m.type || 'DECIMAL(16,4)'}`)
        .join(',\n        ')},
      etl_date DATETIME DEFAULT CURRENT_TIMESTAMP,

      ${dimensions
        .map(
          (dim) =>
            `FOREIGN KEY (${dim.name}_key) REFERENCES ${dim.tableName}
(${dim.keyField})`
        )
        .join(',\n        ')},
      FOREIGN KEY (transaction_date_key) REFERENCES dim_date(date_id)
      ${
        config.timeGranularity
          ? `,FOREIGN KEY (transaction_time_key) REFERENCES dim_time(time_id)`
          : ''
      }
    )
  `;

  await this.db.execute(createTableSQL);

  // Add appropriate indexes
  await this.db.execute(`
    CREATE INDEX idx_${tableName}_transaction_date
    ON ${tableName} (transaction_date_key)
  `);

  for (const dim of dimensions) {
    await this.db.execute(`
      CREATE INDEX idx_${tableName}_${dim.name}
      ON ${tableName} (${dim.name}_key)
    `);
  }

  console.log(`Transaction fact table ${tableName} created successfully`);

  return {
    tableName,
    type: 'transaction',
    dimensions,
    measures,
```

```
      incrementalField,
    };
  }

  // Create periodic snapshot fact table
  async createPeriodicSnapshotFactTable(tableName, config) {
    const { dimensions, measures, periodicityField, periodicity } = config;

    console.log(`Creating periodic snapshot fact table: ${tableName}`);

    // Create table with structure suited for periodic snapshots
    let createTableSQL = `
      CREATE TABLE IF NOT EXISTS ${tableName} (
        ${tableName}_id INT AUTO_INCREMENT PRIMARY KEY,
        ${dimensions.map((dim) => `${dim.name}_key INT`).join(',\n        ')},
        ${periodicityField}_key INT,  -- e.g., date_key, month_key
        ${measures
          .map((m) => `${m.name} ${m.type || 'DECIMAL(16,4)'}`)
          .join(',\n        ')},
        snapshot_date DATETIME NOT NULL,
        etl_date DATETIME DEFAULT CURRENT_TIMESTAMP,

        ${dimensions
          .map(
            (dim) =>
              `FOREIGN KEY (${dim.name}_key) REFERENCES ${dim.tableName}
(${dim.keyField})`
          )
          .join(',\n        ')},
        FOREIGN KEY (${periodicityField}_key) REFERENCES dim_${periodicityField}
(${periodicityField}_id)
      )
    `;

    await this.db.execute(createTableSQL);

    // Add appropriate indexes
    await this.db.execute(`
      CREATE INDEX idx_${tableName}_${periodicityField}
      ON ${tableName} (${periodicityField}_key)
    `);

    // Add unique constraint to prevent duplicate snapshots
    await this.db.execute(`
      CREATE UNIQUE INDEX idx_${tableName}_snapshot
      ON ${tableName} (${periodicityField}_key, ${dimensions
      .map((d) => `${d.name}_key`)
      .join(', ')})
    `);

    console.log(
      `Periodic snapshot fact table ${tableName} created successfully`
    );

    return {
```

```
        tableName,
        type: 'periodic_snapshot',
        dimensions,
        measures,
        periodicityField,
        periodicity,
      };
    }

    // Create accumulating snapshot fact table
    async createAccumulatingSnapshotFactTable(tableName, config) {
      const { dimensions, measures, milestones, primaryEventField } = config;

      console.log(`Creating accumulating snapshot fact table: ${tableName}`);

      // Create table with structure suited for accumulating snapshots
      let createTableSQL = `
        CREATE TABLE IF NOT EXISTS ${tableName} (
          ${tableName}_id INT AUTO_INCREMENT PRIMARY KEY,
          ${dimensions.map((dim) => `${dim.name}_key INT`).join(',\n        ')},
          ${milestones.map((m) => `${m.name}_date_key INT`).join(',\n        ')},
          ${measures
            .map((m) => `${m.name} ${m.type || 'DECIMAL(16,4)'}`)
            .join(',\n        ')},
          ${milestones
            .map((m) => `${m.name}_completed BOOLEAN DEFAULT FALSE`)
            .join(',\n        ')},
          current_status VARCHAR(50),
          last_updated DATETIME NOT NULL,
          etl_date DATETIME DEFAULT CURRENT_TIMESTAMP,

          ${dimensions
            .map(
              (dim) =>
                `FOREIGN KEY (${dim.name}_key) REFERENCES ${dim.tableName}
(${dim.keyField})`
            )
            .join(',\n        ')},
          ${milestones
            .map(
              (m) =>
                `FOREIGN KEY (${m.name}_date_key) REFERENCES dim_date(date_id)`
            )
            .join(',\n        ')}
        )
      `;

      await this.db.execute(createTableSQL);

      // Add appropriate indexes
      for (const dim of dimensions) {
        await this.db.execute(`
          CREATE INDEX idx_${tableName}_${dim.name}
          ON ${tableName} (${dim.name}_key)
        `);
```

```
    }

    // Add index on status for filtering
    await this.db.execute(`
      CREATE INDEX idx_${tableName}_status
      ON ${tableName} (current_status)
    `);

    console.log(
      `Accumulating snapshot fact table ${tableName} created successfully`
    );

    return {
      tableName,
      type: 'accumulating_snapshot',
      dimensions,
      measures,
      milestones,
      primaryEventField,
    };
  }

  // Update records in an accumulating snapshot fact table
  async updateAccumulatingSnapshot(factConfig, newEvents) {
    const { tableName, dimensions, milestones, primaryEventField } = factConfig;

    console.log(`Updating accumulating snapshot fact table: ${tableName}`);

    // Process each new event
    for (const event of newEvents) {
      // Identify the milestone this event represents
      const milestone = milestones.find((m) => m.name === event.milestone);

      if (!milestone) {
        console.warn(`Unknown milestone: ${event.milestone}`);
        continue;
      }

      // Find the record to update
      const dimensionConditions = dimensions
        .map((dim) => `${dim.name}_key = ?`)
        .join(' AND ');

      const dimensionValues = dimensions.map((dim) => event[dim.sourceField]);

      const findRecordSQL = `
        SELECT ${tableName}_id, current_status
        FROM ${tableName}
        WHERE ${dimensionConditions}
      `;

      const existingRecord = await this.db.execute(
        findRecordSQL,
        dimensionValues
      );
```

```
        // Convert event date to date key
        const eventDate = new Date(event.eventDate);
        const dateKey = this.dimensionalModel.getDateKey(eventDate);

        if (existingRecord.rows.length === 0) {
          // New record - insert with this milestone
          // Prepare dimension keys and initial milestone
          const insertFields = [
            ...dimensions.map((dim) => `${dim.name}_key`),
            `${milestone.name}_date_key`,
            `${milestone.name}_completed`,
            'current_status',
            'last_updated',
          ];

          const insertValues = [
            ...dimensionValues,
            dateKey,
            true,
            milestone.name,
            new Date(),
          ];

          const placeholders = insertFields.map(() => '?').join(', ');

          await this.db.execute(
            `
            INSERT INTO ${tableName} (${insertFields.join(', ')})
            VALUES (${placeholders})
          `,
            insertValues
          );

          console.log(`Inserted new record for milestone: ${milestone.name}`);
        } else {
          // Update existing record with this milestone
          const recordId = existingRecord.rows[0][`${tableName}_id`];

          await this.db.execute(
            `
            UPDATE ${tableName}
            SET
              ${milestone.name}_date_key = ?,
              ${milestone.name}_completed = TRUE,
              current_status = ?,
              last_updated = NOW()
            WHERE ${tableName}_id = ?
          `,
            [dateKey, milestone.name, recordId]
          );

          console.log(
            `Updated record ${recordId} with milestone: ${milestone.name}`
          );
```

```
        }
      }

      console.log(`Accumulating snapshot update completed`);
    }
  }
```

OLAP (Online Analytical Processing)

OLAP systems enable complex analytical queries on large datasets with fast response times.

1. **OLAP Cube**: Multidimensional data structure for rapid analysis.

```
// OLAP Cube implementation concept
class OLAPCube {
  constructor(db, factTable, dimensions, measures) {
    this.db = db;
    this.factTable = factTable;
    this.dimensions = dimensions;
    this.measures = measures;
    this.cubeName = `cube_${factTable.replace('fact_', '')}`;
  }

  // Create cube structure
  async createCubeStructure() {
    console.log(`Creating OLAP cube: ${this.cubeName}`);

    // Create cube metadata
    await this.db.execute(`
      CREATE TABLE IF NOT EXISTS olap_cube_metadata (
        cube_name VARCHAR(100) PRIMARY KEY,
        fact_table VARCHAR(100) NOT NULL,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        last_refreshed DATETIME,
        dimensions JSON NOT NULL,
        measures JSON NOT NULL
      )
    `);

    // Store cube metadata
    await this.db.execute(
      `
      INSERT INTO olap_cube_metadata
        (cube_name, fact_table, dimensions, measures)
      VALUES (?, ?, ?, ?)
      ON DUPLICATE KEY UPDATE
        dimensions = VALUES(dimensions),
        measures = VALUES(measures)
    `,
      [
        this.cubeName,
        this.factTable,
        JSON.stringify(this.dimensions),
```

```
        JSON.stringify(this.measures),
      ]
    );

    // Create aggregate tables for common query patterns
    await this.createAggregates();

    console.log(`OLAP cube ${this.cubeName} created successfully`);
  }

  // Generate aggregation tables for different combinations of dimensions
  async createAggregates() {
    console.log(`Creating aggregates for cube: ${this.cubeName}`);

    // Create metadata for tracking aggregates
    await this.db.execute(`
      CREATE TABLE IF NOT EXISTS olap_aggregate_metadata (
        aggregate_name VARCHAR(100) PRIMARY KEY,
        cube_name VARCHAR(100) NOT NULL,
        dimensions JSON NOT NULL,
        created_at DATETIME DEFAULT CURRENT_TIMESTAMP,
        last_refreshed DATETIME,
        query_count INT DEFAULT 0,

        FOREIGN KEY (cube_name) REFERENCES olap_cube_metadata(cube_name)
          ON DELETE CASCADE
      )
    `);

    // Generate aggregations for common dimension combinations
    // Start with all dimensions
    await this.createAggregate(this.dimensions);

    // Generate aggregates for each individual dimension
    for (const dimension of this.dimensions) {
      await this.createAggregate([dimension]);
    }

    // Generate aggregates for pairs of dimensions
    // This could get expensive with many dimensions, so limit it
    if (this.dimensions.length > 1) {
      for (let i = 0; i < this.dimensions.length; i++) {
        for (let j = i + 1; j < this.dimensions.length; j++) {
          await this.createAggregate([this.dimensions[i], this.dimensions[j]]);
        }
      }
    }

    console.log(`Created aggregates for cube: ${this.cubeName}`);
  }

  // Create a single aggregate table for a set of dimensions
  async createAggregate(dimensions) {
    // Generate name based on dimensions
    const dimensionNames = dimensions
```

```
      .map((d) => d.name)
      .sort()
      .join('_');

    const aggregateName = `${this.cubeName}_${dimensionNames}`;

    console.log(`Creating aggregate: ${aggregateName}`);

    // Create aggregate table
    const dimensionFields = dimensions
      .map((d) => `${d.name}_key INT`)
      .join(',\n      ');

    const measureFields = this.measures
      .map((m) => {
        let aggregations = [];

        if (m.sum)
          aggregations.push(`sum_${m.name} ${m.type || 'DECIMAL(16,4)'}`);
        if (m.avg)
          aggregations.push(`avg_${m.name} ${m.type || 'DECIMAL(16,4)'}`);
        if (m.min)
          aggregations.push(`min_${m.name} ${m.type || 'DECIMAL(16,4)'}`);
        if (m.max)
          aggregations.push(`max_${m.name} ${m.type || 'DECIMAL(16,4)'}`);
        if (m.count) aggregations.push(`count_${m.name} INT`);

        return aggregations.join(',\n      ');
      })
      .join(',\n      ');

    await this.db.execute(`
      CREATE TABLE IF NOT EXISTS ${aggregateName} (
        ${dimensionFields},
        ${measureFields},
        record_count INT NOT NULL,
        last_refreshed DATETIME,
        PRIMARY KEY (${dimensions.map((d) => `${d.name}_key`).join(', ')})
      )
    `);

    // Register aggregate in metadata
    await this.db.execute(
      `
      INSERT INTO olap_aggregate_metadata
        (aggregate_name, cube_name, dimensions)
      VALUES (?, ?, ?)
      ON DUPLICATE KEY UPDATE
        dimensions = VALUES(dimensions)
    `,
      [
        aggregateName,
        this.cubeName,
        JSON.stringify(dimensions.map((d) => d.name)),
      ]
```

```
    );

    return aggregateName;
  }

  // Refresh cube data
  async refreshCube(options = {}) {
    const {
      refreshAggregates = true,
      incrementalRefresh = false,
      forceFullRefresh = false,
    } = options;

    console.log(`Refreshing OLAP cube: ${this.cubeName}`);

    // Get metadata for incremental refresh
    let lastRefreshed = null;

    if (incrementalRefresh && !forceFullRefresh) {
      const metadataResult = await this.db.execute(
        `
        SELECT last_refreshed FROM olap_cube_metadata
        WHERE cube_name = ?
      `,
        [this.cubeName]
      );

      if (metadataResult.rows.length > 0) {
        lastRefreshed = metadataResult.rows[0].last_refreshed;
      }
    }

    // Update each aggregate table
    const metadataResult = await this.db.execute(
      `
      SELECT aggregate_name, dimensions
      FROM olap_aggregate_metadata
      WHERE cube_name = ?
    `,
      [this.cubeName]
    );

    for (const aggregate of metadataResult.rows) {
      await this.refreshAggregate(
        aggregate.aggregate_name,
        JSON.parse(aggregate.dimensions),
        lastRefreshed
      );
    }

    // Update cube metadata
    await this.db.execute(
      `
      UPDATE olap_cube_metadata
      SET last_refreshed = NOW()
```

```
        WHERE cube_name = ?
      `,
        [this.cubeName]
      );

      console.log(`OLAP cube ${this.cubeName} refreshed successfully`);
    }

    // Refresh a single aggregate
    async refreshAggregate(aggregateName, dimensionNames, lastRefreshed) {
      console.log(`Refreshing aggregate: ${aggregateName}`);

      // Build dimensions clause for GROUP BY
      const dimensionsList = dimensionNames.map((d) => `ft.${d}_key`).join(', ');

      // Build measures clause with aggregations
      const measuresClause = this.measures
        .map((m) => {
          let aggregations = [];

          if (m.sum) aggregations.push(`SUM(ft.${m.name}) AS sum_${m.name}`);
          if (m.avg) aggregations.push(`AVG(ft.${m.name}) AS avg_${m.name}`);
          if (m.min) aggregations.push(`MIN(ft.${m.name}) AS min_${m.name}`);
          if (m.max) aggregations.push(`MAX(ft.${m.name}) AS max_${m.name}`);
          if (m.count)
            aggregations.push(`COUNT(ft.${m.name}) AS count_${m.name}`);

          return aggregations.join(',\n      ');
        })
        .join(',\n      ');

      // Build WHERE clause for incremental refresh
      let whereClause = '';
      let params = [];

      if (lastRefreshed) {
        whereClause = 'WHERE ft.etl_date > ?';
        params.push(lastRefreshed);
      }

      // Execute aggregation query
      if (lastRefreshed) {
        // Incremental refresh - delete existing data and insert new aggregations
        await this.db.execute(
          `
          DELETE FROM ${aggregateName}
          WHERE EXISTS (
            SELECT 1 FROM ${this.factTable} ft
            WHERE ft.etl_date > ?
            ${
              dimensionNames.length > 0
                ? `AND ${dimensionNames
                    .map((d) => `ft.${d}_key = ${aggregateName}.${d}_key`)
                    .join(' AND ')}`
                : ''
```

```
          }
        )
      `,
        [lastRefreshed]
      );
    } else {
      // Full refresh - truncate and rebuild
      await this.db.execute(`TRUNCATE TABLE ${aggregateName}`);
    }

    // Insert new data
    await this.db.execute(
      `
      INSERT INTO ${aggregateName}
      SELECT
        ${dimensionsList.length > 0 ? dimensionsList + ',' : ''}
        ${measuresClause},
        COUNT(*) AS record_count,
        NOW() AS last_refreshed
      FROM ${this.factTable} ft
      ${whereClause}
      ${dimensionsList.length > 0 ? `GROUP BY ${dimensionsList}` : ''}
    `,
      params
    );

    // Update refresh timestamp
    await this.db.execute(
      `
      UPDATE olap_aggregate_metadata
      SET last_refreshed = NOW()
      WHERE aggregate_name = ?
    `,
      [aggregateName]
    );

    console.log(`Aggregate ${aggregateName} refreshed successfully`);
  }

  // Query the cube with specific dimensions and measures
  async queryCube(options) {
    const {
      dimensions = [],
      filters = [],
      measures = [],
      sortBy,
      sortDirection = 'DESC',
      limit = 1000,
    } = options;

    console.log(`Querying OLAP cube: ${this.cubeName}`);

    // Find the best aggregate table to use
    const bestAggregate = await this.findBestAggregate(dimensions);
```

```
      if (!bestAggregate) {
        console.warn(
          `No suitable aggregate found for dimensions: ${dimensions.join(', ')}`
        );
        return this.executeFactTableQuery(options);
      }

      // Build SELECT clause
      const selectClause = [
        // Dimensions
        ...dimensions.map((d) => {
          // Join with dimension table to get descriptive value
          return `d_${d}.${d}_name AS ${d}`;
        }),

        // Measures with aggregations
        ...measures.map((m) => {
          if (m.aggregation === 'SUM') {
            return `agg.sum_${m.name} AS ${m.name}`;
          } else if (m.aggregation === 'AVG') {
            return `agg.avg_${m.name} AS ${m.name}`;
          } else if (m.aggregation === 'MIN') {
            return `agg.min_${m.name} AS ${m.name}`;
          } else if (m.aggregation === 'MAX') {
            return `agg.max_${m.name} AS ${m.name}`;
          } else if (m.aggregation === 'COUNT') {
            return `agg.count_${m.name} AS ${m.name}`;
          } else {
            return `agg.sum_${m.name} AS ${m.name}`; // Default to SUM
          }
        }),

        // Always include record count
        'agg.record_count',
      ].join(',\n      ');

      // Build FROM clause with JOINs for dimension tables
      const fromClause = [
        `FROM ${bestAggregate.aggregate_name} agg`,

        // Join dimension tables
        ...dimensions.map(
          (d) => `JOIN dim_${d} d_${d} ON agg.${d}_key = d_${d}.${d}_key`
        ),
      ].join('\n      ');

      // Build WHERE clause for filters
      let whereClause = '';
      let params = [];

      if (filters.length > 0) {
        whereClause =
          'WHERE ' +
          filters
            .map((filter) => {
```

```
            params.push(filter.value);

            if (filter.operator === '=') {
              return `d_${filter.dimension}.${filter.dimension}_name = ?`;
            } else if (filter.operator === 'CONTAINS') {
              return `d_${filter.dimension}.${filter.dimension}_name LIKE
CONCAT('%', ?, '%')`;
            } else if (filter.operator === 'IN') {
              const placeholders = filter.value.map(() => '?').join(', ');
              params = [...params, ...filter.value];
              return `d_${filter.dimension}.${filter.dimension}_name IN
(${placeholders})`;
            } else {
              return `d_${filter.dimension}.${filter.dimension}_name
${filter.operator} ?`;
            }
          })
          .join(' AND ');
    }

    // Build ORDER BY clause
    let orderByClause = '';

    if (sortBy) {
      orderByClause = `ORDER BY ${sortBy} ${sortDirection}`;
    }

    // Build LIMIT clause
    let limitClause = '';

    if (limit) {
      limitClause = `LIMIT ${limit}`;
    }

    // Combine all clauses into final query
    const query = `
      SELECT
        ${selectClause}
      ${fromClause}
      ${whereClause}
      ${orderByClause}
      ${limitClause}
    `;

    // Execute query
    console.log(
      `Executing OLAP query using aggregate: ${bestAggregate.aggregate_name}`
    );
    const result = await this.db.execute(query, params);

    // Increment usage counter for this aggregate
    await this.db.execute(
      `
      UPDATE olap_aggregate_metadata
      SET query_count = query_count + 1
```

```
        WHERE aggregate_name = ?
      `,
        [bestAggregate.aggregate_name]
      );

      return result.rows;
    }

    // Find the best aggregate table for a query
    async findBestAggregate(queryDimensions) {
      // Get all available aggregates
      const metadataResult = await this.db.execute(
        `
        SELECT aggregate_name, dimensions, query_count
        FROM olap_aggregate_metadata
        WHERE cube_name = ?
      `,
        [this.cubeName]
      );

      if (metadataResult.rows.length === 0) {
        return null;
      }

      // Score each aggregate based on:
      // 1. Coverage of the requested dimensions
      // 2. Minimizing extra dimensions (more specific)
      // 3. Usage frequency
      let bestAggregate = null;
      let bestScore = -1;

      for (const aggregate of metadataResult.rows) {
        const aggregateDimensions = JSON.parse(aggregate.dimensions);

        // Check if aggregate contains all requested dimensions
        const hasCoverage = queryDimensions.every((d) =>
          aggregateDimensions.includes(d)
        );

        if (!hasCoverage) {
          continue;
        }

        // Calculate score
        const coverageScore = queryDimensions.length / aggregateDimensions.length;
        const usageScore = Math.log(aggregate.query_count + 1) / 10;
        const score = coverageScore + usageScore;

        if (score > bestScore) {
          bestScore = score;
          bestAggregate = aggregate;
        }
      }

      return bestAggregate;
```

```
    }

    // Fall back to querying the fact table directly
    async executeFactTableQuery(options) {
      const {
        dimensions = [],
        filters = [],
        measures = [],
        sortBy,
        sortDirection = 'DESC',
        limit = 1000,
      } = options;

      console.log(`Executing query directly on fact table: ${this.factTable}`);

      // Build SELECT clause
      const selectClause = [
        // Dimensions
        ...dimensions.map((d) => `d_${d}.${d}_name AS ${d}`),

        // Measures with aggregations
        ...measures.map((m) => {
          if (m.aggregation === 'SUM') {
            return `SUM(ft.${m.name}) AS ${m.name}`;
          } else if (m.aggregation === 'AVG') {
            return `AVG(ft.${m.name}) AS ${m.name}`;
          } else if (m.aggregation === 'MIN') {
            return `MIN(ft.${m.name}) AS ${m.name}`;
          } else if (m.aggregation === 'MAX') {
            return `MAX(ft.${m.name}) AS ${m.name}`;
          } else if (m.aggregation === 'COUNT') {
            return `COUNT(ft.${m.name}) AS ${m.name}`;
          } else {
            return `SUM(ft.${m.name}) AS ${m.name}`; // Default to SUM
          }
        }),

        // Always include record count
        'COUNT(*) AS record_count',
      ].join(',\n      ');

      // Build FROM clause with JOINs
      const fromClause = [
        `FROM ${this.factTable} ft`,

        // Join dimension tables
        ...dimensions.map(
          (d) => `JOIN dim_${d} d_${d} ON ft.${d}_key = d_${d}.${d}_key`
        ),
      ].join('\n      ');

      // Build WHERE clause
      let whereClause = '';
      let params = [];
```

```
    if (filters.length > 0) {
      whereClause =
        'WHERE ' +
        filters
          .map((filter) => {
            params.push(filter.value);

            if (filter.operator === '=') {
              return `d_${filter.dimension}.${filter.dimension}_name = ?`;
            } else if (filter.operator === 'CONTAINS') {
              return `d_${filter.dimension}.${filter.dimension}_name LIKE
CONCAT('%', ?, '%')`;
            } else if (filter.operator === 'IN') {
              const placeholders = filter.value.map(() => '?').join(', ');
              params = [...params, ...filter.value];
              return `d_${filter.dimension}.${filter.dimension}_name IN
(${placeholders})`;
            } else {
              return `d_${filter.dimension}.${filter.dimension}_name
${filter.operator} ?`;
            }
          })
          .join(' AND ');
    }

    // Build GROUP BY clause
    const groupByClause =
      dimensions.length > 0
        ? `GROUP BY ${dimensions.map((d) => `d_${d}.${d}_name`).join(', ')}`
        : '';

    // Build ORDER BY clause
    let orderByClause = '';

    if (sortBy) {
      orderByClause = `ORDER BY ${sortBy} ${sortDirection}`;
    }

    // Build LIMIT clause
    let limitClause = '';

    if (limit) {
      limitClause = `LIMIT ${limit}`;
    }

    // Combine all clauses
    const query = `
      SELECT
        ${selectClause}
      ${fromClause}
      ${whereClause}
      ${groupByClause}
      ${orderByClause}
      ${limitClause}
    `;
```

```
    // Execute query
    console.log('Executing query directly on fact table');
    const result = await this.db.execute(query, params);

    return result.rows;
  }
}
```

2. **OLAP Operations**: Slice, dice, drill-down, roll-up, and pivot operations for data analysis.

