Database Design and Relationships

1. Database Design Principles

Best Practices for Efficient and Scalable Database Schemas

1. Understand the Requirements

Real-Life Example:

- Entities:
  - **Books** (attributes: Title, Author, ISBN, Price, Genre)
  - **Customers** (attributes: Name, Email, Address, Phone)
  - **Orders** (attributes: OrderID, OrderDate, CustomerID)
  - **OrderDetails** (attributes: OrderID, BookID, Quantity)
- Relationships:
  - A customer can place multiple orders.
  - Each order can contain multiple books.

2. Normalization

Normalization: The Story of "Café Delight"

Imagine you are opening a new café called **Café Delight**. You want to use a database to track your orders, customers, and menu items. When you first design the database, it's a bit chaotic. Let's see how normalization can help us organize things.

1NF (First Normal Form): "No Messy Tables Allowed"

The Problem: You start with a table that looks like this:

OrderID	CustomerName	MenuItems	Quantities
1	Alice	Coffee, Croissant	1, 2
2	Bob	Sandwich, Orange Juice	1, 1

- The `MenuItems` and `Quantities` columns have multiple values (e.g., "Coffee, Croissant" and "1, 2").
- This format is messy and hard to query.

The Rule: In 1NF, **each column must contain atomic (indivisible) values**.

Solution: Split the table so that each piece of data is in its own row:

OrderID	CustomerName	MenuItem	Quantity
1	Alice	Coffee	1
1	Alice	Croissant	2
2	Bob	Sandwich	1
2	Bob	Orange Juice	1

Now each column contains only one value, and the table is in **1NF**.

2NF (Second Normal Form): "No Partial Dependencies"

The Problem: Your new table is much better, but it still has a problem. Let's say you extend it to include the price of each menu item:

OrderID	CustomerName	MenuItem	Quantity	Price
1	Alice	Coffee	1	$3.00
1	Alice	Croissant	2	$2.50
2	Bob	Sandwich	1	$5.00
2	Bob	Orange Juice	1	$4.00

Here's the issue:

- The `Price` of a menu item depends on the `MenuItem`, not the `OrderID`.
- If you update the price of "Coffee," you'd need to update it in multiple rows, increasing the chance of errors.

The Rule: In 2NF, **all non-key attributes must depend on the entire primary key**. (No partial dependencies!)

Solution: Break the table into two tables:

Break the table into two tables:

1. An `Orders` table that tracks orders and quantities:

OrderID	CustomerName	MenuItem	Quantity
1	Alice	Coffee	1
1	Alice	Croissant	2
2	Bob	Sandwich	1
2	Bob	Orange Juice	1

2. A `Menu` table that tracks menu items and their prices:

MenuItem	Price
Coffee	$3.00
Croissant	$2.50
Sandwich	$5.00
Orange Juice	$4.00

Now, the `Price` depends only on the `MenuItem`, and the table is in **2NF**.

3NF (Third Normal Form): "No Transitive Dependencies"

The Problem: Your database is getting better, but there's still room for improvement. Let's say you add a column to track each customer's phone number:

OrderID	CustomerName	MenuItem	Quantity	PhoneNumber
1	Alice	Coffee	1	123-456-7890
1	Alice	Croissant	2	123-456-7890
2	Bob	Sandwich	1	987-654-3210
2	Bob	Orange Juice	1	987-654-3210

Here's the issue:

- The `PhoneNumber` depends on the `CustomerName`, not directly on the `OrderID`.
- If Alice changes her phone number, you'd need to update it in multiple rows.

The Rule: In 3NF, **there should be no transitive dependencies**. (A non-key column should not depend on another non-key column.)

Solution: Split the table into three tables:

Split the table into three tables:

1. An `Orders` table:

OrderID	CustomerName	MenuItem	Quantity
1	Alice	Coffee	1
1	Alice	Croissant	2
2	Bob	Sandwich	1
2	Bob	Orange Juice	1

2. A `Menu` table (unchanged):

MenuItem	Price
Coffee	$3.00
Croissant	$2.50
Sandwich	$5.00
Orange Juice	$4.00

3. A `Customers` table:

CustomerName	PhoneNumber
Alice	123-456-7890
Bob	987-654-3210

Now, the `PhoneNumber` depends only on the `CustomerName`, and the table is in **3NF**.

