Data Model:

Tables:

- ParkingSpots:

- ID (Primary Key)

- SpotNumber

- Floor

- Size (e.g., motorcycle, car, bus)

- Availability (Boolean)

- Vehicles:

- ID (Primary Key)

- PlateNumber

- Size

- CheckInTime

- ParkingTransactions:

- ID (Primary Key)

- VehicleID (Foreign Key)

- CheckInTime

- CheckOutTime

- Duration

- Fee

2. Algorithm for Spot Allocation:

Algorithm:

1. When a vehicle enters:

- Query the database for available parking spots based on size and floor.

- Select the nearest available spot if multiple spots are available.

- Assign the spot to the vehicle.

- Update the availability of the spot in the database.

2. If no spots are available:

- Display a message indicating the parking lot is full.

3. Fee Calculation Logic:

Logic:

1. Calculate the duration of stay based on the check-in and check-out times.

2. Determine the fee according to the duration and vehicle type:

- Base fee per hour for each vehicle type.

- Additional fees for exceeding a certain duration.

3. Apply any discounts or special rates (e.g., early bird discounts, monthly passes).

4. Update the ParkingTransactions table with the calculated fee.

4. Concurrency Handling:

Techniques:

1. Use database transactions for atomicity:

- Begin a transaction before updating any related tables.

- Commit the transaction after all updates are successful.

- Rollback the transaction if any operation fails.

2. Implement locking mechanisms to prevent race conditions:

- Use row-level or table-level locks to ensure data integrity during concurrent operations.

Additional Components:

1. APIs or Services:

- Implement RESTful APIs or microservices to interact with the database.

- APIs for CRUD operations on parking spots, vehicles, and transactions.

2. Real-time Messaging System:

- Use a message broker (e.g., Kafka, RabbitMQ) for real-time updates on parking spot availability.

- Publish events when vehicles enter or exit the parking lot.

3. Logging and Monitoring:

- Log system events, errors, and performance metrics.

- Monitor system health, resource usage, and database performance.

Scalability Considerations:

1. Horizontal Scaling:

- Deploy multiple instances of the backend service to handle increased load.

- Use load balancers to distribute incoming traffic across instances.

2. Database Sharding:

- Shard the database to distribute data across multiple nodes.

- Use techniques such as range-based or hash-based sharding to evenly distribute data.

Deployment Architecture:

1. Backend Servers:

- Deploy backend servers in a cloud environment (e.g., AWS, Azure).

- Use containerization (e.g., Docker, Kubernetes) for easy deployment and scalability.

2. Database:

- Choose a database system that supports high availability and scalability (e.g., PostgreSQL with replication, MongoDB with sharding).

3. Message Broker:

- Deploy a message broker (e.g., Kafka, RabbitMQ) for real-time messaging and event-driven architecture.

Security Considerations:

1. Authentication and Authorization:

- Implement authentication mechanisms (e.g., JWT, OAuth) to authenticate users accessing the system.

- Define roles and permissions to control access to sensitive operations (e.g., admin functions).

2. Data Encryption:

- Encrypt sensitive data (e.g., user credentials, transaction details) in transit and at rest.

- Use SSL/TLS for encrypting data in transit and encryption at the database level for data at rest.

Disaster Recovery and High Availability:

1. Backup and Restore:

- Implement regular backups of the database to prevent data loss in case of system failure.

- Test backup and restore procedures periodically to ensure data integrity.

2. High Availability Architecture:

- Deploy redundant components (e.g., load balancers, database replicas) to ensure high availability.

- Use failover mechanisms to automatically switch to backup components in case of failure.