```
// OLAP query builder with support for common OLAP operations
class OLAPQueryBuilder {
  constructor(olap) {
    this.olap = olap;
    this.dimensions = [];
    this.filters = [];
    this.measures = [];
    this.sortBy = null;
    this.sortDirection = 'DESC';
    this.limit = 1000;
  }

  // Add dimensions to query
  addDimensions(dimensions) {
    this.dimensions = [...this.dimensions, ...dimensions];
    return this;
  }

  // Add measures to query
  addMeasures(measures) {
    this.measures = [...this.measures, ...measures];
    return this;
  }

  // Add filters to query
  addFilters(filters) {
    this.filters = [...this.filters, ...filters];
    return this;
  }

  // Set sorting options
  setSorting(field, direction = 'DESC') {
    this.sortBy = field;
    this.sortDirection = direction;
    return this;
  }

  // Set result limit
  setLimit(limit) {
    this.limit = limit;
    return this;
  }
```

```
  // Slice operation: filter on one dimension
  slice(dimension, value) {
    this.filters.push({
      dimension,
      operator: '=',
      value,
    });

    return this;
  }

  // Dice operation: filter on multiple dimensions
  dice(filterCriteria) {
    for (const [dimension, value] of Object.entries(filterCriteria)) {
      this.filters.push({
        dimension,
        operator: '=',
        value,
      });
    }

    return this;
  }

  // Drill-down: add a dimension to get more detailed view
  drillDown(dimension) {
    if (!this.dimensions.includes(dimension)) {
      this.dimensions.push(dimension);
    }

    return this;
  }

  // Roll-up: remove a dimension to get more aggregated view
  rollUp(dimension) {
    this.dimensions = this.dimensions.filter((d) => d !== dimension);
    return this;
  }

  // Execute the query
  async execute() {
    const options = {
      dimensions: this.dimensions,
      filters: this.filters,
      measures: this.measures,
      sortBy: this.sortBy,
      sortDirection: this.sortDirection,
      limit: this.limit,
    };

    return this.olap.queryCube(options);
  }
}
```

6.4 Cloud Database Services

Cloud database services offer managed database solutions with various deployment models.

AWS Database Services

1. **Amazon RDS (Relational Database Service)**: Managed relational databases including MySQL, PostgreSQL, SQL Server, and Oracle.

```
// Example of connecting to and using Amazon RDS
const { Client } = require('pg');

// RDS connection setup
async function connectToRDS() {
  const client = new Client({
    host: 'mydb.abcdefghijkl.us-east-1.rds.amazonaws.com',
    port: 5432,
    database: 'mydb',
    user: process.env.RDS_USERNAME,
    password: process.env.RDS_PASSWORD,
    ssl: {
      rejectUnauthorized: false, // For development only
    },
  });

  await client.connect();
  console.log('Connected to Amazon RDS PostgreSQL instance');

  return client;
}

// Example usage of RDS
async function rdsExample() {
  let client;

  try {
    // Connect to RDS
    client = await connectToRDS();

    // Create a table
    await client.query(`
      CREATE TABLE IF NOT EXISTS users (
        id SERIAL PRIMARY KEY,
        username VARCHAR(100) NOT NULL UNIQUE,
        email VARCHAR(100) NOT NULL UNIQUE,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )
    `);

    // Insert a user
    const insertResult = await client.query(
      'INSERT INTO users (username, email) VALUES ($1, $2) RETURNING id',
      ['johndoe', 'john@example.com']
    );
```

```
    const userId = insertResult.rows[0].id;
    console.log(`Created user with ID: ${userId}`);

    // Query users
    const { rows } = await client.query('SELECT * FROM users');
    console.log('Users:', rows);
  } catch (error) {
    console.error('RDS operation failed:', error);
  } finally {
    if (client) {
      await client.end();
      console.log('RDS connection closed');
    }
  }
}
```

2. **Amazon DynamoDB**: Fully managed NoSQL database service.

```
// Example of using Amazon DynamoDB
const AWS = require('aws-sdk');

// Configure AWS SDK
AWS.config.update({
  region: 'us-east-1',
  accessKeyId: process.env.AWS_ACCESS_KEY_ID,
  secretAccessKey: process.env.AWS_SECRET_ACCESS_KEY,
});

// Create DynamoDB client
const dynamoDB = new AWS.DynamoDB();
const documentClient = new AWS.DynamoDB.DocumentClient();

// Create a table
async function createTable() {
  const params = {
    TableName: 'Users',
    KeySchema: [
      { AttributeName: 'userId', KeyType: 'HASH' }, // Partition key
    ],
    AttributeDefinitions: [{ AttributeName: 'userId', AttributeType: 'S' }],
    ProvisionedThroughput: {
      ReadCapacityUnits: 5,
      WriteCapacityUnits: 5,
    },
  };

  try {
    const data = await dynamoDB.createTable(params).promise();
    console.log('Table created:', data);

    // Wait for table to become active
    console.log('Waiting for table to become active...');
```

```
    await dynamoDB.waitFor('tableExists', { TableName: 'Users' }).promise();
    console.log('Table is now active');

    return data;
  } catch (error) {
    if (error.code === 'ResourceInUseException') {
      console.log('Table already exists');
    } else {
      console.error('Error creating table:', error);
      throw error;
    }
  }
}

// Put item in DynamoDB
async function createUser(userId, username, email) {
  const params = {
    TableName: 'Users',
    Item: {
      userId: userId,
      username: username,
      email: email,
      createdAt: new Date().toISOString(),
    },
    ConditionExpression: 'attribute_not_exists(userId)',
  };

  try {
    await documentClient.put(params).promise();
    console.log(`User created: ${userId}`);
    return { userId, username, email };
  } catch (error) {
    if (error.code === 'ConditionalCheckFailedException') {
      console.error(`User ${userId} already exists`);
    } else {
      console.error('Error creating user:', error);
    }
    throw error;
  }
}

// Get item from DynamoDB
async function getUser(userId) {
  const params = {
    TableName: 'Users',
    Key: {
      userId: userId,
    },
  };

  try {
    const data = await documentClient.get(params).promise();
    if (data.Item) {
      console.log(`Retrieved user: ${userId}`);
      return data.Item;
```

```
    } else {
      console.log(`User not found: ${userId}`);
      return null;
    }
  } catch (error) {
    console.error('Error getting user:', error);
    throw error;
  }
}

// Update item in DynamoDB
async function updateUser(userId, updates) {
  // Build update expression and attribute values
  let updateExpression = 'SET ';
  const expressionAttributeValues = {};
  const expressionAttributeNames = {};

  Object.entries(updates).forEach(([key, value], index) => {
    const valueKey = `:val${index}`;
    const nameKey = `#attr${index}`;

    updateExpression += index === 0 ? '' : ', ';
    updateExpression += `${nameKey} = ${valueKey}`;

    expressionAttributeValues[valueKey] = value;
    expressionAttributeNames[nameKey] = key;
  });

  // Add updatedAt timestamp
  updateExpression += ', #updatedAt = :updatedAt';
  expressionAttributeValues[':updatedAt'] = new Date().toISOString();
  expressionAttributeNames['#updatedAt'] = 'updatedAt';

  const params = {
    TableName: 'Users',
    Key: {
      userId: userId,
    },
    UpdateExpression: updateExpression,
    ExpressionAttributeValues: expressionAttributeValues,
    ExpressionAttributeNames: expressionAttributeNames,
    ReturnValues: 'ALL_NEW',
    ConditionExpression: 'attribute_exists(userId)',
  };

  try {
    const data = await documentClient.update(params).promise();
    console.log(`User updated: ${userId}`);
    return data.Attributes;
  } catch (error) {
    if (error.code === 'ConditionalCheckFailedException') {
      console.error(`User ${userId} does not exist`);
    } else {
      console.error('Error updating user:', error);
    }
```

```
    throw error;
  }
}

// Delete item from DynamoDB
async function deleteUser(userId) {
  const params = {
    TableName: 'Users',
    Key: {
      userId: userId,
    },
    ReturnValues: 'ALL_OLD',
  };

  try {
    const data = await documentClient.delete(params).promise();
    if (data.Attributes) {
      console.log(`User deleted: ${userId}`);
      return data.Attributes;
    } else {
      console.log(`User not found: ${userId}`);
      return null;
    }
  } catch (error) {
    console.error('Error deleting user:', error);
    throw error;
  }
}

// Query items in DynamoDB using Global Secondary Index
async function queryUsersByEmail(email) {
  // This requires a GSI on the email attribute
  const params = {
    TableName: 'Users',
    IndexName: 'EmailIndex',
    KeyConditionExpression: 'email = :email',
    ExpressionAttributeValues: {
      ':email': email,
    },
  };

  try {
    const data = await documentClient.query(params).promise();
    console.log(`Found ${data.Items.length} users with email: ${email}`);
    return data.Items;
  } catch (error) {
    console.error('Error querying users by email:', error);
    throw error;
  }
}
```

3. **Amazon Aurora**: MySQL and PostgreSQL-compatible relational database with enhanced performance.

```
// Example of using Amazon Aurora
const mysql = require('mysql2/promise');

// Aurora connection setup
async function connectToAurora() {
  const connection = await mysql.createConnection({
    host: 'myaurora.cluster-abcdefghijkl.us-east-1.rds.amazonaws.com',
    user: process.env.AURORA_USERNAME,
    password: process.env.AURORA_PASSWORD,
    database: 'mydb',
    ssl: {
      rejectUnauthorized: false, // For development only
    },
  });

  console.log('Connected to Amazon Aurora cluster');
  return connection;
}

// Create a connection pool for better performance
async function createAuroraPool() {
  const pool = mysql.createPool({
    host: 'myaurora.cluster-abcdefghijkl.us-east-1.rds.amazonaws.com',
    user: process.env.AURORA_USERNAME,
    password: process.env.AURORA_PASSWORD,
    database: 'mydb',
    ssl: {
      rejectUnauthorized: false, // For development only
    },
    waitForConnections: true,
    connectionLimit: 10,
    queueLimit: 0,
  });

  console.log('Created Aurora connection pool');
  return pool;
}

// Example of using Aurora's fast write capability
async function auroraFastWrites() {
  const pool = await createAuroraPool();

  try {
    // Start transaction
    const connection = await pool.getConnection();
    await connection.beginTransaction();

    try {
      // Batch insert for better performance
      const batchSize = 1000;
      const records = [];

      for (let i = 0; i < batchSize; i++) {
        records.push([`user${i}`, `user${i}@example.com`, new Date()]);
```

```
      }

      // Use prepared statement for batch insert
      const startTime = Date.now();

      const [result] = await connection.query(
        'INSERT INTO users (username, email, created_at) VALUES ?',
        [records]
      );

      const duration = Date.now() - startTime;

      console.log(`Inserted ${result.affectedRows} records in ${duration}ms`);
      console.log(
        `Rate: ${Math.round(
          result.affectedRows / (duration / 1000)
        )} records/second`
      );

      await connection.commit();
    } catch (error) {
      await connection.rollback();
      throw error;
    } finally {
      connection.release();
    }
  } catch (error) {
    console.error('Aurora operation failed:', error);
  } finally {
    await pool.end();
    console.log('Aurora connection pool closed');
  }
}

// Example of using Aurora's read replicas
async function auroraReadReplicaExample() {
  // Writer endpoint (primary instance)
  const writerPool = mysql.createPool({
    host: 'myaurora.cluster-abcdefghijkl.us-east-1.rds.amazonaws.com',
    user: process.env.AURORA_USERNAME,
    password: process.env.AURORA_PASSWORD,
    database: 'mydb',
    connectionLimit: 5,
  });

  // Reader endpoint (read replicas)
  const readerPool = mysql.createPool({
    host: 'myaurora.cluster-ro-abcdefghijkl.us-east-1.rds.amazonaws.com', // Read
replica endpoint
    user: process.env.AURORA_USERNAME,
    password: process.env.AURORA_PASSWORD,
    database: 'mydb',
    connectionLimit: 10,
  });
```

```
  try {
    // Write operation using writer endpoint
    const [writeResult] = await writerPool.execute(
      'INSERT INTO products (name, price) VALUES (?, ?)',
      ['New Product', 99.99]
    );

    console.log(`Inserted product ID: ${writeResult.insertId}`);

    // Read operation using reader endpoint
    const [rows] = await readerPool.execute(
      'SELECT * FROM products ORDER BY created_at DESC LIMIT 10'
    );

    console.log('Latest products:', rows);
  } catch (error) {
    console.error('Aurora operation failed:', error);
  } finally {
    await writerPool.end();
    await readerPool.end();
    console.log('Aurora connections closed');
  }
}
```

4. **Amazon DocumentDB**: MongoDB-compatible document database service.

```
// Example of using Amazon DocumentDB
const { MongoClient } = require('mongodb');

// DocumentDB connection setup
async function connectToDocumentDB() {
  const uri =
    'mongodb://username:password@mydocdb.cluster-abcdefghijkl.us-east-
1.docdb.amazonaws.com:27017/?
replicaSet=rs0&readPreference=secondaryPreferred&retryWrites=false';

  const client = new MongoClient(uri, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
    ssl: true,
    sslCA: '/path/to/rds-combined-ca-bundle.pem', // Download from AWS
  });

  await client.connect();
  console.log('Connected to Amazon DocumentDB');

  return client;
}

// Example usage of DocumentDB
async function documentDBExample() {
  let client;
```

```
    try {
      client = await connectToDocumentDB();
      const db = client.db('mydb');
      const collection = db.collection('users');

      // Insert document
      const result = await collection.insertOne({
        username: 'johndoe',
        email: 'john@example.com',
        profile: {
          firstName: 'John',
          lastName: 'Doe',
          age: 30,
        },
        interests: ['reading', 'travel', 'photography'],
        created_at: new Date(),
      });

      console.log(`Inserted document with ID: ${result.insertedId}`);

      // Query documents
      const users = await collection
        .find({
          'profile.age': { $gte: 25 },
        })
        .limit(10)
        .toArray();

      console.log(`Found ${users.length} users`);

      // Update document
      const updateResult = await collection.updateOne(
        { _id: result.insertedId },
        {
          $set: { 'profile.age': 31 },
          $push: { interests: 'cooking' },
        }
      );

      console.log(`Modified ${updateResult.modifiedCount} document`);

      // Aggregate documents
      const aggregation = await collection
        .aggregate([
          { $match: { 'profile.age': { $gte: 25 } } },
          { $group: { _id: null, averageAge: { $avg: '$profile.age' } } },
        ])
        .toArray();

      console.log('Average age:', aggregation[0].averageAge);
    } catch (error) {
      console.error('DocumentDB operation failed:', error);
    } finally {
      if (client) {
        await client.close();
```

```
      console.log('DocumentDB connection closed');
    }
  }
}
```

Azure Database Services

1. **Azure SQL Database**: Managed SQL Server database service.

```
// Example of using Azure SQL Database
const sql = require('mssql');

// Azure SQL Database connection setup
async function connectToAzureSQL() {
  const config = {
    user: process.env.AZURE_SQL_USERNAME,
    password: process.env.AZURE_SQL_PASSWORD,
    server: 'myazuredb.database.windows.net',
    database: 'mydb',
    options: {
      encrypt: true,
      enableArithAbort: true,
    },
  };

  try {
    await sql.connect(config);
    console.log('Connected to Azure SQL Database');
    return sql;
  } catch (error) {
    console.error('Connection to Azure SQL Database failed:', error);
    throw error;
  }
}

// Example usage of Azure SQL Database
async function azureSQLExample() {
  let pool;

  try {
    // Create connection pool
    pool = await new sql.ConnectionPool({
      user: process.env.AZURE_SQL_USERNAME,
      password: process.env.AZURE_SQL_PASSWORD,
      server: 'myazuredb.database.windows.net',
      database: 'mydb',
      options: {
        encrypt: true,
        enableArithAbort: true,
      },
    }).connect();

    console.log('Connected to Azure SQL Database with connection pool');
```

```
// Create a table
await pool.request().query(`
  IF NOT EXISTS (SELECT * FROM sys.tables WHERE name = 'Users')
  BEGIN
    CREATE TABLE Users (
      ID INT IDENTITY(1,1) PRIMARY KEY,
      Username NVARCHAR(100) NOT NULL UNIQUE,
      Email NVARCHAR(100) NOT NULL UNIQUE,
      CreatedAt DATETIME2 DEFAULT GETDATE()
    )
  END
`);

// Insert data using parameterized query
const insertResult = await pool
  .request()
  .input('username', sql.NVarChar, 'johndoe')
  .input('email', sql.NVarChar, 'john@example.com').query(`
    INSERT INTO Users (Username, Email)
    VALUES (@username, @email);
    SELECT SCOPE_IDENTITY() AS ID;
  `);

const userId = insertResult.recordset[0].ID;
console.log(`Inserted user with ID: ${userId}`);

// Query data
const queryResult = await pool
  .request()
  .input('id', sql.Int, userId)
  .query('SELECT * FROM Users WHERE ID = @id');

console.log('User data:', queryResult.recordset[0]);

// Use stored procedure
// First, create the stored procedure
await pool.request().query(`
  IF NOT EXISTS (SELECT * FROM sys.procedures WHERE name = 'GetUserByEmail')
  BEGIN
    EXEC('
      CREATE PROCEDURE GetUserByEmail
        @Email NVARCHAR(100)
      AS
      BEGIN
        SET NOCOUNT ON;
        SELECT * FROM Users WHERE Email = @Email;
      END
    ')
  END
`);

// Then call the stored procedure
const spResult = await pool
  .request()
```

```
      .input('email', sql.NVarChar, 'john@example.com')
      .execute('GetUserByEmail');

    console.log('User from stored procedure:', spResult.recordset[0]);
  } catch (error) {
    console.error('Azure SQL operation failed:', error);
  } finally {
    if (pool) {
      await pool.close();
      console.log('Azure SQL connection pool closed');
    }
  }
}
```

2. **Azure Cosmos DB**: Globally distributed, multi-model database service.

```
// Example of using Azure Cosmos DB SQL API
const { CosmosClient } = require('@azure/cosmos');

// Cosmos DB connection setup
async function connectToCosmosDB() {
  const endpoint = process.env.COSMOS_ENDPOINT;
  const key = process.env.COSMOS_KEY;

  const client = new CosmosClient({ endpoint, key });
  console.log('Created Cosmos DB client');

  return client;
}

// Example usage of Cosmos DB SQL API
async function cosmosDBExample() {
  let client;

  try {
    client = await connectToCosmosDB();

    // Create database and container if they don't exist
    const { database } = await client.databases.createIfNotExists({
      id: 'mydb',
    });
    console.log(`Database: ${database.id}`);

    const { container } = await database.containers.createIfNotExists({
      id: 'users',
      partitionKey: { paths: ['/userId'] },
    });
    console.log(`Container: ${container.id}`);

    // Create an item
    const newUser = {
      userId: 'user1',
      username: 'johndoe',
```

```
      email: 'john@example.com',
      profile: {
        firstName: 'John',
        lastName: 'Doe',
        age: 30,
      },
      interests: ['reading', 'travel', 'photography'],
      type: 'user',
      createdAt: new Date().toISOString(),
    };

    const { resource: createdUser } = await container.items.create(newUser);
    console.log(`Created user: ${createdUser.id}`);

    // Query items
    const querySpec = {
      query:
        'SELECT * FROM users u WHERE u.type = @type AND u.profile.age >= @minAge',
      parameters: [
        { name: '@type', value: 'user' },
        { name: '@minAge', value: 25 },
      ],
    };

    const { resources: users } = await container.items
      .query(querySpec)
      .fetchAll();
    console.log(`Found ${users.length} users`);

    // Read a specific item
    const { resource: user } = await container
      .item(createdUser.id, createdUser.userId)
      .read();
    console.log('Read user:', user.username);

    // Update an item
    user.profile.age = 31;
    user.interests.push('cooking');

    const { resource: updatedUser } = await container
      .item(user.id, user.userId)
      .replace(user);

    console.log('Updated user:', updatedUser.username);

    // Delete an item
    await container.item(user.id, user.userId).delete();
    console.log('Deleted user');
  } catch (error) {
    console.error('Cosmos DB operation failed:', error);
  }
}

// Example of using Cosmos DB with multiple APIs
async function cosmosDBApisExample() {
```

```
  // SQL API example - already shown above

  // MongoDB API
  const { MongoClient } = require('mongodb');

  const mongoConnectionString =
    'mongodb://mycosmosdb:password@mycosmosdb.mongo.cosmos.azure.com:10255/?
ssl=true&replicaSet=globaldb&retrywrites=false&maxIdleTimeMS=120000&appName=@mycosmo
sdb@';

  try {
    const mongoClient = new MongoClient(mongoConnectionString);
    await mongoClient.connect();

    const db = mongoClient.db('mydb');
    const collection = db.collection('users');

    // Insert a document
    const result = await collection.insertOne({
      username: 'janedoe',
      email: 'jane@example.com',
      createdAt: new Date(),
    });

    console.log(`Inserted MongoDB API document: ${result.insertedId}`);

    await mongoClient.close();
  } catch (error) {
    console.error('Cosmos DB MongoDB API operation failed:', error);
  }

  // Gremlin API (Graph)
  const gremlin = require('gremlin');

  const gremlinEndpoint = 'wss://mycosmosdb.gremlin.cosmos.azure.com:443/';
  const gremlinKey = process.env.COSMOS_KEY;

  try {
    const authenticator = new gremlin.driver.auth.PlainTextSaslAuthenticator(
      `/dbs/mydb/colls/graph`,
      gremlinKey
    );

    const client = new gremlin.driver.Client(gremlinEndpoint, {
      authenticator,
      traversalsource: 'g',
      rejectUnauthorized: true,
      mimeType: 'application/vnd.gremlin-v2.0+json',
    });

    await client.open();
    console.log('Connected to Cosmos DB Gremlin API');

    // Add vertices
    await client.submit(
```

```
      'g.addV("person").property("id", "john").property("name", "John Doe")'
    );
    await client.submit(
      'g.addV("person").property("id", "jane").property("name", "Jane Smith")'
    );

    // Add an edge
    await client.submit('g.V("john").addE("knows").to(g.V("jane"))');

    // Query the graph
    const result = await client.submit(
      'g.V().hasLabel("person").values("name")'
    );

    console.log('People in the graph:');
    for (const name of result._items) {
      console.log(`- ${name}`);
    }

    await client.close();
  } catch (error) {
    console.error('Cosmos DB Gremlin API operation failed:', error);
  }

  // Table API
  const {
    TableClient,
    AzureNamedKeyCredential,
  } = require('@azure/data-tables');

  try {
    const account = 'mycosmosdb';
    const accountKey = process.env.COSMOS_KEY;
    const tableName = 'users';

    const credential = new AzureNamedKeyCredential(account, accountKey);
    const tableClient = new TableClient(
      `https://${account}.table.cosmos.azure.com:443/`,
      tableName,
      credential
    );

    // Create table if not exists
    await tableClient.createTable();

    // Add an entity
    const entity = {
      partitionKey: 'users',
      rowKey: 'user3',
      username: 'bobsmith',
      email: 'bob@example.com',
    };

    await tableClient.createEntity(entity);
    console.log('Created Table API entity');
```

```
    // Query entities
    const entities = tableClient.listEntities({
      queryOptions: { filter: "PartitionKey eq 'users'" },
    });

    console.log('Table API entities:');
    for await (const entity of entities) {
      console.log(`- ${entity.rowKey}: ${entity.username}`);
    }
  } catch (error) {
    console.error('Cosmos DB Table API operation failed:', error);
  }
}
```

Google Cloud Platform Database Services

1. **Cloud SQL**: Managed MySQL, PostgreSQL, and SQL Server databases.

```
// Example of connecting to Google Cloud SQL
const mysql = require('mysql2/promise');

// Cloud SQL connection setup
async function connectToCloudSQL() {
  // For development with Cloud SQL Auth Proxy
  const connection = await mysql.createConnection({
    host: '127.0.0.1',
    port: 3306, // The port configured in Cloud SQL Auth Proxy
    user: process.env.CLOUD_SQL_USERNAME,
    password: process.env.CLOUD_SQL_PASSWORD,
    database: 'mydb',
  });

  console.log('Connected to Google Cloud SQL via Auth Proxy');
  return connection;
}

// For production with private IP
async function connectToCloudSQLProduction() {
  const connection = await mysql.createConnection({
    host: 'private-ip-address', // Private IP address of your Cloud SQL instance
    user: process.env.CLOUD_SQL_USERNAME,
    password: process.env.CLOUD_SQL_PASSWORD,
    database: 'mydb',
  });

  console.log('Connected to Google Cloud SQL via Private IP');
  return connection;
}

// For production with public IP and SSL
async function connectToCloudSQLWithSSL() {
  const fs = require('fs');
```

```
  const connection = await mysql.createConnection({
    host: 'public-ip-address', // Public IP address of your Cloud SQL instance
    user: process.env.CLOUD_SQL_USERNAME,
    password: process.env.CLOUD_SQL_PASSWORD,
    database: 'mydb',
    ssl: {
      ca: fs.readFileSync('/path/to/server-ca.pem'),
      key: fs.readFileSync('/path/to/client-key.pem'),
      cert: fs.readFileSync('/path/to/client-cert.pem'),
    },
  });

  console.log('Connected to Google Cloud SQL via Public IP with SSL');
  return connection;
}

// Example usage of Cloud SQL
async function cloudSQLExample() {
  let connection;

  try {
    connection = await connectToCloudSQL();

    // Create a table
    await connection.execute(`
      CREATE TABLE IF NOT EXISTS users (
        id INT AUTO_INCREMENT PRIMARY KEY,
        username VARCHAR(100) NOT NULL UNIQUE,
        email VARCHAR(100) NOT NULL UNIQUE,
        created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
      )
    `);

    // Insert a user
    const [result] = await connection.execute(
      'INSERT INTO users (username, email) VALUES (?, ?)',
      ['johndoe', 'john@example.com']
    );

    console.log(`Inserted user with ID: ${result.insertId}`);

    // Query users
    const [rows] = await connection.execute('SELECT * FROM users');
    console.log('Users:', rows);
  } catch (error) {
    console.error('Cloud SQL operation failed:', error);
  } finally {
    if (connection) {
      await connection.end();
      console.log('Cloud SQL connection closed');
    }
  }
}
```

2. **Firestore**: NoSQL document database for mobile, web, and server development.

```
// Example of using Firestore
const { Firestore } = require('@google-cloud/firestore');

// Firestore connection setup
function connectToFirestore() {
  // Initialize Firestore with auto-credentials from environment
  const firestore = new Firestore();
  console.log('Created Firestore client');
  return firestore;
}

// Example usage of Firestore
async function firestoreExample() {
  const firestore = connectToFirestore();

  try {
    // Add a document
    const usersCollection = firestore.collection('users');

    const newUser = {
      username: 'johndoe',
      email: 'john@example.com',
      profile: {
        firstName: 'John',
        lastName: 'Doe',
        age: 30,
      },
      interests: ['reading', 'travel', 'photography'],
      createdAt: Firestore.FieldValue.serverTimestamp(),
    };

    const docRef = await usersCollection.add(newUser);
    console.log(`Added user with ID: ${docRef.id}`);

    // Get a document
    const docSnapshot = await docRef.get();
    console.log('User data:', docSnapshot.data());

    // Update a document
    await docRef.update({
      'profile.age': 31,
      interests: Firestore.FieldValue.arrayUnion('cooking'),
      updatedAt: Firestore.FieldValue.serverTimestamp(),
    });

    console.log('Updated user document');

    // Query documents
    const snapshot = await usersCollection
      .where('profile.age', '>=', 25)
      .orderBy('profile.age')
      .limit(10)
```

```
      .get();

    console.log(`Found ${snapshot.size} users`);
    snapshot.forEach((doc) => {
      console.log(`- ${doc.id}: ${doc.data().username}`);
    });

    // Delete a document
    await docRef.delete();
    console.log('Deleted user document');

    // Batch writes
    const batch = firestore.batch();

    for (let i = 0; i < 5; i++) {
      const docRef = usersCollection.doc();
      batch.set(docRef, {
        username: `user${i}`,
        email: `user${i}@example.com`,
        createdAt: Firestore.FieldValue.serverTimestamp(),
      });
    }

    await batch.commit();
    console.log('Batch write completed');

    // Transaction
    await firestore.runTransaction(async (transaction) => {
      const counterRef = firestore.collection('counters').doc('users');
      const counterDoc = await transaction.get(counterRef);

      let count = 1;
      if (counterDoc.exists) {
        count = counterDoc.data().count + 1;
      }

      transaction.set(counterRef, { count });

      const newUserRef = usersCollection.doc();
      transaction.set(newUserRef, {
        username: `user${count}`,
        email: `user${count}@example.com`,
        userNumber: count,
        createdAt: Firestore.FieldValue.serverTimestamp(),
      });

      return count;
    });

    console.log('Transaction completed');
  } catch (error) {
    console.error('Firestore operation failed:', error);
  }
}
```

3. **Bigtable**: Scalable, fully managed NoSQL wide-column database service.

```
// Example of using Bigtable
const { Bigtable } = require('@google-cloud/bigtable');

// Bigtable connection setup
function connectToBigtable() {
  const bigtable = new Bigtable();
  console.log('Created Bigtable client');
  return bigtable;
}

// Example usage of Bigtable
async function bigtableExample() {
  const bigtable = connectToBigtable();

  // Define instance and table names
  const instanceId = 'my-bigtable-instance';
  const tableId = 'user-events';

  try {
    // Get instance
    const instance = bigtable.instance(instanceId);

    // Check if instance exists
    const [instanceExists] = await instance.exists();
    if (!instanceExists) {
      console.log(`Instance ${instanceId} does not exist.`);

      // For production, you would create the instance here
      // However, instance creation is typically done through
      // Google Cloud Console or gcloud CLI due to IAM requirements
      return;
    }

    // Get table
    const table = instance.table(tableId);

    // Check if table exists
    const [tableExists] = await table.exists();
    if (!tableExists) {
      console.log(`Creating table ${tableId}...`);

      await table.create({
        families: [
          {
            name: 'events',
            rule: {
              versions: 5, // Keep 5 versions
              age: 60 * 60 * 24 * 7, // 7 days TTL (in seconds)
            },
          },
          {
            name: 'metrics',
```

```
            rule: {
              versions: 1, // Keep only most recent version
            },
          },
        ],
      });

      console.log(`Table ${tableId} created.`);
    }

    // Write data
    console.log('Writing data...');

    // Create a timestamp for versioning (microseconds)
    const timestamp = new Date().getTime() * 1000;

    const rows = [
      {
        key: 'user1#2023-06-15',
        data: {
          events: {
            login: [
              {
                value: JSON.stringify({
                  device: 'mobile',
                  location: 'US',
                  ip: '192.168.1.1',
                }),
                timestamp,
              },
            ],
            page_view: [
              {
                value: JSON.stringify({
                  page: 'home',
                  duration: 45,
                  referrer: 'google',
                }),
                timestamp: timestamp + 1000,
              },
            ],
          },
          metrics: {
            daily_visits: [
              {
                value: '5',
                timestamp,
              },
            ],
            avg_session_time: [
              {
                value: '180',
                timestamp,
              },
            ],
```

```
        },
      },
    },
    {
      key: 'user2#2023-06-15',
      data: {
        events: {
          login: [
            {
              value: JSON.stringify({
                device: 'desktop',
                location: 'UK',
                ip: '192.168.2.2',
              }),
              timestamp,
            },
          ],
        },
        metrics: {
          daily_visits: [
            {
              value: '2',
              timestamp,
            },
          ],
        },
      },
    },
  ];

  await table.insert(rows);
  console.log('Data written successfully.');

  // Read a single row
  console.log('Reading a single row...');
  const [row] = await table.row('user1#2023-06-15').get();
  console.log('Row data:');
  console.log(JSON.stringify(row.data, null, 2));

  // Read specific cells
  console.log('Reading specific cells...');
  const [cells] = await table.row('user1#2023-06-15').get({
    families: ['events'],
    columns: ['events:login'],
  });
  console.log('Login event:');
  console.log(JSON.stringify(cells, null, 2));

  // Scan a range of rows
  console.log('Scanning range of rows...');
  const prefix = 'user';
  const range = {
    start: `${prefix}1`,
    end: `${prefix}3`,
  };
```

```
const [rowsResponse] = await table.getRows({
  start: range.start,
  end: range.end,
  limit: 10,
});

console.log(`Found ${rowsResponse.length} rows`);
rowsResponse.forEach((row) => {
  console.log(`- ${row.id}`);
});

// Create a read stream for large datasets
console.log('Creating read stream...');
const stream = table.createReadStream({
  start: prefix,
  end: `${prefix}z`,
  limit: 1000,
});

let rowCount = 0;
stream
  .on('data', () => {
    rowCount++;
  })
  .on('end', () => {
    console.log(`Read ${rowCount} rows from stream.`);
  })
  .on('error', (err) => {
    console.error('Stream error:', err);
  });

// Filters and complex queries
console.log('Querying with filters...');
const [filteredRows] = await table.getRows({
  filter: [
    {
      column: {
        family: 'events',
        qualifier: 'login',
      },
      value: {
        substring: 'mobile',
      },
    },
  ],
});

console.log(`Found ${filteredRows.length} rows with mobile logins`);

// Delete a row
console.log('Deleting a row...');
await table.row('user2#2023-06-15').delete();
console.log('Row deleted.');
```

```
    // Create and use a mutation object for complex operations
    console.log('Using mutations...');
    const mutation = {
      key: 'user1#2023-06-15',
      method: 'insert',
      data: {
        events: {
          purchase: [
            {
              value: JSON.stringify({
                product_id: 'prod123',
                amount: 99.99,
                currency: 'USD',
              }),
              timestamp: timestamp + 2000,
            },
          ],
        },
      },
    };

    await table.mutate(mutation);
    console.log('Mutation applied.');
  } catch (error) {
    console.error('Bigtable operation failed:', error);
  }
}
```

4. **Cloud Spanner**: Globally distributed, strongly consistent database service.

```
// Example of using Cloud Spanner
const { Spanner } = require('@google-cloud/spanner');

// Spanner connection setup
function connectToSpanner() {
  const spanner = new Spanner();
  console.log('Created Spanner client');
  return spanner;
}

// Example usage of Spanner
async function spannerExample() {
  const spanner = connectToSpanner();

  // Define instance and database names
  const instanceId = 'my-spanner-instance';
  const databaseId = 'mydb';

  try {
    // Get instance
    const instance = spanner.instance(instanceId);

    // Get database
```

```
const database = instance.database(databaseId);

// Create a schema (normally done through gcloud CLI or Console)
// This is for illustration, and requires appropriate permissions
console.log('Checking if database exists...');

const [databaseExists] = await database.exists();
if (!databaseExists) {
  console.log(`Database ${databaseId} does not exist. Creating...`);

  const [, operation] = await instance.createDatabase(databaseId, {
    schema: [
      `CREATE TABLE Users (
        UserId STRING(36) NOT NULL,
        Username STRING(100) NOT NULL,
        Email STRING(100) NOT NULL,
        CreatedAt TIMESTAMP NOT NULL OPTIONS (allow_commit_timestamp=true)
      ) PRIMARY KEY (UserId)`,

      `CREATE TABLE Orders (
        OrderId STRING(36) NOT NULL,
        UserId STRING(36) NOT NULL,
        Amount FLOAT64 NOT NULL,
        Status STRING(20) NOT NULL,
        CreatedAt TIMESTAMP NOT NULL OPTIONS (allow_commit_timestamp=true),
        CONSTRAINT FK_UserOrder FOREIGN KEY (UserId) REFERENCES Users (UserId)
      ) PRIMARY KEY (OrderId)`,
    ],
  });

  console.log('Database creation in progress...');
  await operation.promise();
  console.log('Database created.');
}

// Insert data using a transaction
console.log('Inserting data...');

await database.runTransaction(async (transaction) => {
  try {
    // Generate UUIDs
    const userId = generateUuid();
    const orderId = generateUuid();

    // Insert user
    await transaction.insert('Users', {
      UserId: userId,
      Username: 'johndoe',
      Email: 'john@example.com',
      CreatedAt: Spanner.timestamp(new Date()),
    });

    // Insert order
    await transaction.insert('Orders', {
      OrderId: orderId,
```

```
            UserId: userId,
            Amount: 99.99,
            Status: 'pending',
            CreatedAt: Spanner.timestamp(new Date()),
          });

          // Commit transaction
          await transaction.commit();
          console.log(`Inserted User (${userId}) and Order (${orderId})`);

          return { userId, orderId };
        } catch (error) {
          console.error('Transaction failed, rolling back:', error);
          await transaction.rollback();
          throw error;
        }
      });

      // Read using a strongly consistent read
      console.log('Reading data with strong consistency...');

      const [users] = await database.table('Users').read({
        columns: ['UserId', 'Username', 'Email', 'CreatedAt'],
        keySet: { all: true },
      });

      console.log(`Found ${users.length} users:`);
      users.forEach((user) => {
        console.log(`- ${user.Username} (${user.Email})`);
      });

      // Execute SQL query
      console.log('Executing SQL query...');

      const [orderRows] = await database.run({
        sql: `
          SELECT o.OrderId, o.Amount, o.Status, u.Username
          FROM Orders o
          JOIN Users u ON o.UserId = u.UserId
          WHERE o.Amount > @minAmount
          ORDER BY o.CreatedAt DESC
        `,
        params: {
          minAmount: 50.0,
        },
      });

      console.log(`Found ${orderRows.length} orders above $50:`);
      orderRows.forEach((order) => {
        console.log(
          `- Order ${order.OrderId}: $${order.Amount} (${order.Status}) by
${order.Username}`
        );
      });
```

```
// Execute a read-write transaction
console.log('Executing read-write transaction...');

await database.runTransaction(async (transaction) => {
  try {
    // Get the first order
    const [orderRows] = await transaction.run({
      sql: 'SELECT OrderId, Status FROM Orders LIMIT 1',
    });

    if (orderRows.length === 0) {
      console.log('No orders found.');
      await transaction.rollback();
      return;
    }

    const order = orderRows[0];

    // Update order status
    await transaction.update('Orders', {
      OrderId: order.OrderId,
      Status: 'completed',
    });

    // Commit transaction
    await transaction.commit();
    console.log(`Updated order ${order.OrderId} status to completed`);
  } catch (error) {
    console.error('Transaction failed, rolling back:', error);
    await transaction.rollback();
    throw error;
  }
});

// Partitioned query for large datasets
console.log('Executing partitioned query...');

const [partitions] = await database.createQueryPartitions({
  sql: 'SELECT UserId, Username FROM Users',
});

console.log(`Query partitioned into ${partitions.length} partitions`);

const promises = partitions.map(async (partition) => {
  const [rows] = await database.run({
    sql: partition.sql,
    params: partition.params,
    partition: partition,
  });

  return rows;
});

const results = await Promise.all(promises);
const flatResults = results.flat();
```

```
    console.log(
      `Partitioned query returned ${flatResults.length} total results`
    );

    // Helper function to generate UUID
    function generateUuid() {
      return 'xxxxxxxx-xxxx-4xxx-yxxx-xxxxxxxxxxxx'.replace(
        /[xy]/g,
        function (c) {
          const r = (Math.random() * 16) | 0;
          const v = c === 'x' ? r : (r & 0x3) | 0x8;
          return v.toString(16);
        }
      );
    }
  } catch (error) {
    console.error('Spanner operation failed:', error);
  } finally {
    // Close the client
    spanner.close();
    console.log('Spanner client closed');
  }
}
```

6.5 Database Recommendations

Guidance on choosing the right database system for different use cases.

Decision Framework

1. **Requirements Analysis**: A systematic approach to analyze application requirements.

```
// Database selection framework
class DatabaseSelectionFramework {
  constructor() {
    this.requirements = {
      // Data characteristics
      schema: {
        structured: 0, // Fixed schema
        semiStructured: 0, // Flexible schema
        unstructured: 0, // No schema
      },

      // Data relationships
      relationships: {
        simple: 0, // Few or no relationships
        moderate: 0, // Some relationships
        complex: 0, // Many complex relationships
      },

      // Query patterns
      queryPatterns: {
```

```
        keyValue: 0, // Simple key-value lookups
        documentLookup: 0, // Document retrieval by ID
        complexQueries: 0, // Joins, aggregations
        adhocQueries: 0, // Unpredictable queries
        graphTraversal: 0, // Network/path queries
        fullTextSearch: 0, // Text search
        timeSeries: 0, // Time-ordered data
      },

      // Scale and performance
      scale: {
        readVolume: 0, // Read operations per second
        writeVolume: 0, // Write operations per second
        dataVolume: 0, // Total data size
        growthRate: 0, // Projected growth rate
      },

      // Consistency and availability
      cap: {
        consistency: 0, // Strong consistency
        availability: 0, // High availability
        partitionTolerance: 0, // Network partition tolerance
      },

      // Special requirements
      special: {
        globalDistribution: 0, // Multiple regions
        geospatial: 0, // Geospatial queries
        streaming: 0, // Stream processing
        analytics: 0, // Analytical workloads
        caching: 0, // Caching layer
      },
    };

    this.databaseTypes = {
      // Relational
      rdbms: {
        traditional: {
          // MySQL, PostgreSQL, etc.
          score: 0,
          description: 'Traditional relational database',
          examples: ['MySQL', 'PostgreSQL', 'SQL Server', 'Oracle'],
        },
        distributed: {
          // Spanner, CockroachDB, etc.
          score: 0,
          description: 'Distributed SQL database',
          examples: ['CockroachDB', 'Google Spanner', 'Amazon Aurora', 'TiDB'],
        },

        // NoSQL
        documentDb: {
          // MongoDB, Couchbase, etc.
          score: 0,
          description: 'Document database',
```

```
      examples: [
        'MongoDB',
        'Couchbase',
        'Amazon DocumentDB',
        'Azure Cosmos DB (SQL API)',
      ],
    },
    keyValue: {
      // Redis, DynamoDB, etc.
      score: 0,
      description: 'Key-value database',
      examples: [
        'Redis',
        'Amazon DynamoDB',
        'Riak',
        'Azure Cosmos DB (Table API)',
      ],
    },
    columnFamily: {
      // Cassandra, HBase, etc.
      score: 0,
      description: 'Column-family database',
      examples: [
        'Apache Cassandra',
        'Apache HBase',
        'Google Bigtable',
        'ScyllaDB',
      ],
    },
    graphDb: {
      // Neo4j, Neptune, etc.
      score: 0,
      description: 'Graph database',
      examples: [
        'Neo4j',
        'Amazon Neptune',
        'JanusGraph',
        'Azure Cosmos DB (Gremlin API)',
      ],
    },

    // Specialized
    searchEngine: {
      // Elasticsearch, etc.
      score: 0,
      description: 'Search engine database',
      examples: ['Elasticsearch', 'Solr', 'Algolia', 'Azure Search'],
    },
    timeSeriesDb: {
      // InfluxDB, TimescaleDB, etc.
      score: 0,
      description: 'Time-series database',
      examples: [
        'InfluxDB',
        'TimescaleDB',
```

```
            'Prometheus',
            'Amazon Timestream',
          ],
        },
        inMemoryDb: {
          // Redis, Memcached, etc.
          score: 0,
          description: 'In-memory database',
          examples: ['Redis', 'Memcached', 'Apache Ignite', 'VoltDB'],
        },
        multiModelDb: {
          // ArangoDB, Cosmos DB, etc.
          score: 0,
          description: 'Multi-model database',
          examples: ['ArangoDB', 'Azure Cosmos DB', 'FaunaDB', 'OrientDB'],
        },
      },
    };
  }

  // Rate each requirement on a scale of 0-10
  rateRequirement(category, subcategory, value) {
    if (value < 0) value = 0;
    if (value > 10) value = 10;

    this.requirements[category][subcategory] = value;
    return this;
  }

  // Set requirement directly with a specific value
  setRequirement(category, subcategory, value) {
    this.requirements[category][subcategory] = value;
    return this;
  }

  // Calculate database scores based on requirements
  calculateScores() {
    const r = this.requirements;

    // Relational - Traditional (MySQL, PostgreSQL, etc.)
    this.databaseTypes.rdbms.traditional.score =
      r.schema.structured * 1.0 +
      r.schema.semiStructured * 0.3 +
      r.schema.unstructured * 0.0 +
      r.relationships.simple * 0.5 +
      r.relationships.moderate * 0.9 +
      r.relationships.complex * 1.0 +
      r.queryPatterns.keyValue * 0.8 +
      r.queryPatterns.documentLookup * 0.5 +
      r.queryPatterns.complexQueries * 1.0 +
      r.queryPatterns.adhocQueries * 0.9 +
      r.queryPatterns.graphTraversal * 0.3 +
      r.queryPatterns.fullTextSearch * 0.4 +
      r.queryPatterns.timeSeries * 0.5 +
      (Math.min(r.scale.readVolume, 7) / 7) * 10 + // Up to medium scale
```

```
      (Math.min(r.scale.writeVolume, 6) / 6) * 10 +
      (Math.min(r.scale.dataVolume, 7) / 7) * 10 +
      r.cap.consistency * 0.9 +
      r.cap.availability * 0.6 +
      r.cap.partitionTolerance * 0.3 +
      r.special.globalDistribution * 0.2 +
      r.special.geospatial * 0.7 +
      r.special.streaming * 0.3 +
      r.special.analytics * 0.7 +
      r.special.caching * 0.2;

    // Distributed SQL (Spanner, CockroachDB, etc.)
    this.databaseTypes.rdbms.distributed.score =
      r.schema.structured * 1.0 +
      r.schema.semiStructured * 0.3 +
      r.schema.unstructured * 0.0 +
      r.relationships.simple * 0.5 +
      r.relationships.moderate * 0.9 +
      r.relationships.complex * 0.9 +
      r.queryPatterns.keyValue * 0.8 +
      r.queryPatterns.documentLookup * 0.5 +
      r.queryPatterns.complexQueries * 0.9 +
      r.queryPatterns.adhocQueries * 0.8 +
      r.queryPatterns.graphTraversal * 0.2 +
      r.queryPatterns.fullTextSearch * 0.3 +
      r.queryPatterns.timeSeries * 0.4 +
      (Math.min(r.scale.readVolume, 10) / 10) * 10 +
      (Math.min(r.scale.writeVolume, 9) / 9) * 10 +
      (Math.min(r.scale.dataVolume, 10) / 10) * 10 +
      r.cap.consistency * 0.9 +
      r.cap.availability * 0.8 +
      r.cap.partitionTolerance * 0.8 +
      r.special.globalDistribution * 0.9 +
      r.special.geospatial * 0.6 +
      r.special.streaming * 0.3 +
      r.special.analytics * 0.6 +
      r.special.caching * 0.2;

    // Document Database (MongoDB, etc.)
    this.databaseTypes.documentDb.score =
      r.schema.structured * 0.5 +
      r.schema.semiStructured * 1.0 +
      r.schema.unstructured * 0.7 +
      r.relationships.simple * 0.9 +
      r.relationships.moderate * 0.6 +
      r.relationships.complex * 0.3 +
      r.queryPatterns.keyValue * 0.9 +
      r.queryPatterns.documentLookup * 1.0 +
      r.queryPatterns.complexQueries * 0.6 +
      r.queryPatterns.adhocQueries * 0.7 +
      r.queryPatterns.graphTraversal * 0.2 +
      r.queryPatterns.fullTextSearch * 0.6 +
      r.queryPatterns.timeSeries * 0.4 +
      (Math.min(r.scale.readVolume, 9) / 9) * 10 +
      (Math.min(r.scale.writeVolume, 8) / 8) * 10 +
```

```
      (Math.min(r.scale.dataVolume, 9) / 9) * 10 +
      r.cap.consistency * 0.6 +
      r.cap.availability * 0.8 +
      r.cap.partitionTolerance * 0.8 +
      r.special.globalDistribution * 0.7 +
      r.special.geospatial * 0.8 +
      r.special.streaming * 0.5 +
      r.special.analytics * 0.6 +
      r.special.caching * 0.3;

    // Key-Value (Redis, DynamoDB, etc.)
    this.databaseTypes.keyValue.score =
      r.schema.structured * 0.3 +
      r.schema.semiStructured * 0.6 +
      r.schema.unstructured * 0.5 +
      r.relationships.simple * 1.0 +
      r.relationships.moderate * 0.2 +
      r.relationships.complex * 0.0 +
      r.queryPatterns.keyValue * 1.0 +
      r.queryPatterns.documentLookup * 0.6 +
      r.queryPatterns.complexQueries * 0.1 +
      r.queryPatterns.adhocQueries * 0.1 +
      r.queryPatterns.graphTraversal * 0.0 +
      r.queryPatterns.fullTextSearch * 0.1 +
      r.queryPatterns.timeSeries * 0.3 +
      (Math.min(r.scale.readVolume, 10) / 10) * 10 +
      (Math.min(r.scale.writeVolume, 10) / 10) * 10 +
      (Math.min(r.scale.dataVolume, 8) / 8) * 10 +
      r.cap.consistency * 0.5 +
      r.cap.availability * 0.9 +
      r.cap.partitionTolerance * 0.9 +
      r.special.globalDistribution * 0.6 +
      r.special.geospatial * 0.2 +
      r.special.streaming * 0.7 +
      r.special.analytics * 0.2 +
      r.special.caching * 1.0;

    // Column-Family (Cassandra, HBase, etc.)
    this.databaseTypes.columnFamily.score =
      r.schema.structured * 0.6 +
      r.schema.semiStructured * 0.8 +
      r.schema.unstructured * 0.3 +
      r.relationships.simple * 0.9 +
      r.relationships.moderate * 0.5 +
      r.relationships.complex * 0.1 +
      r.queryPatterns.keyValue * 0.9 +
      r.queryPatterns.documentLookup * 0.7 +
      r.queryPatterns.complexQueries * 0.2 +
      r.queryPatterns.adhocQueries * 0.2 +
      r.queryPatterns.graphTraversal * 0.1 +
      r.queryPatterns.fullTextSearch * 0.2 +
      r.queryPatterns.timeSeries * 0.8 +
      (Math.min(r.scale.readVolume, 10) / 10) * 10 +
      (Math.min(r.scale.writeVolume, 10) / 10) * 10 +
      (Math.min(r.scale.dataVolume, 10) / 10) * 10 +
```

```
      r.cap.consistency * 0.5 +
      r.cap.availability * 0.9 +
      r.cap.partitionTolerance * 1.0 +
      r.special.globalDistribution * 0.9 +
      r.special.geospatial * 0.3 +
      r.special.streaming * 0.8 +
      r.special.analytics * 0.5 +
      r.special.caching * 0.2;

    // Graph Database (Neo4j, etc.)
    this.databaseTypes.graphDb.score =
      r.schema.structured * 0.5 +
      r.schema.semiStructured * 0.8 +
      r.schema.unstructured * 0.3 +
      r.relationships.simple * 0.5 +
      r.relationships.moderate * 0.8 +
      r.relationships.complex * 1.0 +
      r.queryPatterns.keyValue * 0.4 +
      r.queryPatterns.documentLookup * 0.5 +
      r.queryPatterns.complexQueries * 0.6 +
      r.queryPatterns.adhocQueries * 0.7 +
      r.queryPatterns.graphTraversal * 1.0 +
      r.queryPatterns.fullTextSearch * 0.4 +
      r.queryPatterns.timeSeries * 0.2 +
      (Math.min(r.scale.readVolume, 7) / 7) * 10 +
      (Math.min(r.scale.writeVolume, 6) / 6) * 10 +
      (Math.min(r.scale.dataVolume, 7) / 7) * 10 +
      r.cap.consistency * 0.7 +
      r.cap.availability * 0.6 +
      r.cap.partitionTolerance * 0.5 +
      r.special.globalDistribution * 0.4 +
      r.special.geospatial * 0.6 +
      r.special.streaming * 0.3 +
      r.special.analytics * 0.5 +
      r.special.caching * 0.1;

    // Search Engine (Elasticsearch, etc.) score calculation
    this.databaseTypes.searchEngine.score =
      r.schema.structured * 0.4 +
      r.schema.semiStructured * 0.9 +
      r.schema.unstructured * 1.0 +
      r.relationships.simple * 0.7 +
      r.relationships.moderate * 0.4 +
      r.relationships.complex * 0.1 +
      r.queryPatterns.keyValue * 0.5 +
      r.queryPatterns.documentLookup * 0.7 +
      r.queryPatterns.complexQueries * 0.3 +
      r.queryPatterns.adhocQueries * 0.9 +
      r.queryPatterns.graphTraversal * 0.1 +
      r.queryPatterns.fullTextSearch * 1.0 +
      r.queryPatterns.timeSeries * 0.3 +
      (Math.min(r.scale.readVolume, 9) / 9) * 10 +
      (Math.min(r.scale.writeVolume, 7) / 7) * 10 +
      (Math.min(r.scale.dataVolume, 8) / 8) * 10 +
      r.cap.consistency * 0.5 +
```

```
      r.cap.availability * 0.8 +
      r.cap.partitionTolerance * 0.8 +
      r.special.globalDistribution * 0.6 +
      r.special.geospatial * 0.8 +
      r.special.streaming * 0.4 +
      r.special.analytics * 0.8 +
      r.special.caching * 0.2;

    // Time-Series Database (InfluxDB, TimescaleDB, etc.)
    this.databaseTypes.timeSeriesDb.score =
      r.schema.structured * 0.7 +
      r.schema.semiStructured * 0.6 +
      r.schema.unstructured * 0.2 +
      r.relationships.simple * 0.9 +
      r.relationships.moderate * 0.3 +
      r.relationships.complex * 0.1 +
      r.queryPatterns.keyValue * 0.6 +
      r.queryPatterns.documentLookup * 0.4 +
      r.queryPatterns.complexQueries * 0.5 +
      r.queryPatterns.adhocQueries * 0.6 +
      r.queryPatterns.graphTraversal * 0.1 +
      r.queryPatterns.fullTextSearch * 0.1 +
      r.queryPatterns.timeSeries * 1.0 +
      (Math.min(r.scale.readVolume, 9) / 9) * 10 +
      (Math.min(r.scale.writeVolume, 10) / 10) * 10 +
      (Math.min(r.scale.dataVolume, 9) / 9) * 10 +
      r.cap.consistency * 0.6 +
      r.cap.availability * 0.8 +
      r.cap.partitionTolerance * 0.8 +
      r.special.globalDistribution * 0.5 +
      r.special.geospatial * 0.3 +
      r.special.streaming * 0.9 +
      r.special.analytics * 0.9 +
      r.special.caching * 0.3;

    // In-Memory Database (Redis, Memcached, etc.)
    this.databaseTypes.inMemoryDb.score =
      r.schema.structured * 0.4 +
      r.schema.semiStructured * 0.7 +
      r.schema.unstructured * 0.4 +
      r.relationships.simple * 1.0 +
      r.relationships.moderate * 0.3 +
      r.relationships.complex * 0.0 +
      r.queryPatterns.keyValue * 1.0 +
      r.queryPatterns.documentLookup * 0.7 +
      r.queryPatterns.complexQueries * 0.2 +
      r.queryPatterns.adhocQueries * 0.1 +
      r.queryPatterns.graphTraversal * 0.1 +
      r.queryPatterns.fullTextSearch * 0.1 +
      r.queryPatterns.timeSeries * 0.3 +
      (Math.min(r.scale.readVolume, 10) / 10) * 10 +
      (Math.min(r.scale.writeVolume, 10) / 10) * 10 +
      (Math.min(r.scale.dataVolume, 6) / 6) * 10 +
      r.cap.consistency * 0.6 +
      r.cap.availability * 0.9 +
```

```
        r.cap.partitionTolerance * 0.5 +
        r.special.globalDistribution * 0.4 +
        r.special.geospatial * 0.2 +
        r.special.streaming * 0.7 +
        r.special.analytics * 0.2 +
        r.special.caching * 1.0;

      // Multi-Model Database (ArangoDB, Cosmos DB, etc.)
      this.databaseTypes.multiModelDb.score =
        r.schema.structured * 0.7 +
        r.schema.semiStructured * 0.9 +
        r.schema.unstructured * 0.6 +
        r.relationships.simple * 0.8 +
        r.relationships.moderate * 0.8 +
        r.relationships.complex * 0.7 +
        r.queryPatterns.keyValue * 0.8 +
        r.queryPatterns.documentLookup * 0.9 +
        r.queryPatterns.complexQueries * 0.7 +
        r.queryPatterns.adhocQueries * 0.8 +
        r.queryPatterns.graphTraversal * 0.7 +
        r.queryPatterns.fullTextSearch * 0.6 +
        r.queryPatterns.timeSeries * 0.5 +
        (Math.min(r.scale.readVolume, 8) / 8) * 10 +
        (Math.min(r.scale.writeVolume, 8) / 8) * 10 +
        (Math.min(r.scale.dataVolume, 8) / 8) * 10 +
        r.cap.consistency * 0.7 +
        r.cap.availability * 0.8 +
        r.cap.partitionTolerance * 0.7 +
        r.special.globalDistribution * 0.8 +
        r.special.geospatial * 0.7 +
        r.special.streaming * 0.5 +
        r.special.analytics * 0.6 +
        r.special.caching * 0.4;

      return this;
    }

    // Get database recommendations based on scores
    getRecommendations(limit = 3) {
      this.calculateScores();

      // Flatten database types
      const recommendations = [
        { type: 'Traditional RDBMS', ...this.databaseTypes.rdbms.traditional },
        { type: 'Distributed SQL', ...this.databaseTypes.rdbms.distributed },
        { type: 'Document Database', ...this.databaseTypes.documentDb },
        { type: 'Key-Value Database', ...this.databaseTypes.keyValue },
        { type: 'Column-Family Database', ...this.databaseTypes.columnFamily },
        { type: 'Graph Database', ...this.databaseTypes.graphDb },
        { type: 'Search Engine', ...this.databaseTypes.searchEngine },
        { type: 'Time-Series Database', ...this.databaseTypes.timeSeriesDb },
        { type: 'In-Memory Database', ...this.databaseTypes.inMemoryDb },
        { type: 'Multi-Model Database', ...this.databaseTypes.multiModelDb },
      ];
```

```
    // Sort by score
    recommendations.sort((a, b) => b.score - a.score);

    // Take top N recommendations
    return recommendations.slice(0, limit);
  }

  // Generate a report with recommendations and explanations
  generateReport() {
    const recommendations = this.getRecommendations(3);
    const topScore = recommendations[0].score;

    // Start building the report
    let report = {
      requirements: this.requirements,
      recommendations: recommendations,
      explanation: `Based on your requirements, the recommended database types
are:\n\n`,
    };

    // Add explanations for top recommendations
    recommendations.forEach((rec, index) => {
      const scorePercentage = Math.round((rec.score / topScore) * 100);

      report.explanation += `${index + 1}. **${
        rec.type
      }** (${scorePercentage}% match)\n`;
      report.explanation += `   Examples: ${rec.examples.join(', ')}\n`;
      report.explanation += `   ${rec.description}\n\n`;

      // Add strengths for this database type
      report.explanation += `   **Strengths** for your use case:\n`;

      // Data characteristics
      if (
        this.requirements.schema.structured > 7 &&
        rec.type.includes('RDBMS')
      ) {
        report.explanation += `   - Strong support for your structured data
needs\n`;
      }
      if (
        this.requirements.schema.semiStructured > 7 &&
        (rec.type.includes('Document') || rec.type.includes('Multi-Model'))
      ) {
        report.explanation += `   - Excellent for your semi-structured data
requirements\n`;
      }

      // Relationships
      if (
        this.requirements.relationships.complex > 7 &&
        rec.type.includes('Graph')
      ) {
        report.explanation += `   - Optimized for your complex relationship data
```

```
model\n`;
      }

      // Query patterns
      if (
        this.requirements.queryPatterns.complexQueries > 7 &&
        (rec.type.includes('RDBMS') || rec.type.includes('SQL'))
      ) {
        report.explanation += `   - Strong support for your complex query
requirements\n`;
      }
      if (
        this.requirements.queryPatterns.fullTextSearch > 7 &&
        rec.type.includes('Search')
      ) {
        report.explanation += `   - Built specifically for your full-text search
needs\n`;
      }
      if (
        this.requirements.queryPatterns.timeSeries > 7 &&
        rec.type.includes('Time-Series')
      ) {
        report.explanation += `   - Optimized for your time-series data workload\n`;
      }

      // Scale
      if (
        this.requirements.scale.readVolume > 8 ||
        this.requirements.scale.writeVolume > 8
      ) {
        if (
          rec.type.includes('Key-Value') ||
          rec.type.includes('Column-Family')
        ) {
          report.explanation += `   - Handles your high throughput requirements
well\n`;
        }
      }

      // Special requirements
      if (
        this.requirements.special.globalDistribution > 7 &&
        (rec.type.includes('Distributed') || rec.type.includes('Multi-Model'))
      ) {
        report.explanation += `   - Good support for your global distribution
needs\n`;
      }
      if (
        this.requirements.special.caching > 7 &&
        rec.type.includes('In-Memory')
      ) {
        report.explanation += `   - Perfect for your caching requirements\n`;
      }

      report.explanation += `\n`;
```

```
    });

    // Add notes on hybrid approaches if relevant
    if (recommendations[0].score - recommendations[1].score < 50) {
      report.explanation += `**Note about hybrid approaches:**\n`;
      report.explanation += `Your requirements might benefit from a hybrid approach
using multiple database types. Consider:\n`;
      report.explanation += `- Using ${recommendations[0].type} as your primary
database\n`;
      report.explanation += `- Supplementing with ${recommendations[1].type} for
specific functionality\n`;

      if (
        this.requirements.special.caching > 6 &&
        !recommendations[0].type.includes('In-Memory')
      ) {
        report.explanation += `- Adding an In-Memory Database like Redis for
caching\n`;
      }
      if (
        this.requirements.queryPatterns.fullTextSearch > 6 &&
        !recommendations[0].type.includes('Search')
      ) {
        report.explanation += `- Integrating a Search Engine like Elasticsearch for
full-text search\n`;
      }
    }

    return report;
  }
}

// Example usage:
function databaseRecommendationExample() {
  const selector = new DatabaseSelectionFramework();

  // Social media application example
  selector
    // Data characteristics
    .rateRequirement('schema', 'structured', 4)
    .rateRequirement('schema', 'semiStructured', 8)
    .rateRequirement('schema', 'unstructured', 6)

    // Relationships
    .rateRequirement('relationships', 'simple', 3)
    .rateRequirement('relationships', 'moderate', 7)
    .rateRequirement('relationships', 'complex', 8)

    // Query patterns
    .rateRequirement('queryPatterns', 'keyValue', 7)
    .rateRequirement('queryPatterns', 'documentLookup', 8)
    .rateRequirement('queryPatterns', 'complexQueries', 6)
    .rateRequirement('queryPatterns', 'adhocQueries', 5)
    .rateRequirement('queryPatterns', 'graphTraversal', 9)
    .rateRequirement('queryPatterns', 'fullTextSearch', 8)
```

```
    .rateRequirement('queryPatterns', 'timeSeries', 4)

    // Scale
    .rateRequirement('scale', 'readVolume', 9)
    .rateRequirement('scale', 'writeVolume', 7)
    .rateRequirement('scale', 'dataVolume', 8)
    .rateRequirement('scale', 'growthRate', 7)

    // CAP
    .rateRequirement('cap', 'consistency', 5)
    .rateRequirement('cap', 'availability', 9)
    .rateRequirement('cap', 'partitionTolerance', 8)

    // Special requirements
    .rateRequirement('special', 'globalDistribution', 8)
    .rateRequirement('special', 'geospatial', 7)
    .rateRequirement('special', 'streaming', 6)
    .rateRequirement('special', 'analytics', 7)
    .rateRequirement('special', 'caching', 9);

  const report = selector.generateReport();
  console.log(report.explanation);
}
```

2. **Use Case-Based Recommendations**: Specific database recommendations for common application types.

```
// Use case-based database recommendations
const useCaseRecommendations = {
  // E-commerce platform
  eCommerce: {
    description:
      'Online shopping platform with product catalog, shopping cart, and order
processing',
    primaryDatabase: {
      type: 'Relational Database',
      examples: ['PostgreSQL', 'MySQL', 'SQL Server'],
      suitability: 'High',
      reason:
        'Strong transaction support for orders and inventory, complex queries for
reporting, and good data integrity',
    },
    secondaryDatabases: [
      {
        type: 'Search Engine',
        examples: ['Elasticsearch', 'Algolia'],
        suitability: 'High',
        reason: 'Fast and relevant product search with faceted navigation',
      },
      {
        type: 'Key-Value Store',
        examples: ['Redis', 'Memcached'],
        suitability: 'Medium',
        reason:
```

```
        'Session management, shopping cart data, and caching product info',
    },
    {
      type: 'Document Database',
      examples: ['MongoDB', 'DocumentDB'],
      suitability: 'Medium',
      reason: 'Product catalog with varying attributes across categories',
    },
  ],
  schema: {
    tables: [
      {
        name: 'users',
        description: 'Customer information and authentication',
        fields: [
          'id',
          'email',
          'password_hash',
          'name',
          'address',
          'created_at',
        ],
      },
      {
        name: 'products',
        description: 'Product catalog',
        fields: [
          'id',
          'name',
          'description',
          'price',
          'inventory_count',
          'category_id',
        ],
      },
      {
        name: 'orders',
        description: 'Customer orders',
        fields: [
          'id',
          'user_id',
          'status',
          'total_amount',
          'created_at',
          'updated_at',
        ],
      },
      {
        name: 'order_items',
        description: 'Line items for each order',
        fields: ['id', 'order_id', 'product_id', 'quantity', 'price'],
      },
    ],
  },
  implementationNotes: [
```

```
        'Use database transactions for order processing to maintain data consistency',
        'Implement inventory locking to prevent overselling',
        'Set up read replicas for high-traffic product catalog access',
        'Use Redis for shopping cart storage with appropriate TTL',
        'Consider sharding by product category for large catalogs',
      ],
    },

    // Social network
    socialNetwork: {
      description:
        'Platform for user connections, posts, and real-time interactions',
      primaryDatabase: {
        type: 'Graph Database',
        examples: ['Neo4j', 'Amazon Neptune', 'JanusGraph'],
        suitability: 'High',
        reason:
          'Optimized for relationship queries, friend suggestions, and network
analysis',
      },
      secondaryDatabases: [
        {
          type: 'Document Database',
          examples: ['MongoDB', 'DocumentDB', 'Firestore'],
          suitability: 'High',
          reason:
            'Flexible schema for user-generated content, posts, and comments',
        },
        {
          type: 'In-Memory Database',
          examples: ['Redis', 'Aerospike'],
          suitability: 'High',
          reason: 'Real-time features, activity feeds, and notification delivery',
        },
        {
          type: 'Search Engine',
          examples: ['Elasticsearch', 'Solr'],
          suitability: 'Medium',
          reason: 'Content search and user discovery',
        },
      ],
      schema: {
        nodes: [
          {
            type: 'User',
            properties: ['id', 'name', 'email', 'profile_pic', 'created_at'],
          },
          {
            type: 'Post',
            properties: ['id', 'content', 'created_at', 'privacy_level'],
          },
          {
            type: 'Comment',
            properties: ['id', 'content', 'created_at'],
          },
```

```
      ],
      relationships: [
        {
          type: 'FOLLOWS',
          from: 'User',
          to: 'User',
          properties: ['since'],
        },
        {
          type: 'POSTED',
          from: 'User',
          to: 'Post',
          properties: [],
        },
        {
          type: 'COMMENTED_ON',
          from: 'User',
          to: 'Post',
          properties: [],
        },
        {
          type: 'LIKED',
          from: 'User',
          to: 'Post',
          properties: ['created_at'],
        },
      ],
    },
    implementationNotes: [
      'Use graph database for social connections and feed generation',
      'Implement Redis sorted sets for activity feeds with time-based scoring',
      'Consider Cassandra for high-volume message storage',
      'Implement eventual consistency for non-critical operations like Like counts',
      'Use sharding strategies based on user ID for large user bases',
    ],
  },

  // IoT platform
  iotPlatform: {
    description:
      'System for collecting, storing, and analyzing data from IoT devices',
    primaryDatabase: {
      type: 'Time-Series Database',
      examples: ['InfluxDB', 'TimescaleDB', 'Amazon Timestream'],
      suitability: 'Very High',
      reason:
        'Optimized for high-volume time-stamped data with efficient storage and
time-based queries',
    },
    secondaryDatabases: [
      {
        type: 'Document Database',
        examples: ['MongoDB', 'Couchbase'],
        suitability: 'Medium',
        reason: 'Device metadata and configuration',
```

```
      },
      {
        type: 'Column-Family Store',
        examples: ['Apache Cassandra', 'ScyllaDB'],
        suitability: 'High',
        reason:
          'High write throughput for sensor data and horizontal scalability',
      },
      {
        type: 'In-Memory Database',
        examples: ['Redis', 'Apache Ignite'],
        suitability: 'Medium',
        reason: 'Real-time device state and recent readings',
      },
    ],
    schema: {
      timeSeries: [
        {
          measurement: 'device_readings',
          tags: ['device_id', 'sensor_type', 'location'],
          fields: ['value', 'battery_level'],
          timestamp: 'reading_time',
        },
      ],
      collections: [
        {
          name: 'devices',
          fields: [
            'id',
            'name',
            'type',
            'firmware_version',
            'last_connected',
            'location',
          ],
        },
        {
          name: 'alerts',
          fields: [
            'id',
            'device_id',
            'alert_type',
            'message',
            'severity',
            'created_at',
          ],
        },
      ],
    },
    implementationNotes: [
      'Use downsampling for historical data to optimize storage',
      'Implement data retention policies based on importance and regulatory
requirements',
      'Consider data partitioning by time periods (hour, day, month)',
      'Use specialized time-series functions for aggregations and trend analysis',
```

```
        'Implement real-time dashboards with in-memory caching for current values',
      ],
    },

    // Financial application
    financialSystem: {
      description:
        'Banking or trading platform with strict consistency and audit requirements',
      primaryDatabase: {
        type: 'Relational Database',
        examples: ['Oracle', 'PostgreSQL', 'SQL Server'],
        suitability: 'Very High',
        reason:
          'Strong ACID compliance, transaction support, and data integrity for
financial records',
      },
      secondaryDatabases: [
        {
          type: 'Time-Series Database',
          examples: ['InfluxDB', 'TimescaleDB', 'KDB+'],
          suitability: 'High',
          reason: 'Market data and price history',
        },
        {
          type: 'In-Memory Database',
          examples: ['Redis', 'VoltDB'],
          suitability: 'Medium',
          reason: 'Real-time processing and caching',
        },
        {
          type: 'NewSQL Database',
          examples: ['CockroachDB', 'Google Spanner', 'YugabyteDB'],
          suitability: 'High',
          reason: 'Horizontal scalability with ACID guarantees',
        },
      ],
      schema: {
        tables: [
          {
            name: 'accounts',
            description: 'Customer accounts',
            fields: [
              'id',
              'customer_id',
              'account_number',
              'type',
              'balance',
              'currency',
              'status',
            ],
          },
          {
            name: 'transactions',
            description: 'Financial transactions',
            fields: [
```

```
            'id',
            'account_id',
            'transaction_type',
            'amount',
            'currency',
            'description',
            'timestamp',
          ],
        },
        {
          name: 'audit_logs',
          description: 'Audit trail for regulatory compliance',
          fields: [
            'id',
            'entity_type',
            'entity_id',
            'action',
            'changes',
            'user_id',
            'timestamp',
            'ip_address',
          ],
        },
      ],
    },
    implementationNotes: [
      'Implement strict transaction isolation levels for financial data',
      'Use database-level encryption for sensitive data',
      'Set up point-in-time recovery for disaster scenarios',
      'Implement comprehensive audit logging for regulatory compliance',
      'Consider active-active clustering for high availability',
      'Use read replicas for reporting workloads to avoid impacting transaction
processing',
    ],
  },

  // Content management system
  contentManagementSystem: {
    description:
      'Platform for creating, managing, and delivering digital content',
    primaryDatabase: {
      type: 'Document Database',
      examples: ['MongoDB', 'Couchbase', 'Firestore'],
      suitability: 'High',
      reason:
        'Flexible schema for varied content types, good query capabilities, and
horizontal scaling',
    },
    secondaryDatabases: [
      {
        type: 'Search Engine',
        examples: ['Elasticsearch', 'Solr'],
        suitability: 'Very High',
        reason: 'Full-text search, faceted navigation, and relevance scoring',
      },
```

```
        {
          type: 'Key-Value Store',
          examples: ['Redis', 'DynamoDB'],
          suitability: 'Medium',
          reason: 'Caching rendered content and session management',
        },
        {
          type: 'Relational Database',
          examples: ['PostgreSQL', 'MySQL'],
          suitability: 'Medium',
          reason: 'User management, permissions, and structured metadata',
        },
      ],
      schema: {
        collections: [
          {
            name: 'content',
            fields: [
              'id',
              'title',
              'body',
              'author_id',
              'content_type',
              'status',
              'tags',
              'created_at',
              'updated_at',
              'published_at',
            ],
          },
          {
            name: 'assets',
            fields: [
              'id',
              'content_id',
              'type',
              'file_path',
              'file_size',
              'metadata',
              'created_at',
            ],
          },
          {
            name: 'users',
            fields: [
              'id',
              'username',
              'email',
              'role',
              'permissions',
              'created_at',
            ],
          },
        ],
      },
```

```
    implementationNotes: [
      'Use document database for flexible content models',
      'Implement denormalized data models for efficient content retrieval',
      'Set up Elasticsearch for full-text search with custom analyzers and relevance
tuning',
      'Use Redis for caching frequently accessed content',
      'Consider content versioning strategies for editorial workflows',
      'Implement CDN integration for asset delivery',
    ],
  },
};

// Function to get recommendations for a specific use case
function getUseCaseRecommendation(useCase) {
  if (!useCaseRecommendations[useCase]) {
    return {
      error: 'Use case not found',
      availableUseCases: Object.keys(useCaseRecommendations),
    };
  }

  return {
    useCase: useCase,
    ...useCaseRecommendations[useCase],
  };
}

// Example usage
function useCaseRecommendationExample() {
  const recommendation = getUseCaseRecommendation('iotPlatform');
  console.log(`Database Recommendations for ${recommendation.useCase}:`);
  console.log(`\nDescription: ${recommendation.description}`);

  console.log(`\nPrimary Database: ${recommendation.primaryDatabase.type}`);
  console.log(
    `Examples: ${recommendation.primaryDatabase.examples.join(', ')}`
  );
  console.log(`Suitability: ${recommendation.primaryDatabase.suitability}`);
  console.log(`Reason: ${recommendation.primaryDatabase.reason}`);

  console.log('\nSecondary Databases:');
  recommendation.secondaryDatabases.forEach((db) => {
    console.log(`- ${db.type} (${db.suitability}): ${db.reason}`);
    console.log(`  Examples: ${db.examples.join(', ')}`);
  });

  console.log('\nImplementation Notes:');
  recommendation.implementationNotes.forEach((note) => {
    console.log(`- ${note}`);
  });
}
```

Performance Optimization Checklists

1. **Relational Database Optimization Checklist**: A comprehensive guide to optimize relational databases.

```
// Relational database optimization checklist
const rdbmsOptimizationChecklist = {
  schemaDesign: [
    {
      name: 'Proper normalization level',
      description:
        'Ensure appropriate level of normalization (usually 3NF) with strategic
denormalization for performance',
      priority: 'High',
      impact: 'High',
      checkQuery: `
        -- Look for tables with redundant data
        SELECT table_name, column_name
        FROM information_schema.columns
        WHERE table_schema = 'your_schema'
        AND column_name LIKE '%id' OR column_name LIKE '%_name'
        ORDER BY table_name, column_name;
      `,
    },
    {
      name: 'Appropriate data types',
      description:
        'Use smallest appropriate data types for columns to reduce storage and
improve performance',
      priority: 'High',
      impact: 'Medium',
      checkQuery: `
        -- Check for overallocated data types
        SELECT
          table_name,
          column_name,
          data_type,
          character_maximum_length
        FROM information_schema.columns
        WHERE
          (data_type = 'varchar' AND character_maximum_length > 1000)
          OR (data_type = 'int' AND numeric_precision > 10)
        ORDER BY character_maximum_length DESC, numeric_precision DESC;
      `,
    },
    {
      name: 'Index strategy',
      description:
        'Create appropriate indexes for query patterns while avoiding over-
indexing',
      priority: 'Very High',
      impact: 'Very High',
      checkQuery: `
  -- Find missing indexes based on foreign keys
  SELECT
    tc.table_schema,
    tc.table_name,
```

```
      kcu.column_name,
      ccu.table_schema AS foreign_table_schema,
      ccu.table_name AS foreign_table_name,
      ccu.column_name AS foreign_column_name
    FROM
      information_schema.table_constraints AS tc
      JOIN information_schema.key_column_usage AS kcu
        ON tc.constraint_name = kcu.constraint_name
        AND tc.table_schema = kcu.table_schema
      JOIN information_schema.constraint_column_usage AS ccu
        ON ccu.constraint_name = tc.constraint_name
        AND ccu.table_schema = tc.table_schema
    WHERE tc.constraint_type = 'FOREIGN KEY'
    AND NOT EXISTS (
      SELECT 1 FROM information_schema.statistics
      WHERE table_schema = tc.table_schema
      AND table_name = tc.table_name
      AND column_name = kcu.column_name
      AND seq_in_index = 1 -- This column should be first in an index
    )
    ORDER BY tc.table_name;
  `,
      },
      {
        name: 'Constraint enforcement',
        description:
          'Implement appropriate constraints (PKs, FKs, unique, check) to ensure data
integrity',
        priority: 'High',
        impact: 'Medium',
        checkQuery: `
      -- Find tables without primary keys
      SELECT table_schema, table_name
      FROM information_schema.tables
      WHERE table_type = 'BASE TABLE'
      AND table_schema NOT IN ('pg_catalog', 'information_schema')
      AND table_name NOT IN (
        SELECT tc.table_name
        FROM information_schema.table_constraints tc
        WHERE tc.constraint_type = 'PRIMARY KEY'
        AND tc.table_schema = tables.table_schema
      )
      ORDER BY table_schema, table_name;
    `,
      },
    ],

    queryOptimization: [
      {
        name: 'Optimal query patterns',
        description: 'Review and optimize frequently-run queries for efficiency',
        priority: 'Very High',
        impact: 'Very High',
        checkQuery: `
      -- Find slow queries (PostgreSQL)
```

```
      SELECT
        query,
        calls,
        total_time,
        mean_time,
        rows
      FROM pg_stat_statements
      ORDER BY mean_time DESC
      LIMIT 20;

      -- Or for MySQL:
      -- SELECT * FROM performance_schema.events_statements_summary_by_digest
      -- ORDER BY sum_timer_wait DESC LIMIT 20;
    `,
      },
      {
        name: 'Proper use of JOINs',
        description:
          'Use appropriate join types and join order for optimal performance',
        priority: 'High',
        impact: 'High',
        checkQuery: `
      -- Check query plans for inefficient joins (run EXPLAIN on problematic queries)
      EXPLAIN ANALYZE
      SELECT * FROM large_table a
      JOIN medium_table b ON a.id = b.large_id
      JOIN small_table c ON b.id = c.medium_id
      WHERE c.some_column = 'some_value';
    `,
      },
      {
        name: 'Efficient WHERE clauses',
        description: 'Ensure WHERE clauses are index-friendly and filter early',
        priority: 'High',
        impact: 'High',
        checkQuery: `
      -- Example of checking if a specific query uses indexes effectively
      EXPLAIN ANALYZE
      SELECT * FROM your_table
      WHERE indexed_column = 'some_value'
      AND non_indexed_column = 'other_value';
    `,
      },
      {
        name: 'Pagination implementation',
        description: 'Use efficient pagination techniques for large result sets',
        priority: 'Medium',
        impact: 'High',
        checkQuery: `
      -- Check for queries that return large result sets without limits
      -- PostgreSQL
      SELECT
        query,
        rows
      FROM pg_stat_statements
```

```
      WHERE rows > 1000
      AND query NOT ILIKE '%LIMIT%'
      ORDER BY rows DESC
      LIMIT 10;
    `,
      },
    ],

    indexing: [
      {
        name: 'Index utilization',
        description:
          'Ensure indexes are being used by queries and remove unused indexes',
        priority: 'Very High',
        impact: 'Very High',
        checkQuery: `
      -- PostgreSQL: Find unused indexes
      SELECT
        s.schemaname,
        s.relname AS tablename,
        s.indexrelname AS indexname,
        pg_size_pretty(pg_relation_size(s.indexrelid)) AS index_size,
        s.idx_scan AS index_scans
      FROM pg_catalog.pg_stat_user_indexes s
      JOIN pg_catalog.pg_index i ON s.indexrelid = i.indexrelid
      WHERE s.idx_scan = 0 -- No scans
      AND NOT i.indisprimary -- Not a primary key
      AND NOT i.indisunique -- Not a unique constraint
      ORDER BY pg_relation_size(s.indexrelid) DESC;
    `,
      },
      {
        name: 'Composite indexes',
        description:
          'Create composite indexes for multi-column queries with correct column
order',
        priority: 'High',
        impact: 'High',
        checkQuery: `
      -- Find queries that might benefit from composite indexes (simplified)
      -- This would typically be done by reviewing query plans
      EXPLAIN ANALYZE
      SELECT * FROM your_table
      WHERE column1 = 'value1'
      AND column2 = 'value2'
      ORDER BY column3;
    `,
      },
      {
        name: 'Covering indexes',
        description: 'Use covering indexes for high-performance queries',
        priority: 'Medium',
        impact: 'High',
        checkQuery: `
      -- Check if a query could benefit from a covering index
```

```
    EXPLAIN ANALYZE
    SELECT column1, column2, column3
    FROM your_table
    WHERE column1 = 'value';
  `,
    },
    {
      name: 'Index maintenance',
      description:
        'Regularly rebuild and optimize indexes to prevent fragmentation',
      priority: 'Medium',
      impact: 'Medium',
      checkQuery: `
    -- PostgreSQL: Check for index bloat
    SELECT
      schemaname, tablename, reltuples::bigint, relpages::bigint, otta,
      ROUND(CASE WHEN otta=0 THEN 0.0 ELSE sml.relpages/otta::numeric END,1) AS
tbloat,
      CASE WHEN relpages < otta THEN 0 ELSE relpages::bigint - otta END AS
wastedpages,
      CASE WHEN relpages < otta THEN 0 ELSE bs*(relpages-otta)::bigint END AS
wastedbytes
    FROM (
      SELECT
        schemaname, tablename, cc.reltuples, cc.relpages, bs,
        CEIL((cc.reltuples*((datahdr+ma-
          (CASE WHEN datahdr%ma=0 THEN ma ELSE datahdr%ma END))+nullhdr2+4))/(bs-
20::float)) AS otta
      FROM (
        SELECT
          ma,bs,schemaname,tablename,
          (datawidth+(hdr+ma-(case when hdr%ma=0 THEN ma ELSE hdr%ma END)))::numeric
AS datahdr,
          (maxfracsum*(nullhdr+ma-(case when nullhdr%ma=0 THEN ma ELSE nullhdr%ma
END))) AS nullhdr2
        FROM (
          SELECT
            schemaname, tablename, hdr, ma, bs,
            SUM((1-null_frac)*avg_width) AS datawidth,
            MAX(null_frac) AS maxfracsum,
            hdr+(1+(count(*)/8)) AS nullhdr
          FROM pg_stats CROSS JOIN (
            SELECT
              (SELECT current_setting('block_size')::numeric) AS bs,
              CASE WHEN SUBSTRING(SPLIT_PART(v, ' ', 2) FROM '#"[0-9]+.[0-9]+#"%'
for '#')
                IS NULL THEN 8 ELSE SUBSTRING(SPLIT_PART(v, ' ', 2) FROM '#"[0-9]+.
[0-9]+#"%' for '#')::int END AS hdr,
              CASE WHEN v ~ 'mingw32' OR v ~ '64-bit' THEN 8 ELSE 4 END AS ma
            FROM (SELECT version() AS v) AS foo
          ) AS constants
          GROUP BY 1,2,3,4,5
        ) AS foo
      ) AS foo
      CROSS JOIN (
```

```
        SELECT
          c.oid,n.nspname AS schemaname,c.relname AS tablename,
          c.reltuples,c.relpages
        FROM pg_class c LEFT JOIN pg_namespace n ON n.oid = c.relnamespace
        WHERE nspname NOT IN ('pg_catalog','information_schema')
        AND c.relkind='r'
      ) AS cc
      WHERE schemaname = cc.schemaname AND tablename = cc.relname
    ) AS sml
    WHERE sml.relpages - otta > 128
    ORDER BY wastedbytes DESC LIMIT 20;
  `,
    },
  ],

  serverConfiguration: [
    {
      name: 'Memory allocation',
      description:
        'Configure appropriate memory settings for buffer cache, shared buffers,
etc.',
      priority: 'High',
      impact: 'High',
      checkQuery: `
    -- PostgreSQL: Check current memory-related settings
    SELECT name, setting, unit, context
    FROM pg_settings
    WHERE name IN (
      'shared_buffers', 'work_mem', 'maintenance_work_mem',
      'effective_cache_size', 'max_connections'
    );
  `,
    },
    {
      name: 'Connection pooling',
      description: 'Implement connection pooling to reduce connection overhead',
      priority: 'High',
      impact: 'Medium',
      checkQuery: `
    -- Check current connections
    SELECT count(*) as active_connections FROM pg_stat_activity;

    -- Check connection settings
    SELECT name, setting, unit, context
    FROM pg_settings
    WHERE name IN ('max_connections', 'superuser_reserved_connections');
  `,
    },
    {
      name: 'Disk I/O configuration',
      description: 'Optimize storage for database workload (RAID, SSDs, etc.)',
      priority: 'Medium',
      impact: 'High',
      checkQuery: `
    -- Check I/O statistics (PostgreSQL)
```

```
      SELECT
        pg_stat_reset_shared('bgwriter'),
        pg_stat_reset();

      -- Wait a few minutes, then check
      SELECT * FROM pg_stat_bgwriter;

      -- Check for tables with high I/O
      SELECT
        schemaname, relname,
        heap_blks_read, heap_blks_hit,
        idx_blks_read, idx_blks_hit,
        toast_blks_read, toast_blks_hit,
        tidx_blks_read, tidx_blks_hit
      FROM pg_statio_user_tables
      ORDER BY heap_blks_read + idx_blks_read DESC
      LIMIT 20;
    `,
      },
      {
        name: 'WAL/transaction log settings',
        description: 'Configure appropriate write-ahead logging settings',
        priority: 'Medium',
        impact: 'Medium',
        checkQuery: `
      -- PostgreSQL: Check WAL settings
      SELECT name, setting, unit, context
      FROM pg_settings
      WHERE name LIKE 'wal%' OR name IN ('checkpoint_timeout',
  'checkpoint_completion_target');
    `,
      },
    ],

    maintenancePractices: [
      {
        name: 'Statistics collection',
        description: 'Ensure statistics are up-to-date for query optimizer',
        priority: 'High',
        impact: 'High',
        checkQuery: `
      -- PostgreSQL: Check last analyze time
      SELECT
        schemaname,
        relname,
        last_analyze,
        last_autoanalyze
      FROM pg_stat_user_tables
      WHERE (last_analyze IS NULL AND last_autoanalyze IS NULL)
      OR last_analyze < NOW() - INTERVAL '1 week'
      ORDER BY relname;
    `,
      },
      {
        name: 'Regular VACUUM/maintenance',
```

```
        description:
          'Implement regular cleanup to reclaim space and prevent bloat',
        priority: 'High',
        impact: 'Medium',
        checkQuery: `
      -- PostgreSQL: Check last vacuum time
      SELECT
        schemaname,
        relname,
        last_vacuum,
        last_autovacuum,
        n_dead_tup,
        n_live_tup
      FROM pg_stat_user_tables
      ORDER BY n_dead_tup DESC
      LIMIT 20;
    `,
      },
      {
        name: 'Monitoring and alerts',
        description:
          'Set up monitoring system with alerts for database performance issues',
        priority: 'High',
        impact: 'Medium',
        checkQuery: `
      -- PostgreSQL: Get current activity
      SELECT
        pid,
        usename,
        application_name,
        client_addr,
        state,
        query_start,
        NOW() - query_start AS duration,
        wait_event_type,
        wait_event,
        query
      FROM pg_stat_activity
      WHERE state <> 'idle'
      ORDER BY duration DESC;
    `,
      },
      {
        name: 'Backup strategy',
        description: 'Implement comprehensive backup and recovery strategy',
        priority: 'Very High',
        impact: 'Medium',
        checkQuery: `
      -- PostgreSQL: Check WAL archiving status
      SELECT
        pg_current_wal_lsn(),
        pg_walfile_name(pg_current_wal_lsn()),
        pg_wal_lsn_diff(pg_current_wal_lsn(), '0/0') AS bytes_since_start;

      -- Check backup history if using pg_basebackup
```

```
    -- This would typically be done outside the database
  `,
    },
  ],
};

// Function to run checklist and generate optimization report
async function generateRDBMSOptimizationReport(db, options = {}) {
  const {
    includeCategories = [
      'schemaDesign',
      'queryOptimization',
      'indexing',
      'serverConfiguration',
      'maintenancePractices',
    ],
    priorityThreshold = 'Medium', // 'Low', 'Medium', 'High', 'Very High'
    runQueries = false,
  } = options;

  const priorityLevels = {
    Low: 1,
    Medium: 2,
    High: 3,
    'Very High': 4,
  };

  const minimumPriority = priorityLevels[priorityThreshold];

  // Start building the report
  let report = {
    databaseName: await getDatabaseName(db),
    generatedAt: new Date(),
    summary: {
      totalChecks: 0,
      highPriorityIssues: 0,
      mediumPriorityIssues: 0,
      lowPriorityIssues: 0,
    },
    categories: {},
  };

  // Helper function to get DB name
  async function getDatabaseName(db) {
    try {
      const result = await db.query('SELECT current_database() AS dbname');
      return result.rows[0].dbname;
    } catch (error) {
      return 'unknown';
    }
  }

  // Process each category
  for (const category of includeCategories) {
    if (!rdbmsOptimizationChecklist[category]) continue;
```

```
    report.categories[category] = {
      checks: [],
    };

    // Process each check in the category
    for (const check of rdbmsOptimizationChecklist[category]) {
      // Skip checks below priority threshold
      if (priorityLevels[check.priority] < minimumPriority) continue;

      report.summary.totalChecks++;

      const checkResult = {
        name: check.name,
        description: check.description,
        priority: check.priority,
        impact: check.impact,
        queryResults: null,
        status: 'pending', // pending, passed, warning, failed
        recommendations: [],
      };

      // Run the check query if enabled
      if (runQueries && check.checkQuery) {
        try {
          const result = await db.query(check.checkQuery);
          checkResult.queryResults = result.rows;

          // Simple heuristic: if query returns any rows, it might indicate an issue
          if (result.rows.length > 0) {
            checkResult.status = 'warning';

            // Increment the appropriate counter
            if (check.priority === 'High' || check.priority === 'Very High') {
              report.summary.highPriorityIssues++;
            } else if (check.priority === 'Medium') {
              report.summary.mediumPriorityIssues++;
            } else {
              report.summary.lowPriorityIssues++;
            }

            // Generate generic recommendations based on check name
            if (check.name.includes('index')) {
              checkResult.recommendations.push(
                'Review index strategy for the affected tables',
                'Consider adding or removing indexes based on query patterns'
              );
            } else if (check.name.includes('query')) {
              checkResult.recommendations.push(
                'Analyze query execution plans with EXPLAIN ANALYZE',
                'Optimize problematic queries by rewriting or adding appropriate
indexes'
              );
            } else if (check.name.includes('maintenance')) {
              checkResult.recommendations.push(
```

```
                'Implement regular maintenance jobs for VACUUM, ANALYZE, REINDEX',
                'Set up monitoring and alerts for database health'
              );
            }
          } else {
            checkResult.status = 'passed';
          }
        } catch (error) {
          checkResult.status = 'failed';
          checkResult.error = error.message;
        }
      }

      report.categories[category].checks.push(checkResult);
    }
  }

  // Generate overall summary
  report.overallAssessment = generateOverallAssessment(report);

  return report;

  // Helper function to generate overall assessment
  function generateOverallAssessment(report) {
    const totalIssues =
      report.summary.highPriorityIssues +
      report.summary.mediumPriorityIssues +
      report.summary.lowPriorityIssues;

    if (report.summary.highPriorityIssues > 5) {
      return 'Critical: Significant high-priority issues detected that should be
addressed immediately';
    } else if (report.summary.highPriorityIssues > 0) {
      return 'Warning: Some high-priority issues detected that require attention';
    } else if (report.summary.mediumPriorityIssues > 5) {
      return 'Needs Improvement: Multiple medium-priority issues detected';
    } else if (totalIssues > 0) {
      return 'Fair: Some minor issues detected, but generally good performance';
    } else {
      return 'Excellent: No issues detected in the checked categories';
    }
  }
}
```

2. **NoSQL Database Optimization Checklist**: Optimization guidelines for various NoSQL database types.

```
// NoSQL database optimization checklist
const nosqlOptimizationChecklist = {
  documentDatabase: {
    dataModeling: [
      {
        name: 'Document structure',
        description: 'Design documents to support common access patterns',
```

```
        priority: 'Very High',
        impact: 'High',
        checkMethod: 'Review document schemas and query patterns',
        bestPractices: [
          'Embed related data that is frequently accessed together',
          'Keep document size under 16MB (MongoDB limit)',
          'Use references for large objects or rarely accessed data',
          'Consider read/write ratios when deciding between embedding and
referencing',
        ],
      },
      {
        name: 'Field naming consistency',
        description: 'Use consistent field names across documents',
        priority: 'Medium',
        impact: 'Medium',
        checkMethod: 'Run aggregation to identify inconsistent field names',
        checkCode: `
          // MongoDB: Find field name variations
          db.collection.aggregate([
            { $project: { fieldNames: { $objectToArray: "$$ROOT" } } },
            { $unwind: "$fieldNames" },
            { $group: { _id: "$fieldNames.k", count: { $sum: 1 } } },
            { $sort: { count: -1 } }
          ])
        `,
        bestPractices: [
          'Standardize field naming conventions',
          'Use camelCase or snake_case consistently',
          'Document your schema design and naming patterns',
          'Consider using a schema validation tool',
        ],
      },
      {
        name: 'Denormalization strategy',
        description:
          'Appropriate duplication of data to optimize read performance',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Analyze query patterns and document structure',
        bestPractices: [
          'Duplicate data that is frequently read but rarely updated',
          'Implement a strategy to handle updates to duplicated data',
          'Monitor storage usage from denormalization',
          'Consider materialized views for complex aggregation results',
        ],
      },
    ],
    indexing: [
      {
        name: 'Index coverage',
        description:
          'Ensure common queries are supported by appropriate indexes',
        priority: 'Very High',
        impact: 'Very High',
```

```
        checkMethod: 'Analyze query performance and explain plans',
        checkCode: `
          // MongoDB: Get current indexes
          db.collection.getIndexes()

          // Check if a query uses an index
          db.collection.find({ field: "value" }).explain("executionStats")
        `,
        bestPractices: [
          'Create indexes for fields in query filter conditions',
          'Use compound indexes for queries with multiple filter conditions',
          'Create indexes to support sorting operations',
          'Use covered queries where possible (include all fields in the index)',
        ],
      },
      {
        name: 'Index size and memory usage',
        description: 'Monitor index size and memory usage',
        priority: 'Medium',
        impact: 'High',
        checkMethod: 'Check index size and memory statistics',
        checkCode: `
          // MongoDB: Check index sizes
          db.collection.stats().indexSizes

          // Check index usage statistics
          db.collection.aggregate([
            { $indexStats: { } }
          ])
        `,
        bestPractices: [
          'Remove unused indexes',
          'Monitor working set size vs available memory',
          'Consider partial indexes for large collections with specific queries',
          'Use sparse indexes for fields that exist only in some documents',
        ],
      },
    ],
    queryOptimization: [
      {
        name: 'Query efficiency',
        description: 'Optimize queries for performance',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Review slow query logs and explain plans',
        checkCode: `
          // MongoDB: Check for slow queries
          db.adminCommand({ getLog: "slowquery" })

          // Analyze a specific query
          db.collection.find({ field: "value" }).explain("executionStats")
        `,
        bestPractices: [
          'Use specific queries instead of generic ones',
          'Avoid negation operators ($ne, $nin) when possible',
```

```
          'Limit the number of documents returned',
          'Use projection to return only needed fields',
        ],
      },
      {
        name: 'Aggregation pipeline optimization',
        description: 'Optimize complex aggregation queries',
        priority: 'Medium',
        impact: 'High',
        checkMethod: 'Review aggregation performance',
        checkCode: `
          // MongoDB: Explain an aggregation pipeline
          db.collection.aggregate([
            { $match: { field: "value" } },
            { $group: { _id: "$category", count: { $sum: 1 } } }
          ], { explain: true })
        `,
        bestPractices: [
          'Use $match early in the pipeline to reduce documents processed',
          'Place $sort operations after $match and before $limit',
          'Use $project to limit fields as early as possible',
          'Consider using $merge or $out for large result sets',
        ],
      },
    ],
    shardingStrategy: [
      {
        name: 'Shard key selection',
        description: 'Choose appropriate shard key for data distribution',
        priority: 'Very High',
        impact: 'Very High',
        checkMethod:
          'Review current sharding configuration and chunk distribution',
        checkCode: `
          // MongoDB: Check current sharding status
          sh.status()

          // Check chunk distribution
          db.getSiblingDB("config").chunks.aggregate([
            { $group: { _id: "$shard", count: { $sum: 1 } } }
          ])
        `,
        bestPractices: [
          'Choose shard keys with high cardinality to distribute data evenly',
          'Avoid monotonically increasing shard keys (like timestamps) to prevent
hot shards',
          'Consider compound shard keys for better distribution',
          'Ensure common queries include the shard key for targeted routing',
        ],
      },
      {
        name: 'Chunk size and balance',
        description: 'Monitor and optimize chunk distribution',
        priority: 'Medium',
        impact: 'High',
```

```
        checkMethod: 'Check chunk distribution and migration statistics',
        bestPractices: [
          'Monitor chunk sizes and migrations',
          'Adjust chunk size based on document size and shard capacity',
          'Plan for regular balancing operations during off-peak hours',
          'Be aware of jumbo chunks that cannot be moved automatically',
        ],
      },
    ],
  },
  keyValueDatabase: {
    keyDesign: [
      {
        name: 'Key naming conventions',
        description: 'Implement consistent and meaningful key naming',
        priority: 'High',
        impact: 'Medium',
        checkMethod: 'Review key patterns',
        checkCode: `
          // Redis: Sample keys to review patterns
          SCAN 0 COUNT 100

          // Get key statistics
          INFO keyspace
        `,
        bestPractices: [
          'Use namespaces in keys (e.g., "user:1000:profile")',
          'Include entity type in key names',
          'Consider data lifetime in key naming (temp:, perm:)',
          'Ensure keys are reasonably short but descriptive',
        ],
      },
      {
        name: 'Key distribution',
        description: 'Ensure even distribution of keys for sharded deployments',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Analyze key distribution across cluster',
        bestPractices: [
          'Avoid key patterns that hash to the same slot in Redis Cluster',
          'Use multi-part keys with consistent hash components',
          'Include high-cardinality values in keys',
          'Consider using hash tags for related keys ({user:1000}:profile)',
        ],
      },
    ],
    dataStructures: [
      {
        name: 'Data structure selection',
        description: 'Use appropriate Redis data structures',
        priority: 'Very High',
        impact: 'High',
        checkMethod: 'Review key types and access patterns',
        checkCode: `
          // Redis: Check data types for keys
```

```
          SCAN 0 COUNT 100 TYPE string
          SCAN 0 COUNT 100 TYPE hash
          SCAN 0 COUNT 100 TYPE list
          SCAN 0 COUNT 100 TYPE set
          SCAN 0 COUNT 100 TYPE zset
        `,
        bestPractices: [
          'Use Strings for simple values and small objects (serialized)',
          'Use Hashes for object storage with field access',
          'Use Lists for queues and time-series data',
          'Use Sets for unique collections and relationship modeling',
          'Use Sorted Sets for ranked data and time-based access',
        ],
      },
      {
        name: 'Memory efficiency',
        description: 'Optimize memory usage for data structures',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Analyze memory usage',
        checkCode: `
          // Redis: Get memory statistics
          INFO memory

          // Find big keys
          redis-cli --bigkeys

          // Check memory usage of specific key
          MEMORY USAGE keyname
        `,
        bestPractices: [
          'Prefer Hashes over individual keys for objects',
          'Use Redis integer encoding when possible',
          'Set appropriate expiration times (TTL) for temporary data',
          'Consider compression for large values',
        ],
      },
    ],
    caching: [
      {
        name: 'Expiration policy',
        description: 'Implement appropriate expiration for cached data',
        priority: 'High',
        impact: 'Medium',
        checkMethod: 'Review TTL settings',
        checkCode: `
          // Redis: Find keys without expiration
          redis-cli --scan --pattern '*' | xargs -L 100 redis-cli TTL | grep -B 1 -E
'(^-1$)'
        `,
        bestPractices: [
          'Set TTL based on data volatility and access patterns',
          'Use different TTLs for different types of data',
          'Implement cache invalidation strategy for updates',
          'Consider using EXPIRE AT for time-of-day based expiration',
```

```
        ],
      },
      {
        name: 'Eviction policy',
        description:
          'Configure appropriate eviction when memory limit is reached',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Check Redis configuration',
        checkCode: `
          // Redis: Check maxmemory and policy
          CONFIG GET maxmemory
          CONFIG GET maxmemory-policy
        `,
        bestPractices: [
          'Set maxmemory limit to prevent swapping',
          'Choose appropriate policy: allkeys-lru for cache, volatile-lru for mixed
use',
          'Monitor eviction statistics with INFO stats',
          'Consider using volatile-ttl for data with explicit expiration',
        ],
      },
    ],
    performance: [
      {
        name: 'Connection management',
        description: 'Optimize client connections',
        priority: 'Medium',
        impact: 'Medium',
        checkMethod: 'Monitor client connections',
        checkCode: `
          // Redis: Check client connections
          INFO clients

          // Get client list
          CLIENT LIST
        `,
        bestPractices: [
          'Use connection pooling in clients',
          'Monitor and limit the number of connections',
          'Set appropriate timeout values',
          'Use pipelining for bulk operations',
        ],
      },
      {
        name: 'Persistence configuration',
        description: 'Optimize persistence settings for workload',
        priority: 'Medium',
        impact: 'High',
        checkMethod: 'Review persistence configuration',
        checkCode: `
          // Redis: Check persistence config
          CONFIG GET save
          CONFIG GET appendonly
          CONFIG GET appendfsync
```

```
        `,
        bestPractices: [
          'Choose appropriate persistence model: RDB, AOF, or both',
          'Configure save frequency based on data importance and performance needs',
          'Consider using appendfsync everysec for a balance of performance and
durability',
          'Use background save to minimize impact on performance',
          'Monitor the background save process statistics',
        ],
      },
      {
        name: 'Lua script usage',
        description: 'Optimize server-side scripting',
        priority: 'Medium',
        impact: 'Medium',
        checkMethod: 'Review Lua script usage',
        checkCode: `
    // Redis: List stored scripts
    SCRIPT EXISTS <sha1-hash>

    // Get script debug mode
    CONFIG GET lua-time-limit
  `,
        bestPractices: [
          'Use EVAL for complex atomic operations',
          'Load scripts with SCRIPT LOAD for reuse',
          'Keep scripts simple and efficient',
          'Be aware of script execution timeout (lua-time-limit)',
        ],
      },
    ],
  },
  columnFamilyDatabase: {
    dataModeling: [
      {
        name: 'Column family design',
        description: 'Optimize column family structure for access patterns',
        priority: 'Very High',
        impact: 'Very High',
        checkMethod: 'Review table schemas and query patterns',
        checkCode: `
    // Cassandra: Describe keyspace
    DESCRIBE KEYSPACE your_keyspace;

    // Describe table structure
    DESCRIBE TABLE your_table;
  `,
        bestPractices: [
          'Design tables for specific query patterns',
          'Group related columns that are accessed together in the same column
family',
          'Keep column family count low to minimize overhead',
          'Use wide rows for time-series or event data',
        ],
      },
```

```
      {
        name: 'Partition key selection',
        description: 'Choose partition keys that distribute data evenly',
        priority: 'Very High',
        impact: 'Very High',
        checkMethod: 'Analyze data distribution',
        checkCode: `
    // Cassandra: Get token distribution
    nodetool describering your_keyspace

    // Check table size per node
    nodetool tablehistograms your_keyspace your_table
  `,
        bestPractices: [
          'Select partition keys with high cardinality',
          'Avoid keys that create hotspots',
          'Keep partition size between 10MB and 100MB',
          'Consider composite partition keys for better distribution',
        ],
      },
      {
        name: 'Clustering key design',
        description: 'Optimize row ordering within partitions',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Review query patterns and data model',
        bestPractices: [
          'Order clustering keys based on query patterns',
          'Use clustering keys for range queries',
          'Consider the most common sorting requirements',
          'Be mindful of the impact of DESC/ASC ordering',
        ],
      },
    ],
    queriesAndWrites: [
      {
        name: 'Query efficiency',
        description: 'Ensure queries follow efficient patterns',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Review CQL queries and trace execution',
        checkCode: `
    // Cassandra: Enable tracing for a query
    TRACING ON;
    SELECT * FROM your_table WHERE partition_key = value;
    TRACING OFF;
  `,
        bestPractices: [
          'Always include the partition key in queries',
          'Avoid using ALLOW FILTERING',
          'Create tables to match query patterns instead of using secondary
indexes',
          'Use prepared statements for repetitive queries',
        ],
      },
```

```
        {
          name: 'Lightweight transactions usage',
          description: 'Optimize usage of conditional updates (LWTs)',
          priority: 'Medium',
          impact: 'High',
          checkMethod: 'Review code for LWT usage',
          checkCode: `
      // Cassandra: Example of LWT usage
      UPDATE your_table
      SET field = 'new_value'
      WHERE partition_key = value
      IF field = 'old_value';
    `,
          bestPractices: [
            'Minimize use of LWTs due to performance impact',
            'Use LWTs only when consistency is critical',
            'Batch LWT operations when possible',
            'Monitor LWT latency separately from regular operations',
          ],
        },
        {
          name: 'Batch operation usage',
          description: 'Optimize batched mutations',
          priority: 'Medium',
          impact: 'Medium',
          checkMethod: 'Review batch usage patterns',
          bestPractices: [
            'Use unlogged batches for operations within the same partition',
            'Avoid large logged batches across multiple partitions',
            'Keep batch size under 100 statements',
            'Use client-side batching for high-volume ingestion',
          ],
        },
      ],
      systemConfiguration: [
        {
          name: 'Compaction strategy',
          description: 'Choose appropriate compaction strategy for workload',
          priority: 'High',
          impact: 'High',
          checkMethod: 'Review table configurations',
          checkCode: `
      // Cassandra: Check compaction strategy
      SELECT keyspace_name, table_name, compaction_strategy_class
      FROM system_schema.tables
      WHERE keyspace_name = 'your_keyspace';
    `,
          bestPractices: [
            'Use SizeTieredCompactionStrategy for write-heavy workloads',
            'Use LeveledCompactionStrategy for read-heavy workloads',
            'Use TimeWindowCompactionStrategy for time-series data',
            'Monitor compaction performance and adjust settings if needed',
          ],
        },
        {
```

```
        name: 'Memory configuration',
        description: 'Optimize memory settings for workload',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Review configuration files and JVM settings',
        checkCode: `
  // Cassandra: Check memory settings
  nodetool info
`,
        bestPractices: [
          'Allocate 1/4 to 1/2 of system RAM to Cassandra heap',
          'Set heap size between 8GB and 16GB maximum',
          'Leave sufficient memory for OS page cache',
          'Configure memtable settings based on write patterns',
        ],
      },
      {
        name: 'Timeout and cache settings',
        description: 'Configure timeouts and caches appropriately',
        priority: 'Medium',
        impact: 'Medium',
        checkMethod: 'Review timeouts in configuration',
        checkCode: `
  // Cassandra: View current timeout settings
  nodetool gettimeout
`,
        bestPractices: [
          'Adjust read/write timeout values based on network and workload',
          'Configure key cache size based on number of unique partition keys',
          'Monitor cache hit rates with nodetool info',
          'Adjust row cache for frequently accessed static data',
        ],
      },
    ],
  },
  graphDatabase: {
    dataModeling: [
      {
        name: 'Node and relationship design',
        description: 'Implement efficient graph model',
        priority: 'Very High',
        impact: 'Very High',
        checkMethod: 'Review graph model and query patterns',
        checkCode: `
  // Neo4j: Count nodes by label
  MATCH (n) RETURN labels(n) AS label, count(*) AS count ORDER BY count DESC;

  // Count relationships by type
  MATCH ()-[r]->() RETURN type(r) AS type, count(*) AS count ORDER BY count DESC;
`,
        bestPractices: [
          'Use descriptive labels for nodes',
          'Use relationship types that express semantic meaning',
          'Add properties to nodes and relationships as needed',
          'Keep node properties minimal, focusing on intrinsic properties',
```

```
        ],
      },
      {
        name: 'Property usage optimization',
        description: 'Optimize property storage and access',
        priority: 'Medium',
        impact: 'Medium',
        checkMethod: 'Analyze property usage patterns',
        checkCode: `
    // Neo4j: Analyze property cardinality
    MATCH (n)
    WITH properties(n) AS props, labels(n) AS labels
    UNWIND keys(props) AS key
    RETURN labels, key, count(*) AS usage
    ORDER BY usage DESC;
  `,
        bestPractices: [
          'Minimize property count on frequently accessed nodes',
          'Use arrays for small collections, relationships for larger ones',
          'Consider storing large texts in separate nodes',
          'Use appropriate data types (integers vs strings)',
        ],
      },
      {
        name: 'Relationship direction',
        description: 'Choose appropriate relationship directions',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Review relationship directions and query patterns',
        bestPractices: [
          'Design relationships in the direction most frequently traversed',
          'Use bidirectional relationships only when frequently traversed in both
directions',
          'Be consistent with relationship directions',
          'Consider query patterns when choosing direction',
        ],
      },
    ],
    queryOptimization: [
      {
        name: 'Query optimization',
        description: 'Optimize Cypher queries',
        priority: 'Very High',
        impact: 'Very High',
        checkMethod: 'Analyze query execution with PROFILE/EXPLAIN',
        checkCode: `
    // Neo4j: Explain query execution
    EXPLAIN
    MATCH (p:Person)-[:FRIENDS_WITH]->(friend)
    WHERE p.name = 'John'
    RETURN friend.name;

    // Profile query
    PROFILE
    MATCH (p:Person)-[:FRIENDS_WITH]->(friend)
```

```
      WHERE p.name = 'John'
      RETURN friend.name;
    `,
        bestPractices: [
          'Start queries with specific node labels and indexed properties',
          'Use parameters instead of literals',
          'Limit the result set size with LIMIT',
          'Add labels to patterns to reduce node scans',
          'Use OPTIONAL MATCH for optional relationships',
        ],
      },
      {
        name: 'Pattern optimization',
        description: 'Optimize query patterns for performance',
        priority: 'High',
        impact: 'High',
        checkMethod: 'Review complex query patterns',
        bestPractices: [
          'Break complex patterns into simpler ones',
          'Use variable length paths judiciously',
          'Avoid multiple variable length paths in a single query',
          'Consider direction in path patterns (→ vs ←)',
        ],
      },
      {
        name: 'Aggregate query optimization',
        description: 'Optimize aggregation and collection operations',
        priority: 'Medium',
        impact: 'Medium',
        checkMethod: 'Profile queries with aggregation',
        bestPractices: [
          'Filter before aggregation to reduce processed nodes',
          'Use UNWIND for working with collections efficiently',
          'Consider memory constraints with large aggregations',
          'Use subqueries for complex aggregations',
        ],
      },
    ],
    indexing: [
      {
        name: 'Index strategy',
        description: 'Implement appropriate indexes',
        priority: 'Very High',
        impact: 'Very High',
        checkMethod: 'Review index usage',
        checkCode: `
    // Neo4j: List all indexes
    CALL db.indexes();

    // Check index selectivity
    MATCH (n:Label)
    RETURN DISTINCT n.propertyName, count(*) as count
    ORDER BY count DESC;
  `,
        bestPractices: [
```

```
            'Create indexes on properties used in WHERE clauses',
            'Use composite indexes for multi-property queries',
            'Create indexes with high selectivity',
            'Consider unique constraints for unique properties',
          ],
        },
        {
          name: 'Full-text and spatial indexes',
          description: 'Implement specialized indexes when needed',
          priority: 'Medium',
          impact: 'High',
          checkMethod: 'Review special index needs',
          checkCode: `
    // Neo4j: Create full-text index
    CALL db.index.fulltext.createNodeIndex("nameIndex", ["Person"], ["name"]);

    // Use full-text index
    CALL db.index.fulltext.queryNodes("nameIndex", "John") YIELD node
    RETURN node;
  `,
          bestPractices: [
            'Use full-text indexes for text search needs',
            'Implement spatial indexes for geo-queries',
            'Keep full-text indexes focused on relevant fields',
            'Consider performance impact of specialized indexes',
          ],
        },
      ],
      scaleAndPerformance: [
        {
          name: 'Cache utilization',
          description: 'Optimize Neo4j cache usage',
          priority: 'High',
          impact: 'High',
          checkMethod: 'Monitor cache performance',
          checkCode: `
    // Neo4j: Check cache stats
    CALL db.stats.retrieve("cache.statistics")
  `,
          bestPractices: [
            'Adjust page cache size based on data size and available memory',
            'Monitor cache hit ratio (aim for >95%)',
            'Keep working set in memory if possible',
            'Use heap wisely for complex queries',
          ],
        },
        {
          name: 'Transaction management',
          description: 'Optimize transaction usage',
          priority: 'High',
          impact: 'Medium',
          checkMethod: 'Review transaction patterns',
          bestPractices: [
            'Keep transactions focused and short',
            'Use explicit transactions for related operations',
```

```
            'Batch creations and updates where appropriate',
            'Be mindful of lock contention on central nodes',
          ],
        },
        {
          name: 'Connection management',
          description: 'Optimize client connections',
          priority: 'Medium',
          impact: 'Medium',
          checkMethod: 'Review connection patterns in application',
          bestPractices: [
            'Use connection pooling',
            'Set appropriate timeout values',
            'Close sessions explicitly when done',
            'Monitor active connections and sessions',
          ],
        },
      ],
    },
  };

  // Function to generate optimization report for NoSQL database
  function generateNoSQLOptimizationReport(databaseType, options = {}) {
    const { includeCategories = [], priorityThreshold = 'Medium' } = options;

    // Check if database type is supported
    if (!nosqlOptimizationChecklist[databaseType]) {
      return {
        error: `Unsupported database type: ${databaseType}`,
        supportedTypes: Object.keys(nosqlOptimizationChecklist),
      };
    }

    const priorityLevels = {
      Low: 1,
      Medium: 2,
      High: 3,
      'Very High': 4,
    };

    const minimumPriority = priorityLevels[priorityThreshold];

    // Start building the report
    let report = {
      databaseType: databaseType,
      generatedAt: new Date(),
      summary: {
        totalChecks: 0,
        highPriorityItems: 0,
        mediumPriorityItems: 0,
        lowPriorityItems: 0,
      },
      categories: {},
    };
```

```
    // Get categories for the database type
    const dbChecklist = nosqlOptimizationChecklist[databaseType];

    // If no specific categories provided, include all
    const categoriesToInclude =
      includeCategories.length > 0 ? includeCategories : Object.keys(dbChecklist);

    // Process each category
    for (const category of categoriesToInclude) {
      if (!dbChecklist[category]) continue;

      report.categories[category] = {
        items: [],
      };

      // Process each item in the category
      for (const item of dbChecklist[category]) {
        // Skip items below priority threshold
        if (priorityLevels[item.priority] < minimumPriority) continue;

        report.summary.totalChecks++;

        // Track priority counts
        if (item.priority === 'High' || item.priority === 'Very High') {
          report.summary.highPriorityItems++;
        } else if (item.priority === 'Medium') {
          report.summary.mediumPriorityItems++;
        } else {
          report.summary.lowPriorityItems++;
        }

        // Add item to report
        report.categories[category].items.push({
          name: item.name,
          description: item.description,
          priority: item.priority,
          impact: item.impact,
          checkMethod: item.checkMethod,
          checkCode: item.checkCode,
          bestPractices: item.bestPractices,
        });
      }
    }

    // Generate recommendations based on priorities
    report.recommendations = generateRecommendations(report);

    return report;

    // Helper function to generate recommendations
    function generateRecommendations(report) {
      const recommendations = {
        highPriority: [],
        mediumPriority: [],
        lowPriority: [],
```

```
    };

    // Process each category
    for (const categoryName in report.categories) {
      const category = report.categories[categoryName];

      // Process each item
      for (const item of category.items) {
        // Create recommendation
        const recommendation = {
          category: categoryName,
          item: item.name,
          description: item.description,
          actions: item.bestPractices,
        };

        // Add to appropriate priority
        if (item.priority === 'High' || item.priority === 'Very High') {
          recommendations.highPriority.push(recommendation);
        } else if (item.priority === 'Medium') {
          recommendations.mediumPriority.push(recommendation);
        } else {
          recommendations.lowPriority.push(recommendation);
        }
      }
    }

    return recommendations;
  }
}
```

6.6 Interview Preparation Materials

This section provides common database interview questions and their answers.

General Database Concepts

```
const generalDatabaseInterviewQuestions = [
  {
    question: 'Explain the difference between SQL and NoSQL databases.',
    answer: `SQL (Relational) databases:
- Structured with predefined schema
- Tables with rows and columns, using normalization
- ACID compliance for reliability
- Vertical scaling (more resources on server)
- Examples: MySQL, PostgreSQL, Oracle, SQL Server

NoSQL databases:
- Schema-less or flexible schema
- Various data models (document, key-value, column-family, graph)
- Eventual consistency (though many now support ACID transactions)
- Horizontal scaling (distributing across machines)
- Examples: MongoDB, Redis, Cassandra, Neo4j
```

```
Key differences:
1. Data Structure: SQL is structured and relational, NoSQL offers various models
2. Schema: SQL requires predefined schema, NoSQL is flexible
3. Scaling: SQL typically scales vertically, NoSQL scales horizontally
4. Query Language: SQL uses standardized SQL, NoSQL uses various query methods
5. Consistency: SQL offers strong ACID guarantees, NoSQL often prioritizes
availability and partition tolerance
6. Use Cases: SQL is ideal for complex transactions and joins, NoSQL for large-scale
and schema-less applications`,
    followUp: [
      'When would you choose SQL over NoSQL?',
      'What are the CAP theorem tradeoffs for different database types?',
    ],
  },
  {
    question: 'Explain ACID properties in databases.',
    answer: `ACID stands for Atomicity, Consistency, Isolation, and Durability.
These properties ensure reliable transaction processing in database systems:

1. Atomicity: Transactions are all-or-nothing. If any part fails, the entire
transaction fails and the database state is left unchanged. It guarantees that
transactions are completed as a single, indivisible unit.

2. Consistency: Transactions bring the database from one valid state to another. All
data written to the database must comply with defined rules, constraints, cascades,
and triggers.

3. Isolation: Concurrent transactions execute as if they were running sequentially.
One transaction cannot see the intermediate or uncommitted changes made by another
transaction, preventing dirty reads, non-repeatable reads, and phantom reads.

4. Durability: Once a transaction is committed, it remains committed even in the
event of system failure. Committed data is saved by the system in non-volatile
memory.

ACID properties are crucial for applications requiring high reliability and data
integrity, such as financial systems, inventory management, and booking systems.
Traditional relational databases like PostgreSQL, MySQL, and Oracle fully support
ACID transactions. Many modern NoSQL databases like MongoDB (since v4.0) now also
support ACID transactions, though sometimes with performance trade-offs.`,
    followUp: [
      'What is the difference between a dirty read, non-repeatable read, and phantom
read?',
      'How do different transaction isolation levels affect ACID properties?',
    ],
  },
  {
    question:
      'What is database normalization and what are its advantages and
disadvantages?',
    answer: `Database normalization is the process of structuring a relational
database to minimize data redundancy and dependency by organizing fields and tables.
It involves dividing large tables into smaller tables and defining relationships
between them.
```

```
The primary normalization forms are:
1. First Normal Form (1NF): Eliminate repeating groups and identify primary key
2. Second Normal Form (2NF): Meet 1NF requirements and remove partial dependencies
3. Third Normal Form (3NF): Meet 2NF requirements and remove transitive dependencies
4. Boyce-Codd Normal Form (BCNF): Stricter version of 3NF
5. Fourth Normal Form (4NF): Deal with multi-valued dependencies
6. Fifth Normal Form (5NF): Deal with join dependencies

Advantages:
- Reduces data redundancy and storage space
- Minimizes data anomalies (update, insertion, deletion)
- Improves data integrity and consistency
- Makes the database structure more flexible for future changes
- Better supports referential integrity constraints

Disadvantages:
- Increases the number of tables, making schema more complex
- More joins are required for queries, potentially reducing performance
- Can make some types of reporting queries more complex
- May require more complex application code for data manipulation
- Can impact read performance for complex queries on large datasets

In practice, most applications use a balance of normalization (usually to 3NF) with
strategic denormalization for performance-critical operations. Modern database
systems with improved join performance and query optimization have reduced the
performance penalty of normalized designs.`,
    followUp: [
      'When would you deliberately denormalize a database?',
      "Explain referential integrity and how it's enforced in relational
databases.",
    ],
  },
  {
    question:
      'Explain the concept of database indexes. How do they work, and what are the
trade-offs?',
    answer: `Database indexes are data structures that improve the speed of data
retrieval operations by providing quick access paths to data in database tables.
They work similarly to book indexes, where you can quickly find information without
reading the entire book.

How indexes work:
1. Structure: Most commonly implemented as B-trees or B+ trees (balanced search
trees)
2. Storage: Indexes store a copy of selected columns along with pointers to the
actual rows
3. Organization: Data is organized in a sorted structure for efficient searches
4. Operation: When a query with a WHERE clause is executed, the database can use the
index to quickly locate matching rows rather than scanning the entire table

Types of indexes:
- Single-column indexes: Created on one column
- Composite indexes: Created on multiple columns
- Unique indexes: Enforce uniqueness of values
```

```
- Clustered indexes: Determine physical ordering of data
- Non-clustered indexes: Maintain logical ordering separate from physical storage
- Full-text indexes: Specialized for text search
- Spatial indexes: Specialized for geographic data

Trade-offs:

Advantages:
- Significantly speeds up query performance for SELECT statements with WHERE clauses
- Improves performance of ORDER BY and GROUP BY operations
- Enforces uniqueness through unique indexes
- Helps in finding MIN or MAX values efficiently

Disadvantages:
- Additional storage space required for each index
- Performance overhead on INSERT, UPDATE, and DELETE operations (indexes must be
updated)
- Indexes must be maintained and potentially rebuilt
- Unused indexes waste resources
- Too many indexes can lead to the optimizer making poor choices

Best practices:
- Index columns frequently used in WHERE clauses, JOIN conditions, and for sorting
- Consider the cardinality (uniqueness) of columns when creating indexes
- Regularly analyze index usage and remove unused indexes
- Consider composite indexes for queries that filter on multiple columns
- Be mindful of index size and maintenance overhead`,
    followUp: [
      'How would you decide which columns to index in a database?',
      'Explain the difference between clustered and non-clustered indexes.',
    ],
  },
  {
    question: 'What is database sharding, and when would you use it?',
    answer: `Database sharding is a horizontal partitioning technique that splits a
large database into smaller, more manageable pieces called shards. Each shard
contains a subset of the data and operates as an independent database, often
distributed across multiple servers.

How sharding works:
1. Data is partitioned based on a shard key (e.g., customer_id, geographic region)
2. Each shard contains a complete subset of data (rows) based on the shard key
3. Shards can be hosted on separate servers, potentially in different locations
4. A routing layer directs queries to the appropriate shard(s)

Sharding strategies:
- Range-based: Partitioning based on ranges of a key value (e.g., customers A-M on
Shard 1, N-Z on Shard 2)
- Hash-based: Using a hash function on the shard key for even distribution
- Directory-based: Using a lookup service to track which data is on which shard
- Geographic: Partitioning based on geographic regions

When to use sharding:
- When database size exceeds the capacity of a single server
- When write throughput is too high for a single server to handle
```

```
- To improve query response time by reducing the data to scan
- When high availability and fault tolerance are required
- To distribute data geographically closer to users

Challenges and considerations:
- Increased complexity in application design and maintenance
- Joins across shards are difficult and often need to be handled at the application
level
- Transactions spanning multiple shards are complex
- Rebalancing data when adding new shards can be challenging
- Need for a robust shard key selection to prevent hotspots
- Additional infrastructure for coordination and routing

Sharding is typically implemented after other scaling strategies (like read
replicas, query optimization, vertical scaling) have been exhausted. It's commonly
used in very large systems like social networks, large e-commerce platforms, and
global SaaS applications.`,
    followUp: [
      'How would you choose an appropriate shard key?',
      'What are the challenges of implementing transactions across multiple
shards?',
    ],
  },
];
```

SQL Database Questions

```
const sqlDatabaseInterviewQuestions = [
  {
    question: "Explain the difference between INNER JOIN, LEFT JOIN, RIGHT JOIN, and
FULL JOIN in SQL.",
    answer: `SQL JOINs are used to combine rows from two or more tables based on a
related column. The main types of JOINs are:

1. INNER JOIN: Returns only the matching rows from both tables where the join
condition is met.
   ```sql
   SELECT * FROM table1 INNER JOIN table2 ON table1.column = table2.column;
```

Only returns rows where there's a match in both tables.

2. LEFT JOIN (or LEFT OUTER JOIN): Returns all rows from the left table and matching rows from the right table. If there's no match in the right table, NULL values are returned for those columns.

   ```sql
   SELECT * FROM table1 LEFT JOIN table2 ON table1.column = table2.column;
   ```

   All rows from table1 will appear in the result, even if they don't have matches in table2.
   ```

3. RIGHT JOIN (or RIGHT OUTER JOIN): Returns all rows from the right table and matching rows from the left table. If there's no match in the left table, NULL values are returned for those columns.

```
SELECT * FROM table1 RIGHT JOIN table2 ON table1.column = table2.column;
```

   All rows from table2 will appear in the result, even if they don't have matches in table1.

4. FULL JOIN (or FULL OUTER JOIN): Returns all rows when there's a match in either the left or right table. If there's no match, NULL values are returned for columns from the table with no match.

```
SELECT * FROM table1 FULL JOIN table2 ON table1.column = table2.column;
```

   Combines the effect of LEFT and RIGHT joins, including all rows from both tables.

Additional JOIN types:

- CROSS JOIN: Returns the Cartesian product of both tables (all possible combinations of rows)
- SELF JOIN: Joining a table to itself, useful for hierarchical or self-referential data

When choosing which JOIN to use, consider what data you need in your result set:

- Use INNER JOIN when you only want rows that exist in both tables
- Use LEFT JOIN when you need all rows from the left table regardless of matches
- Use RIGHT JOIN when you need all rows from both tables regardless of matches
- Use FULL JOIN when you need all rows from both tables regardless of matches, followUp: ["When would you use a CROSS JOIN?", "What is the difference between JOIN and UNION in SQL?"] }, { question: "What are the different types of SQL indexes, and when would you use each type?", answer: SQL databases support several types of indexes, each suited for different use cases:

1. B-tree Indexes (Standard Indexes):
   - Most common index type in databases
   - Efficient for equality (=) and range queries (<, >, BETWEEN)
   - Good for high-cardinality columns (many unique values)
   - Example: `CREATE INDEX idx_customer_email ON customers(email);`
   - Use for: Primary keys, unique constraints, columns in WHERE clauses

2. Unique Indexes:
   - Enforces uniqueness of values in the indexed column(s)
   - Example: `CREATE UNIQUE INDEX idx_unique_email ON users(email);`
   - Use for: Enforcing business rules requiring uniqueness

3. Composite/Multi-column Indexes:
   - Index on multiple columns together
   - Order of columns matters significantly
   - Example: `CREATE INDEX idx_lastname_firstname ON customers(last_name, first_name);`

- Use for: Queries that filter on multiple columns together

4. Covering Indexes:
   - An index that includes all the columns needed by a query
   - Allows the database to retrieve data directly from the index without accessing the table
   - Example: `CREATE INDEX idx_covering ON orders(order_date, customer_id, status);`
   - Use for: Frequently run queries to avoid table lookups

5. Clustered Indexes:
   - Determines the physical order of data in a table
   - Only one clustered index per table
   - Primary keys are often clustered indexes by default
   - Example: `CREATE CLUSTERED INDEX idx_clustered ON transactions(transaction_date);`
   - Use for: Columns that are frequently accessed in ranges or sorting operations

6. Non-Clustered Indexes:
   - Separate structure from the data rows
   - Multiple non-clustered indexes allowed per table
   - Example: Most standard indexes are non-clustered
   - Use for: Secondary access paths to data

7. Full-Text Indexes:
   - Specialized for text search operations
   - Example: `CREATE FULLTEXT INDEX idx_fulltext ON articles(content);`
   - Use for: Searching within text fields, natural language queries

8. Spatial Indexes:
   - Optimized for geographic data
   - Example: `CREATE SPATIAL INDEX idx_location ON stores(location);`
   - Use for: Geospatial queries (e.g., finding points within a radius)

9. Hash Indexes:
   - Use hash functions for lookups
   - Very fast for equality comparisons, useless for ranges
   - Example: In-memory tables or specific database engines
   - Use for: Exact match lookups only

10. Partial/Filtered Indexes:
    - Index only a subset of rows meeting a condition
    - Example: `CREATE INDEX idx_active_users ON users(username) WHERE active = TRUE;`
    - Use for: Tables where queries target a specific subset of rows

11. Expression/Functional Indexes:
    - Index based on an expression rather than a column directly
    - Example: `CREATE INDEX idx_lower_email ON customers(LOWER(email));`
    - Use for: Queries that use functions on columns

Best practices for using indexes:

- Index columns used in JOIN, WHERE, and ORDER BY clauses
- Avoid over-indexing as it slows down writes
- Consider the workload (read-heavy vs. write-heavy)
- Monitor index usage and remove unused indexes
- Regularly rebuild/reorganize indexes to prevent fragmentation
- Be mindful of index selectivity (uniqueness of values)`, followUp: [ "How would you identify unused indexes in a production database?", "What is index selectivity and why is it important?" ] }, { question: "What is a database transaction, and what are the different transaction isolation levels in SQL?", answer:` A database transaction is a logical unit of work that contains one or more SQL operations (SELECT, INSERT, UPDATE, DELETE). Transactions have ACID properties (Atomicity, Consistency, Isolation, Durability) ensuring that database operations are processed reliably.

The standard SQL transaction isolation levels, from least strict to most strict, are:

1. READ UNCOMMITTED:
    - Lowest isolation level
    - Allows dirty reads (reading uncommitted changes), non-repeatable reads, and phantom reads
    - Provides highest concurrency but least consistency
    - Example: `SET TRANSACTION ISOLATION LEVEL READ UNCOMMITTED;`
    - Use case: When you need maximum performance and can tolerate some inconsistency
2. READ COMMITTED:
    - Default level in many databases (PostgreSQL, SQL Server, Oracle)
    - Prevents dirty reads but allows non-repeatable reads and phantom reads
    - Each query sees only committed data as of the start of that query
    - Example: `SET TRANSACTION ISOLATION LEVEL READ COMMITTED;`
    - Use case: General-purpose transactional processing
3. REPEATABLE READ:
    - Prevents dirty reads and non-repeatable reads, but allows phantom reads
    - All queries in a transaction see data as it existed at the start of the transaction
    - Example: `SET TRANSACTION ISOLATION LEVEL REPEATABLE READ;`
    - Use case: When consistency of accessed records is important
    - Default in MySQL/InnoDB
4. SERIALIZABLE:
    - Highest isolation level
    - Prevents dirty reads, non-repeatable reads, and phantom reads
    - Transactions execute as if they were run one after another (serially)
    - Example: `SET TRANSACTION ISOLATION LEVEL SERIALIZABLE;`
    - Use case: When absolute consistency is required (financial transactions)

Concurrency phenomena prevented at different levels:

1. Dirty Read: Reading uncommitted changes from another transaction

    - Example: Transaction A reads data modified by Transaction B, but B hasn't committed yet. If B rolls back, A has read invalid data.
    - Prevented by: READ COMMITTED and higher

2. Non-repeatable Read: Getting different results when reading the same data twice in the same transaction
    - Example: Transaction A reads a row, Transaction B modifies and commits that row, then A reads it again and gets a different result.
    - Prevented by: REPEATABLE READ and higher

3. Phantom Read: A query executed twice returns different sets of rows
    - Example: Transaction A queries rows matching a condition, Transaction B inserts new rows matching that condition, then A runs the query again and sees additional rows.
    - Prevented by: SERIALIZABLE

Trade-offs to consider:

- Higher isolation levels provide more consistency but reduce concurrency
- Lower isolation levels allow more concurrent operations but with more anomalies
- The appropriate level depends on application requirements for consistency vs. performance

Database-specific variations:

- Oracle uses slightly different terminology and implementation
- PostgreSQL's REPEATABLE READ prevents phantom reads in some cases
- SQL Server offers SNAPSHOT isolation as an alternative

Choosing an isolation level requires understanding the specific application requirements, concurrency needs, and consistency guarantees needed.`, followUp: [ "How do optimistic and pessimistic concurrency control relate to isolation levels?", "What is MVCC (Multi-Version Concurrency Control) and how does it work?" ] }, { question: "Explain the differences between a clustered and non-clustered index.", answer:` Clustered and non-clustered indexes are two fundamental types of database indexes with significant differences in how they store and organize data:

Clustered Index:

1. Data Organization:
    - Determines the physical order of data rows in a table
    - The table data itself is sorted and stored based on the clustered index key
    - Each table can have only ONE clustered index

2. Structure:
    - Leaf nodes contain the actual data rows of the table
    - The data pages are the leaf level of the index B-tree

3. Performance Characteristics:
    - Typically faster for range queries on the clustered key
    - Faster for queries that return many rows
    - Very efficient for ORDER BY operations on the clustered key
    - Usually slower for inserts/updates (may require data page reorganization)

4. Common Usage:
    - Primary keys are often implemented as clustered indexes by default
    - Columns frequently used in range scans or sorting operations
    - Commonly created on sequential columns (e.g., date, ID)
5. Example in SQL:

```
-- SQL Server
CREATE CLUSTERED INDEX IX_Customers_ID ON Customers(CustomerID);
```

Non-Clustered Index:

1. Data Organization:
    - Creates a separate structure from the data rows
    - Contains only the indexed columns and a pointer to the actual data rows
    - A table can have MULTIPLE non-clustered indexes (typically up to 999 in SQL Server)
2. Structure:
    - Leaf nodes contain index key values and row locators (pointers)
    - Row locators point to:
        - The clustered index key (if a clustered index exists)
        - The actual data row (if no clustered index exists)
3. Performance Characteristics:
    - Typically faster for selective queries (returning few rows)
    - Often faster for single-record lookups
    - Lower impact on insert/update operations
    - Additional storage overhead
4. Common Usage:
    - Secondary access paths (in addition to the primary/clustered access path)
    - Columns used in WHERE clauses and join conditions
    - Enforcing uniqueness when the column isn't the primary key
5. Example in SQL:

```
-- Standard in most SQL databases
CREATE INDEX IX_Customers_Email ON Customers(Email);
```

Key Differences Summarized:

1. Storage:
    - Clustered: Defines the physical order of the table data itself
    - Non-clustered: Creates a separate structure with pointers to the data

2. Quantity:
    - Clustered: Limited to one per table
    - Non-clustered: Multiple allowed per table (database-dependent limit)
3. Retrieval Method:
    - Clustered: Direct access to data rows
    - Non-clustered: Requires additional lookup to access full row data
4. Ideal Use Cases:
    - Clustered: Primary access path, range queries, sorting
    - Non-clustered: Selective queries, secondary access paths
5. Performance Impact:
    - Clustered: Higher impact on inserts/updates, faster range retrievals
    - Non-clustered: Lower impact on inserts/updates, may require key lookups

Both index types have their place in database design, and understanding their differences allows for optimizing database performance for specific workloads., followUp: ["In what scenarios would you choose NOT to create a clustered index?", "How does the choice of clustered index affect the performance of non-clustered indexes?"] }, { question: "What is a SQL injection attack and how can you prevent it?", answer: A SQL injection attack is a code injection technique where an attacker inserts malicious SQL code into input fields that are used to build dynamic SQL queries in an application. When successful, this allows attackers to:

1. Access unauthorized data
2. Modify database data
3. Delete database content
4. Execute administrative operations
5. In some cases, issue commands to the operating system

Example of vulnerable code:

```
// PHP example of vulnerable code
$username = $_POST['username'];
$query = "SELECT * FROM users WHERE username = '$username'";
$result = db_query($query);
```

If an attacker enters `' OR '1'='1` as the username, the resulting query becomes:

```
SELECT * FROM users WHERE username = '' OR '1'='1'
```

This would return all users in the database since '1'='1' is always true.

Prevention Techniques:

1. Prepared Statements with Parameterized Queries:

- The most effective defense
- SQL structure and data are separated

```
// Java example with PreparedStatement
String query = "SELECT * FROM users WHERE username = ?";
PreparedStatement stmt = connection.prepareStatement(query);
stmt.setString(1, username);
ResultSet results = stmt.executeQuery();
```

```
// Node.js example with parameterized query
const query = 'SELECT * FROM users WHERE username = $1';
const result = await client.query(query, [username]);
```

2. Use ORMs (Object-Relational Mappers):

- Most modern ORMs use parameterized queries internally

```
// Sequelize ORM example
const user = await User.findOne({ where: { username: username } });
```

```
# SQLAlchemy ORM example
user = session.query(User).filter(User.username == username).first()
```

3. Input Validation:

- Validate input type, length, format, and range
- Whitelist allowed characters rather than blacklisting bad ones

```
// Basic validation example
if (!/^[a-zA-Z0-9_]+$/.test(username)) {
  throw new Error('Invalid username');
}
```

4. Escape Special Characters:

- A secondary defense to be used alongside prepared statements
- Database-specific escaping functions

```
// PHP example with mysqli
$username = mysqli_real_escape_string($conn, $username);
```

5. Principle of Least Privilege:

- Use database accounts with restricted permissions
- Different accounts for different application functions
- Avoid connecting to the database as an admin user

6. Use Stored Procedures:

- Encapsulate SQL logic on the database server
- Call procedures with parameters instead of building dynamic SQL

```
-- Create stored procedure
CREATE PROCEDURE get_user(IN user_param VARCHAR(50))
BEGIN
  SELECT * FROM users WHERE username = user_param;
END;

-- Call procedure
CALL get_user('john');
```

7. Error Handling:

- Implement custom error pages
- Avoid exposing database error messages to users
- Log errors for administrators

8. Web Application Firewalls (WAF):

- Additional layer of security to filter malicious requests

9. Regular Security Audits:

- Code reviews focused on SQL injection vulnerabilities
- Automated scanning tools
- Penetration testing

By implementing these measures, particularly prepared statements with parameterized queries and input validation, applications can be effectively protected against SQL injection attacks., followUp: ["What is the difference between a first-order and second-order SQL injection attack?", "How would you test an application for SQL injection vulnerabilities?"] }, { question: "Explain the difference between a view, a temporary table, and a materialized view in SQL.", answer: Views, temporary tables, and materialized views are database objects that store query results, but they differ in how and when they store data, their persistence, and their performance characteristics:

1. View:

- Definition: A virtual table defined by a stored SQL query
- Storage: No data is stored; the query is executed each time the view is referenced
- Data freshness: Always returns current data
- Creation syntax:

```
CREATE VIEW employee_details AS
SELECT e.id, e.name, d.department_name
```

```
FROM employees e
JOIN departments d ON e.department_id = d.id;
```

- Use cases:
  - Simplifying complex queries
  - Abstracting underlying table structures
  - Implementing row/column-level security
  - Presenting a consistent interface to data
- Advantages:
  - Always returns up-to-date data
  - Doesn't use additional storage (except for view definition)
  - Changes to underlying tables are automatically reflected
- Disadvantages:
  - May have performance overhead for complex queries
  - Cannot be indexed directly (depends on underlying table indexes)

2. Temporary Table:

- Definition: An actual table that exists temporarily during a session or transaction
- Storage: Physically stores data in the temporary tablespace
- Data freshness: Static once created; doesn't automatically update
- Types:
  - Session-level: Visible only to the creating session, dropped when session ends
  - Transaction-level: Dropped at the end of the transaction
- Creation syntax:

```
CREATE TEMPORARY TABLE temp_employee_report AS
SELECT e.id, e.name, SUM(s.amount) as total_sales
FROM employees e
JOIN sales s ON e.id = s.employee_id
GROUP BY e.id, e.name;
```

- Use cases:
  - Complex multi-step data processing
  - Breaking down complex queries
  - Intermediate result storage
  - Performance optimization for repeated access
- Advantages:
  - Can be indexed, improving query performance
  - Reduces repeated complex calculations
  - Suitable for session-specific data processing
- Disadvantages:
  - Consumes storage space
  - Data becomes stale if source data changes
  - Limited persistence (session or transaction)

3. Materialized View:

- Definition: A view that physically stores the query results
- Storage: Actually stores data, similar to a table
- Data freshness: Contains data as of the last refresh; not automatically updated
- Creation syntax (PostgreSQL):

```
CREATE MATERIALIZED VIEW sales_summary AS
SELECT product_id, SUM(amount) as total_sales
FROM sales
GROUP BY product_id;
```

- Refresh syntax:

```
REFRESH MATERIALIZED VIEW sales_summary;
```

- Use cases:
  - Caching results of expensive queries
  - Data warehouse reporting
  - Periodic analytical processing
  - Improving read performance for complex calculations
- Advantages:
  - Significantly faster query performance for complex calculations
  - Can be indexed independently
  - Persists between sessions
  - Good for read-heavy workloads with infrequent data changes
- Disadvantages:
  - Data can become stale (not automatically updated)
  - Requires manual or scheduled refreshes
  - Consumes storage space
  - Refresh operation can be resource-intensive

Database Support and Variations:

- Views: Supported by all major SQL databases
- Temporary Tables: Widely supported, with syntax variations
- Materialized Views: Native support in Oracle, PostgreSQL, SQL Server (as indexed views), not directly in MySQL (can use tables + triggers)

Selection Criteria:

- Use a view when you need a logical abstraction with always current data
- Use a temporary table for multi-step data processing within a session
- Use a materialized view when performance is critical and data doesn't change frequently

In practice, a good database design often involves all three types for different purposes within the same application.`, followUp: ["How would you decide when to refresh a materialized view?", "What are the performance implications of joining multiple views together?"] }];

```

```

NoSQL Database Questions

```javascript
const nosqlDatabaseInterviewQuestions = [
  {
    question: "Explain the CAP theorem and how it applies to different NoSQL databases.",
    answer: `The CAP theorem, formulated by computer scientist Eric Brewer in 2000, states that a distributed database system can only provide at most two of the following three guarantees simultaneously:

1. Consistency (C): All nodes see the same data at the same time. Every read receives the most recent write.

2. Availability (A): Every request receives a response, without guarantee that it contains the most recent data. The system remains operational even with node failures.

3. Partition Tolerance (P): The system continues to operate despite network partitions (communication breakdowns between nodes).

Since network partitions are inevitable in distributed systems, in practice, the CAP theorem forces a choice between consistency and availability during a partition. Different NoSQL databases make different trade-offs:

CP (Consistency and Partition Tolerance) Systems:
- MongoDB (in its default configuration with majority writes)
- HBase
- Redis (in cluster mode)
- Neo4j (in cluster configuration)

These systems prioritize consistency over availability during partitions. They will refuse writes rather than risk data inconsistency.

AP (Availability and Partition Tolerance) Systems:
- Cassandra
- CouchDB
- Riak
- DynamoDB (in some configurations)

These systems prioritize availability over consistency during partitions. They will accept writes on all nodes, potentially creating conflicts that need resolution later.

CA (Consistency and Availability) Systems:
- Single-node relational databases
- Note: True CA systems don't exist in distributed environments because partition tolerance is required

It's important to note that CAP represents extreme trade-offs during a partition. In normal operation, systems can provide both consistency and availability.

Modern distributed databases often implement more nuanced approaches:
```

```
- Tunable consistency levels (e.g., Cassandra allows per-query consistency settings)
- Eventual consistency with conflict resolution strategies
- Strong consistency for critical operations, eventual consistency for others

Real-world examples of CAP trade-offs:
1. MongoDB: Uses a replica set model where the primary accepts writes and replicates
to secondaries. During a partition, the side with the majority of nodes can elect a
new primary, maintaining consistency but potentially sacrificing availability on the
minority side.

2. Cassandra: Uses a masterless, peer-to-peer architecture where any node can accept
writes. During a partition, all nodes remain available for reads and writes, but
consistency is temporarily sacrificed until the partition is resolved and data is
reconciled.

3. DynamoDB: Allows developers to choose between eventually consistent reads
(favoring availability) or strongly consistent reads (favoring consistency) on a
per-request basis.

The CAP theorem helps architects make appropriate database choices based on
application requirements, considering which guarantee (consistency or availability)
is more important during failure scenarios.`,
    followUp: [
      "How has PACELC extended the CAP theorem?",
      "What consistency models exist between strong consistency and eventual
consistency?"
    ]
  },
  {
    question: "Describe the data modeling approaches in MongoDB compared to
traditional relational databases.",
    answer: `MongoDB data modeling differs significantly from traditional relational
database modeling due to its document-oriented, schemaless nature. Understanding
these differences is crucial for effective database design.

Relational Database Modeling:
1. Normalization: Data is split across multiple tables to minimize redundancy
2. Relationships: Managed through foreign keys and JOINs
3. Schema: Rigid, predefined structure with fixed tables and columns
4. Design approach: Entity-Relationship modeling, focused on data integrity

MongoDB Data Modeling:
1. Denormalization: Data often embedded in a single document
2. Relationships: Handled through embedding or references
3. Schema: Flexible, allowing documents in the same collection to have different
fields
4. Design approach: Focused on application query patterns and performance

Key Differences in Modeling Approaches:

1. Document Structure vs. Tables:
   - Relational: Data spread across multiple tables, related by keys
   - MongoDB: Related data often stored in a single document with embedded sub-
documents
```

````
2. Relationship Handling:
   - Relational: Relationships implemented via foreign keys and JOIN operations
   - MongoDB: Two approaches:
     a) Embedding: Nesting related data within a document
     ```javascript
     {
       _id: 1,
       name: "John Doe",
       email: "john@example.com",
       addresses: [
         { type: "home", street: "123 Main St", city: "New York" },
         { type: "work", street: "456 Market St", city: "San Francisco" }
       ]
     }
     ```
     b) Referencing: Storing references (similar to foreign keys)
     ```javascript
     // User document
     {
       _id: 1,
       name: "John Doe",
       email: "john@example.com"
     }

     // Order document
     {
       _id: 101,
       user_id: 1,
       items: ["item1", "item2"],
       total: 59.99
     }
     ```

3. Schema Flexibility:
   - Relational: Fixed schema with ALTER TABLE needed for changes
   - MongoDB: Schema flexibility allowing fields to be added dynamically
     - Can enforce schema validation if needed
     - Allows for schema evolution without downtime

4. Design Decision Guidelines for MongoDB:

   a) When to Embed (One-to-One or One-to-Few):
   - When data is always queried together
   - When the embedded data is specific to the parent
   - When the embedded data doesn't grow unbounded
   - Example: User profile with address information

   b) When to Reference (One-to-Many or Many-to-Many):
   - When data can be queried independently
   - When the related data is updated frequently
   - When data is shared across multiple parents
   - When data can grow significantly
   - Example: Users and their orders

5. MongoDB-Specific Modeling Patterns:
````

````
   a) Subset Pattern:
   - Store a subset of frequently accessed fields together
   ```javascript
   // Product document with subset of data for listings
   {
     _id: 1,
     name: "Laptop",
     price: 1299.99,
     summary: "Latest model laptop",
     product_details: { /* full technical specifications */ },
     reviews: [/* potentially hundreds of reviews */]
   }

   // Product listing collection (subset)
   {
     _id: 1,
     name: "Laptop",
     price: 1299.99,
     summary: "Latest model laptop"
   }
````

b) Extended Reference Pattern:

- Duplicate some data to avoid frequent joins

```
// Order with extended reference to user
{
  _id: 101,
  user_id: 1,
  user_info: { name: "John Doe", email: "john@example.com" },
  items: ["item1", "item2"],
  total: 59.99
}
```

c) Computed Pattern:

- Store calculated values to avoid runtime computation

```
// Product with pre-calculated average rating
{
  _id: 1,
  name: "Laptop",
  price: 1299.99,
  reviews_count: 24,
  average_rating: 4.7
}
```

6. Practical Considerations:
```

a) Document Size: MongoDB has a 16MB document size limit

b) Write Performance:

- Embedded models excel for read-heavy workloads
- Referenced models can be better for write-heavy scenarios

c) Atomic Operations:

- MongoDB guarantees atomicity at the document level
- Multi-document transactions are supported but have performance implications

Best Practice: Design your MongoDB schema based on how your application accesses data, not solely on the conceptual data relationships. The ideal schema minimizes read operations while keeping write operations efficient., followUp: ["What are the performance implications of embedding vs. referencing in MongoDB?", "How would you implement a many-to-many relationship in MongoDB?"] }, { question: "What is sharding in MongoDB and how does it work?", answer: Sharding in MongoDB is a horizontal scaling method that distributes data across multiple machines (called shards) to support deployments with very large data sets and high throughput operations. This approach allows MongoDB to scale beyond the constraints of a single server.

Key Components of MongoDB Sharding:

1. Shards:
   - Each shard is a separate MongoDB instance (or replica set) that holds a subset of the data
   - Each shard operates independently for most operations
   - Together, shards form the complete dataset
2. Config Servers:
   - Store metadata and configuration settings for the cluster
   - Track which data is on which shard
   - Implemented as a replica set for reliability
3. Mongos Routers:
   - Query routers that direct operations to the appropriate shard(s)
   - Act as an interface between client applications and the sharded cluster
   - Can have multiple mongos instances for load balancing

How Sharding Works in MongoDB:

1. Shard Key Selection:
   - Sharded collections require a shard key (an indexed field or compound fields)
   - The shard key determines how data is distributed across shards
   - Example: `sh.shardCollection("mydatabase.users", { "user_id": 1 })`
2. Data Distribution Strategies:
   - Range-Based Sharding: Divides data into ranges based on the shard key value
     - Example: Users with IDs 1-1000 on Shard A, 1001-2000 on Shard B

        - Good for range queries but vulnerable to hotspots
    - Hash-Based Sharding: Uses a hash of the shard key to distribute data evenly
        - Example: `sh.shardCollection("mydatabase.users", { "user_id": "hashed" })`
        - Better distribution but not ideal for range queries
    - Zone Sharding: Associates specific ranges of shard key values with specific shards
        - Example: Users from Europe on Shard A, Users from North America on Shard B
        - Useful for data locality and regulatory requirements

3. Chunks:
    - Data is divided into chunks (default size is 64MB)
    - Each chunk contains a subset of sharded data
    - MongoDB automatically migrates chunks between shards to achieve balanced distribution
    - Example: A chunk might contain users with IDs 1000-2000

4. Query Execution:
    - Targeted queries (including shard key) go directly to the specific shard
        - Example: Query for user_id = 1500 goes directly to the shard containing that ID
    - Scatter-gather queries (not using shard key) must query all shards
        - Example: Query for all users with premium = true checks all shards

5. Balancing Process:
    - A background process redistributes chunks to keep data evenly distributed
    - Triggered automatically when distribution becomes imbalanced
    - Can be scheduled during off-peak hours

Important Considerations for Sharding:

1. Shard Key Selection Criteria:
    - High cardinality: Many possible values
    - Even write distribution: Prevent hotspots
    - Frequency of targeted queries: Efficiency for common queries
    - Consider future growth patterns

2. Limitations and Challenges:
    - Shard key is immutable after sharding
    - No efficient way to perform non-shard key sorts across the entire dataset
    - Jumbo chunks may develop and be difficult to move
    - Operations spanning multiple shards have higher latency

3. Operations in a Sharded Environment:
    - Multi-document transactions have special considerations
    - Some admin commands must be run on specific components

- Index creation should be coordinated across shards

4. Typical Sharding Use Cases:
   - Datasets exceeding single server storage capacity
   - Write throughput exceeding single server capabilities
   - Workloads requiring more RAM than a single server
   - Geographically distributed data for lower latency

5. Monitoring and Maintenance:
   - Regular checks for chunk distribution balance
   - Monitoring migration process performance
   - Tracking query patterns to ensure shard key effectiveness

Example Sharding Setup in MongoDB Shell:

```
// Configure a sharded cluster
sh.enableSharding('ecommerce');

// Create an index on the shard key field
db.products.createIndex({ category: 1, item_id: 1 });

// Shard the collection using the index
sh.shardCollection('ecommerce.products', { category: 1, item_id: 1 });

// Add zone-based sharding for geographical distribution
sh.addShardToZone('shard0001', 'us-east');
sh.addShardToZone('shard0002', 'us-west');
sh.addShardToZone('shard0003', 'europe');

// Configure zone ranges
sh.updateZoneKeyRange(
  'ecommerce.products',
  { category: 'electronics', item_id: MinKey },
  { category: 'electronics', item_id: MaxKey },
  'us-east'
);
```

Sharding is a powerful capability that allows MongoDB to scale horizontally for large datasets and high throughput applications, but it requires careful planning, particularly for shard key selection, to ensure optimal performance.`, followUp: [ "What makes a good shard key vs. a poor shard key in MongoDB?", "What is the impact of a poorly chosen shard key on database performance?" ] }, { question: "Explain the consistency models in Apache Cassandra and how they compare to traditional ACID transactions.", answer:` Apache Cassandra offers tunable consistency with different consistency levels as an alternative to traditional ACID transactions. This approach prioritizes availability and partition tolerance while allowing developers to choose the appropriate consistency level based on their application needs.

Traditional ACID Transactions:

1. Atomicity: Operations are all-or-nothing
2. Consistency: The database moves from one valid state to another

3. Isolation: Concurrent transactions don't interfere with each other
4. Durability: Committed changes persist even after system failures

Cassandra's eventual consistency model differs fundamentally from ACID, but offers tunable consistency levels for greater flexibility.

Cassandra Consistency Model:

1. Write Path in Cassandra:
    - Client connects to any node (coordinator)
    - Coordinator forwards write to replicas based on the consistency level
    - Writes go to commit log (durability) and memtable (in-memory)
    - Periodically flushed to SSTables on disk
    - Consistency level determines how many replicas must acknowledge the write
2. Read Path in Cassandra:
    - Coordinator requests data from replicas based on consistency level
    - Performs digest query to detect inconsistencies
    - If inconsistencies exist, initiates read repair
    - Returns most recent version based on timestamp
3. Consistency Levels in Cassandra:

    Write Consistency Levels:

    - ANY: Write to at least one node (lowest consistency, highest availability)
    - ONE: Write must be committed to at least one replica's commit log and memtable
    - QUORUM: Write must be committed to a quorum of replicas (majority: N/2+1)
    - LOCAL_QUORUM: Quorum in the local datacenter only
    - EACH_QUORUM: Quorum in each datacenter
    - ALL: Write must be committed to all replicas (highest consistency, lowest availability)

    Read Consistency Levels:

    - ONE: Read from one replica (may not be the latest data)
    - QUORUM: Read from a quorum of replicas, compare versions and return the most recent
    - LOCAL_QUORUM: Quorum in the local datacenter
    - ALL: Read from all replicas, return the most recent version
4. Consistency Guarantees:
    - Strong Consistency: Achieved when (R + W > N), where:
        - R = Read consistency level (number of nodes queried for read)
        - W = Write consistency level (number of nodes required for successful write)
        - N = Replication factor (total number of replicas)
    - Example of Strong Consistency:
        - N = 3 (3 replicas)
        - W = QUORUM (2 nodes)
        - R = QUORUM (2 nodes)

        - Here, R + W = 4, which is > N (3), ensuring strong consistency
    - Eventual Consistency: When (R + W ≤ N)
        - Example: W = ONE, R = ONE, N = 3
        - R + W = 2, which is < N (3), allowing eventual consistency

5. Lightweight Transactions (LWT):
    - Closest to ACID transactions in Cassandra
    - Implements Compare-And-Set (CAS) operations using Paxos consensus protocol
    - Syntax: `UPDATE ... IF ...` or `INSERT ... IF NOT EXISTS`
    - Significantly slower than regular operations (3-4x overhead)
    - Example:

```
UPDATE accounts
SET balance = 500
WHERE id = 123
IF balance = 1000;
```

6. Comparison with ACID Transactions:

   Atomicity:
    - ACID: All-or-nothing operations across multiple entities
    - Cassandra: Single-row atomicity guaranteed; multi-row atomicity only through LWTs with performance impact

   Consistency:
    - ACID: Enforces integrity constraints and rules
    - Cassandra: Tunable consistency based on consistency level; application must manage domain consistency

   Isolation:
    - ACID: Multiple isolation levels to control transaction visibility
    - Cassandra: Row-level isolation; no multi-statement transactions except with LWTs

   Durability:
    - ACID: Committed transactions survive system failures
    - Cassandra: Durable once acknowledgment received based on consistency level

7. Real-world Consistency Strategy Examples:

   a) Banking Application:
    - Use QUORUM/ALL for critical financial transactions
    - Use LWTs for balance updates
    - Example:

```
BEGIN BATCH
  UPDATE accounts SET balance = balance - 100 WHERE id = 123 IF balance >=
100;
  UPDATE accounts SET balance = balance + 100 WHERE id = 456;
APPLY BATCH;
```

b) Product Catalog:

- Use LOCAL_QUORUM for product updates
- Use ONE for product views
- Use counters for view counts

c) Session Management:

- Use ONE for session writes and reads
- Accept eventual consistency for non-critical user data

8. Best Practices for Consistency in Cassandra:

- Choose consistency levels based on business requirements
- Use strong consistency (QUORUM, ALL) for critical operations
- Use eventual consistency (ONE, ANY) for non-critical operations
- Design around consistency limitations (e.g., denormalized data models)
- Implement application-level conflict resolution when needed
- Use LWTs sparingly due to performance impact
- Consider multi-datacenter consistency requirements

Cassandra's approach prioritizes availability and partition tolerance while offering tunable consistency, making it suitable for use cases where high availability and scalability are more important than strict consistency, such as time-series data, product catalogs, and high-volume write applications.`, followUp: [ "How does Cassandra handle conflict resolution during read repair?", "What are the performance implications of using LWTs in Cassandra?" ] }, { question: "Explain the different types of NoSQL databases and their use cases.", answer:` NoSQL databases can be categorized into four main types, each with distinct data models and optimized for specific use cases. Understanding these differences is crucial for choosing the right database for your application requirements.

1. Document Databases:

   Data Model:

   - Stores data in flexible, semi-structured documents (typically JSON or BSON)
   - Schema-free or schema-flexible
   - Documents can have varying structure within the same collection
   - Each document contains both data and its associated metadata

   Key Examples:

   - MongoDB
   - Couchbase
   - Amazon DocumentDB
   - Firebase Firestore

Strengths:

- Flexible schema allows for agile development
- Intuitive data model that maps well to object-oriented programming
- Good for nested data
- Supports rich queries and indexing
- Easy horizontal scaling

Ideal Use Cases:

- Content management systems
- User profiles and preferences
- Product catalogs
- Real-time analytics
- IoT applications
- Mobile applications

Example Document:

```
{
  "id": "12345",
  "user": "john_doe",
  "email": "john@example.com",
  "profile": {
    "firstName": "John",
    "lastName": "Doe",
    "address": {
      "city": "New York",
      "zip": "10001"
    }
  },
  "interests": ["programming", "databases", "hiking"],
  "lastLogin": "2023-06-15T10:30:00Z"
}
```

2. Key-Value Stores:

Data Model:

- Simplest NoSQL data model
- Each item is stored as a key-value pair
- Values are opaque to the database (no query on value contents)
- Highly optimized for basic operations (get, put, delete)

Key Examples:

- Redis
- Amazon DynamoDB
- Riak
- Memcached

Strengths:

- Extremely fast operations
- Highly scalable
- Low latency
- Simple interface
- Great caching layer

Ideal Use Cases:

- Caching
- Session management
- User preferences
- Shopping carts
- Real-time leaderboards
- Message queuing
- Distributed locks

Example Operations:

```
SET user:1000:session "encrypted_session_data"
GET user:1000:session
EXPIRE user:1000:session 3600
```

3. Column-Family Stores:

Data Model:

- Stores data in tables, rows, and columns
- Sparse data (not every row has every column)
- Designed for large amounts of data across many machines
- Column families group related columns

Key Examples:

- Apache Cassandra
- HBase
- ScyllaDB
- Google Bigtable

Strengths:

- Highly scalable for massive datasets
- Optimized for write-heavy workloads
- Tunable consistency
- Excellent performance for specific query patterns
- Good for time-series and log data

Ideal Use Cases:

- Time-series data
- IoT sensor data
- Log data

- Monitoring systems
- Recommendation engines
- Fraud detection systems
- High-volume write applications

Example Data Model (Cassandra):

```
CREATE TABLE user_activity (
  user_id uuid,
  timestamp timestamp,
  activity_type text,
  details map<text, text>,
  PRIMARY KEY (user_id, timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

4. Graph Databases:

   Data Model:

   - Stores nodes, edges, and properties
   - Optimized for complex relationships between entities
   - Native support for traversing connections
   - Property graphs with labeled relationships

   Key Examples:

   - Neo4j
   - Amazon Neptune
   - JanusGraph
   - ArangoDB

   Strengths:

   - Efficient for complex relationship queries
   - Intuitive for network and relationship modeling
   - Flexible schema
   - Powerful query capabilities for connected data
   - Natural representation of hierarchical structures

   Ideal Use Cases:

   - Social networks
   - Recommendation engines
   - Fraud detection
   - Knowledge graphs
   - Network and IT operations
   - Identity and access management
   - Supply chain management

   Example Query (Cypher for Neo4j):

```
MATCH (user:Person {name: 'John'})-[:FRIEND]->(friend)-[:LIKES]->(product)
WHERE NOT (user)-[:LIKES]->(product)
RETURN product.name, count(friend) AS frequency
ORDER BY frequency DESC
LIMIT 5;
```

5. Multi-model Databases:

   Data Model:

   - Support multiple data models within a single database
   - Flexible approach combining aspects of other NoSQL types
   - Single platform for different data representation needs

   Key Examples:

   - ArangoDB
   - OrientDB
   - Couchbase
   - FaunaDB

   Strengths:

   - Versatility for diverse application requirements
   - Simplified infrastructure (fewer databases to manage)
   - Unified query language
   - Consistency across different data models

   Ideal Use Cases:

   - Complex applications requiring multiple data models
   - Microservices architectures
   - Applications with evolving data requirements

Choosing the Right NoSQL Database:

1. Decision Framework:

   - What is your primary data access pattern? (Key-based, document retrieval, relationship traversal, etc.)
   - Scale requirements (read/write volume, data size)
   - Consistency requirements
   - Query complexity
   - Development agility needs

2. Common Decision Patterns:

   - Simple high-performance key retrieval → Key-Value Store
   - Flexible schema with rich queries → Document Database
   - Massive write-heavy workloads with time-based data → Column-Family Store
   - Relationship-heavy data with complex queries → Graph Database

3. Hybrid Approaches:

- Using multiple databases for different aspects of an application
- Polyglot persistence: matching data store to specific requirements
- Example: MongoDB for user profiles, Redis for caching, Neo4j for recommendations

4. Factors Beyond Data Model:

- Operational complexity
- Developer familiarity
- Community and support
- Cloud vendor integration
- Total cost of ownership

NoSQL databases excel in specific domains where relational databases have limitations, particularly in terms of scalability, schema flexibility, and specialized query patterns. The right choice depends on understanding your data characteristics, access patterns, and application requirements.`, followUp: [ "When would you choose a multi-model database over specialized databases?", "What are the operational challenges of running different types of NoSQL databases?" ] }, { question: "How does Redis achieve its high performance, and what are its key data structures?", answer:` Redis achieves its exceptional performance through a combination of architectural decisions and optimizations. Understanding both the performance factors and Redis' versatile data structures is crucial for leveraging its full potential.

Performance Factors That Make Redis Fast

1. In-Memory Storage:

- All data is stored primarily in RAM
- Eliminates disk I/O latency (typically microseconds vs. milliseconds)
- Access to data at memory speed
- Data structures optimized for memory access patterns

2. Single-Threaded Architecture:

- Core operations execute in a single thread
- Eliminates complex locking and concurrency issues
- Predictable performance without thread coordination overhead
- Recent versions have moved some operations to background threads

3. Efficient Data Structures:

- Specialized implementations optimized for specific operations
- Highly optimized algorithms for common operations
- Memory-efficient internal representations
- Constant or near-constant time complexity for most operations

4. Event-Driven I/O:

- Non-blocking I/O using an event loop (similar to Node.js)
- Handles thousands of connections simultaneously
- Efficient multiplexing with minimal context switching
- Uses epoll/kqueue/select depending on the platform

5. Minimalist Design:

- Focused feature set with clear trade-offs
- Simple, well-defined commands
- Optimized network protocol (RESP - Redis Serialization Protocol)
- Binary-safe implementation

6. Memory Management:

   - Custom memory allocator (jemalloc)
   - Reduces fragmentation
   - Efficient reuse of memory
   - Pooled memory allocation for common sizes

7. Persistence Options:

   - Asynchronous persistence mechanisms
   - Background saving doesn't block the main thread
   - Options for different durability requirements
   - Trade-offs between performance and data safety

Key Redis Data Structures

1. Strings:

   - Most basic data type
   - Can store text, integers, or binary data up to 512MB
   - Memory-efficient storage with automatic conversion for small integers

   Operations and Use Cases:

   - GET/SET for simple key-value storage
   - INCR/DECR for atomic counters
   - SETEX for automatic expiration
   - GETSET for atomic update and retrieval

   Example:

   ```
   SET user:1000:session "encrypted_session_data"
   GET user:1000:session
   INCR pageviews
   SETEX login:token:abc123 3600 "user:1000"  # Expires in 1 hour
   ```

2. Lists:

   - Linked lists of string values
   - Order is preserved (insertion order)
   - Efficient for push/pop operations on both ends

   Operations and Use Cases:

   - LPUSH/RPUSH for adding elements
   - LPOP/RPOP for removing elements

- LRANGE for retrieving ranges of elements
- Ideal for queues, recent activity lists, and messaging

Example:

```
LPUSH notifications:user:1000 "New message from Alice"
LPUSH notifications:user:1000 "Payment received"
LRANGE notifications:user:1000 0 -1  # Get all notifications
RPOP tasks:queue  # Process next task
```

3. Sets:

   - Unordered collections of unique strings
   - Fast membership testing
   - Set operations: union, intersection, difference

   Operations and Use Cases:

   - SADD for adding members
   - SISMEMBER for membership testing
   - SINTER/SUNION/SDIFF for set operations
   - Perfect for unique item tracking, relationships, tagging

   Example:

```
SADD user:1000:interests "databases" "programming" "hiking"
SADD user:1001:interests "hiking" "photography" "travel"
SISMEMBER user:1000:interests "databases"  # Check membership
SINTER user:1000:interests user:1001:interests  # Common interests
```

4. Sorted Sets:

   - Sets with each member associated with a score
   - Members sorted by score
   - Efficient operations for ordered data

   Operations and Use Cases:

   - ZADD for adding members with scores
   - ZRANGE for range retrieval by position
   - ZRANGEBYSCORE for range retrieval by score
   - ZRANK for determining rank
   - Perfect for leaderboards, priority queues, time-based data

   Example:

```
ZADD leaderboard 1000 "user:1000"
ZADD leaderboard 2500 "user:1001"
ZADD leaderboard 1800 "user:1002"
```

```
ZREVRANGE leaderboard 0 2 WITHSCORES  # Top 3 users
ZRANK leaderboard "user:1000"  # Get rank
```

5. Hashes:
    - Maps of field-value pairs
    - Efficient storage for objects/structs
    - Field-level operations

    Operations and Use Cases:

    - HSET for setting field values
    - HGET for retrieving field values
    - HMGET for retrieving multiple fields
    - HINCRBY for incrementing numeric fields
    - Ideal for representing objects and structured data

    Example:

```
HSET user:1000 name "John Doe" email "john@example.com" age 35
HGET user:1000 email
HMGET user:1000 name email
HINCRBY user:1000 login_count 1
```

6. Bit Arrays (Bitmaps):
    - String operations treating strings as bit arrays
    - Extremely memory efficient for boolean information

    Operations and Use Cases:

    - SETBIT for setting individual bits
    - GETBIT for retrieving bits
    - BITCOUNT for counting set bits
    - Perfect for user presence, feature flags, real-time analytics

    Example:

```
SETBIT user:activity:2023-06-15 1000 1  # User 1000 was active today
GETBIT user:activity:2023-06-15 1000    # Check if user was active
BITCOUNT user:activity:2023-06-15       # Count active users
```

7. HyperLogLog:
    - Probabilistic data structure for cardinality estimation
    - Extremely memory efficient (constant space requirement)
    - Approximates count of unique elements

    Operations and Use Cases:

- PFADD for adding elements
- PFCOUNT for estimating unique count
- Ideal for unique visitor counts, metrics

Example:

```
PFADD visitors:2023-06-15 "user:1000" "user:1001" "user:1002"
PFADD visitors:2023-06-15 "user:1000" "user:1003"  # Duplicate is ignored
PFCOUNT visitors:2023-06-15  # Returns approximately 3
```

8. Streams:

   - Append-only log-like data structure (added in Redis 5.0)
   - Consumer groups for reliable message processing

   Operations and Use Cases:

   - XADD for adding entries
   - XREAD for reading streams
   - XGROUP for consumer group management
   - Ideal for event sourcing, messaging, and time-series data

   Example:

```
XADD events * type sensor-reading value 42 unit celsius
XREAD COUNT 10 STREAMS events 0  # Read 10 messages from the beginning
```

9. Geospatial Indexes:

   - Store and query geospatial data
   - Calculate distances, find points within radius

   Operations and Use Cases:

   - GEOADD for adding locations
   - GEODIST for distance calculation
   - GEORADIUS for radius search
   - Perfect for location-based applications

   Example:

```
GEOADD locations 13.361389 38.115556 "Palermo" 15.087269 37.502669 "Catania"
GEODIST locations "Palermo" "Catania" km  # Distance in kilometers
GEORADIUS locations 15 37 100 km  # Points within 100km of coordinates
```

Advanced Performance Considerations

1. Memory Optimization:

- Use appropriate data structures (e.g., HASH instead of multiple STRING keys)
- Configure maxmemory and eviction policies
- Use Redis object encoding optimizations (e.g., shared integers)
- Consider key expiration for temporary data

2. Command Optimization:

- Use pipelining for batching commands
- Leverage multi-key operations (MGET, MSET)
- Use Lua scripts for atomic multi-step operations
- Command complexity awareness (avoid O(N) commands on large data sets)

3. Connection Management:

- Connection pooling in clients
- Persistent connections
- Connection timeout settings
- Thread-safe clients for concurrent access

Redis combines these architectural choices and data structures to deliver microsecond response times, making it suitable for a wide range of use cases requiring high-performance data access.`, followUp: ["What are the trade-offs between different Redis persistence options?", "How would you design a rate limiting system using Redis?"] }];

````
#### Database Design Questions

```javascript
const databaseDesignInterviewQuestions = [
  {
    question: "How would you design a database schema for a social media
application?",
    answer: `Designing a database schema for a social media application requires
balancing performance, scalability, and feature support. The schema needs to handle
user relationships, content creation, engagement interactions, and real-time
features efficiently.

## Core Entities and Relationships

### 1. Users and Authentication

```sql
CREATE TABLE users (
    user_id BIGINT PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    password_hash VARCHAR(255) NOT NULL,
    full_name VARCHAR(100) NOT NULL,
    bio TEXT,
    profile_image_url VARCHAR(255),
    is_verified BOOLEAN DEFAULT FALSE,
    is_active BOOLEAN DEFAULT TRUE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
````

```
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    last_login_at TIMESTAMP,
    INDEX idx_username (username),
    INDEX idx_email (email)
);

CREATE TABLE user_auth_tokens (
    token_id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    token_hash VARCHAR(255) NOT NULL,
    token_type ENUM('access', 'refresh', 'password_reset') NOT NULL,
    expires_at TIMESTAMP NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    INDEX idx_user_token_type (user_id, token_type)
);

CREATE TABLE user_devices (
    device_id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    device_token VARCHAR(255) NOT NULL,
    device_type VARCHAR(50) NOT NULL,
    last_used_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    INDEX idx_user_device (user_id, device_type)
);
```

2. User Profile and Settings

```
CREATE TABLE user_profiles (
    profile_id BIGINT PRIMARY KEY,
    user_id BIGINT UNIQUE NOT NULL,
    location VARCHAR(100),
    website VARCHAR(255),
    birthday DATE,
    phone VARCHAR(20),
    gender VARCHAR(20),
    language VARCHAR(10) DEFAULT 'en',
    theme VARCHAR(20) DEFAULT 'light',
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);

CREATE TABLE user_settings (
    settings_id BIGINT PRIMARY KEY,
    user_id BIGINT UNIQUE NOT NULL,
    notification_email BOOLEAN DEFAULT TRUE,
    notification_push BOOLEAN DEFAULT TRUE,
    account_privacy ENUM('public', 'private') DEFAULT 'public',
    who_can_message ENUM('everyone', 'followers', 'none') DEFAULT 'everyone',
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE
);
```

3. Followers/Connections

```
CREATE TABLE user_connections (
    connection_id BIGINT PRIMARY KEY,
    follower_id BIGINT NOT NULL,
    following_id BIGINT NOT NULL,
    connection_status ENUM('pending', 'accepted', 'rejected', 'blocked') DEFAULT
'accepted',
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (follower_id) REFERENCES users(user_id) ON DELETE CASCADE,
    FOREIGN KEY (following_id) REFERENCES users(user_id) ON DELETE CASCADE,
    UNIQUE KEY unique_connection (follower_id, following_id),
    INDEX idx_follower (follower_id),
    INDEX idx_following (following_id)
);
```

4. Posts and Content

```
CREATE TABLE posts (
    post_id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    content TEXT,
    post_type ENUM('text', 'image', 'video', 'link', 'poll') NOT NULL,
    privacy_setting ENUM('public', 'followers', 'private') DEFAULT 'public',
    is_edited BOOLEAN DEFAULT FALSE,
    is_pinned BOOLEAN DEFAULT FALSE,
    is_archived BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    INDEX idx_user_created (user_id, created_at),
    INDEX idx_privacy_created (privacy_setting, created_at)
);

CREATE TABLE post_media (
    media_id BIGINT PRIMARY KEY,
    post_id BIGINT NOT NULL,
    media_type ENUM('image', 'video', 'audio') NOT NULL,
    media_url VARCHAR(255) NOT NULL,
    thumbnail_url VARCHAR(255),
    width INT,
    height INT,
    duration INT,  -- for videos/audio in seconds
    alt_text TEXT,
    position INT DEFAULT 0,  -- for ordering multiple media
    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
    INDEX idx_post_position (post_id, position)
);

CREATE TABLE hashtags (
    hashtag_id BIGINT PRIMARY KEY,
```

```
    name VARCHAR(100) UNIQUE NOT NULL,
    post_count INT DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    INDEX idx_name (name),
    INDEX idx_popularity (post_count DESC)
);

CREATE TABLE post_hashtags (
    post_hashtag_id BIGINT PRIMARY KEY,
    post_id BIGINT NOT NULL,
    hashtag_id BIGINT NOT NULL,
    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
    FOREIGN KEY (hashtag_id) REFERENCES hashtags(hashtag_id) ON DELETE CASCADE,
    UNIQUE KEY unique_post_hashtag (post_id, hashtag_id),
    INDEX idx_hashtag_post (hashtag_id, post_id)
);

CREATE TABLE user_mentions (
    mention_id BIGINT PRIMARY KEY,
    post_id BIGINT NOT NULL,
    mentioned_user_id BIGINT NOT NULL,
    position INT,  -- character position in the post
    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
    FOREIGN KEY (mentioned_user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    INDEX idx_user_mentions (mentioned_user_id, post_id)
);
```

5. Comments and Reactions

```
CREATE TABLE comments (
    comment_id BIGINT PRIMARY KEY,
    post_id BIGINT NOT NULL,
    user_id BIGINT NOT NULL,
    parent_comment_id BIGINT,  -- For nested comments
    content TEXT NOT NULL,
    is_edited BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP ON UPDATE CURRENT_TIMESTAMP,
    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    FOREIGN KEY (parent_comment_id) REFERENCES comments(comment_id) ON DELETE
CASCADE,
    INDEX idx_post_time (post_id, created_at),
    INDEX idx_parent_time (parent_comment_id, created_at)
);

CREATE TABLE reactions (
    reaction_id BIGINT PRIMARY KEY,
    reaction_type ENUM('like', 'love', 'haha', 'wow', 'sad', 'angry') NOT NULL,
    user_id BIGINT NOT NULL,
    content_type ENUM('post', 'comment') NOT NULL,
    content_id BIGINT NOT NULL,  -- Either post_id or comment_id
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
```

```
    UNIQUE KEY unique_user_reaction (user_id, content_type, content_id),
    INDEX idx_content (content_type, content_id, reaction_type),
    INDEX idx_user_reactions (user_id, created_at)
);
```

6. Messages

```
CREATE TABLE conversations (
    conversation_id BIGINT PRIMARY KEY,
    is_group BOOLEAN DEFAULT FALSE,
    title VARCHAR(100),  -- For group conversations
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

CREATE TABLE conversation_participants (
    participant_id BIGINT PRIMARY KEY,
    conversation_id BIGINT NOT NULL,
    user_id BIGINT NOT NULL,
    nickname VARCHAR(50),
    role ENUM('admin', 'member') DEFAULT 'member',
    joined_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    left_at TIMESTAMP,
    FOREIGN KEY (conversation_id) REFERENCES conversations(conversation_id) ON
DELETE CASCADE,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    UNIQUE KEY unique_conversation_user (conversation_id, user_id),
    INDEX idx_user_conversations (user_id, conversation_id)
);

CREATE TABLE messages (
    message_id BIGINT PRIMARY KEY,
    conversation_id BIGINT NOT NULL,
    sender_id BIGINT NOT NULL,
    message_type ENUM('text', 'image', 'video', 'audio', 'file', 'location',
'system') NOT NULL,
    content TEXT,
    media_url VARCHAR(255),
    is_deleted BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (conversation_id) REFERENCES conversations(conversation_id) ON
DELETE CASCADE,
    FOREIGN KEY (sender_id) REFERENCES users(user_id) ON DELETE SET NULL,
    INDEX idx_conversation_time (conversation_id, created_at)
);

CREATE TABLE message_status (
    status_id BIGINT PRIMARY KEY,
    message_id BIGINT NOT NULL,
    user_id BIGINT NOT NULL,
    is_delivered BOOLEAN DEFAULT FALSE,
    is_read BOOLEAN DEFAULT FALSE,
    delivered_at TIMESTAMP,
```

```
    read_at TIMESTAMP,
    FOREIGN KEY (message_id) REFERENCES messages(message_id) ON DELETE CASCADE,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    UNIQUE KEY unique_message_user (message_id, user_id),
    INDEX idx_user_unread (user_id, is_read)
);
```

7. Notifications

```
CREATE TABLE notifications (
    notification_id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    sender_id BIGINT,
    notification_type ENUM('follow', 'like', 'comment', 'mention', 'message', 'tag',
'system') NOT NULL,
    content_type VARCHAR(20),  -- 'post', 'comment', etc.
    content_id BIGINT,         -- ID of the related content
    message TEXT,
    is_read BOOLEAN DEFAULT FALSE,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    FOREIGN KEY (sender_id) REFERENCES users(user_id) ON DELETE SET NULL,
    INDEX idx_user_read_time (user_id, is_read, created_at),
    INDEX idx_user_type (user_id, notification_type)
);
```

8. Activity and Analytics

```
CREATE TABLE user_activities (
    activity_id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    activity_type VARCHAR(50) NOT NULL, -- 'login', 'post', 'comment', etc.
    ip_address VARCHAR(45),
    user_agent TEXT,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    INDEX idx_user_activity_time (user_id, activity_type, created_at)
);

CREATE TABLE post_analytics (
    analytics_id BIGINT PRIMARY KEY,
    post_id BIGINT NOT NULL,
    view_count INT DEFAULT 0,
    unique_viewer_count INT DEFAULT 0,
    share_count INT DEFAULT 0,
    last_updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
    INDEX idx_popularity (view_count DESC, share_count DESC)
);
```

Optimizations and Considerations

1. Denormalization for Performance

For feed generation, consider:

```
CREATE TABLE user_feeds (
    feed_id BIGINT PRIMARY KEY,
    user_id BIGINT NOT NULL,
    post_id BIGINT NOT NULL,
    post_author_id BIGINT NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    score FLOAT,  -- For algorithm-based feeds
    FOREIGN KEY (user_id) REFERENCES users(user_id) ON DELETE CASCADE,
    FOREIGN KEY (post_id) REFERENCES posts(post_id) ON DELETE CASCADE,
    INDEX idx_user_time (user_id, created_at)
);
```

2. Sharding Strategies

- Partition `posts` by user_id (application-level sharding)
- Partition `messages` by conversation_id
- Time-based partitioning for historical data

3. Caching Strategy

- Cache user profiles and frequently accessed data
- Cache news feeds for active users
- Implement counters for likes, comments, followers in Redis

4. Real-time