Conclusion

Through **Normalization**:

- **1NF** ensures that data is organized into atomic values (no repeating groups).
- **2NF** eliminates partial dependencies by splitting data into smaller, logically related tables.
- **3NF** removes transitive dependencies, ensuring every non-key column depends only on the primary key.

With a properly normalized database, **Café Delight** can now handle updates, queries, and maintenance efficiently without redundant data.

3. Denormalization

Real-Life Example:

For reporting purposes, you might combine `Orders` and `Customers` into a single denormalized view (e.g., "OrdersWithCustomerDetails") for faster queries, accepting some redundancy for performance.

4. Primary Keys

Real-Life Example:

- **Books** table:
  - Primary Key: `BookID` (surrogate key, auto-incremented).
- Avoid composite keys like combining ISBN and Author, as they make the table harder to manage.

5. Foreign Keys

Real-Life Example:

- **Orders** table:
  - `CustomerID` (foreign key referencing the `Customers` table).
- **OrderDetails** table:
  - `OrderID` (foreign key referencing `Orders`).
  - `BookID` (foreign key referencing `Books`).

6. Avoid Redundancy

Real-Life Example:

- Instead of storing the author's name repeatedly in the `Books` table, create an `Authors` table and reference it with `AuthorID`.

7. Data Types

Real-Life Example:

- **Books** table:
  - `Price`: Use `DECIMAL(10,2)` for monetary values.
  - `Title`: Use `VARCHAR(255)` for variable-length titles.
  - `PublishedDate`: Use `DATE` for clarity and comparison.

8. Constraints

Real-Life Example:

- **Books** table:
  - `Price`: Use `CHECK (Price > 0)` to ensure valid pricing.
  - `Title`: Use `NOT NULL` to ensure every book has a title.
  - `ISBN`: Use `UNIQUE` to prevent duplicate entries for the same book.
  - `PublishedDate`: Use a `DEFAULT` value for books with unknown publish dates.

---

2. ER Diagram Notations

Entity-Relationship (ER) Diagram: Visual representation of entities, attributes, and their relationships.

Key Components:

- **Entities**: Represented by rectangles (e.g., User, Product).
- **Attributes**: Represented by ovals connected to entities (e.g., UserName, Email).
- **Relationships**: Represented by diamonds showing how entities are related.
- **Cardinality**:
  - **1:1**: A single entity instance relates to only one instance of another entity.
  - **1:N**: A single entity instance relates to multiple instances of another entity.
  - **N:M**: Multiple instances of an entity relate to multiple instances of another entity.

---

3. Relationships

One-to-One (1:1)

- **Definition**: One record in a table is associated with exactly one record in another table.
- **Example**:
  - Tables: `Users` and `UserProfiles`
  - Schema:
    - `Users`: UserID (PK), UserName
    - `UserProfiles`: ProfileID (PK, FK), UserID, Bio
  - Implementation:

```
CREATE TABLE Users (
    UserID INT PRIMARY KEY,
    UserName VARCHAR(50)
);

CREATE TABLE UserProfiles (
    ProfileID INT PRIMARY KEY,
    UserID INT UNIQUE,
    Bio TEXT,
    FOREIGN KEY (UserID) REFERENCES Users(UserID)
);
```

One-to-Many (1:N)

- **Definition**: One record in a table is associated with multiple records in another table.
- **Example**:
  - Tables: `Authors` and `Books`
  - Schema:
    - `Authors`: AuthorID (PK), Name
    - `Books`: BookID (PK), Title, AuthorID (FK)
  - Implementation:

```
CREATE TABLE Authors (
    AuthorID INT PRIMARY KEY,
    Name VARCHAR(100)
);

CREATE TABLE Books (
    BookID INT PRIMARY KEY,
```

```sql
    Title VARCHAR(100),
    AuthorID INT,
    FOREIGN KEY (AuthorID) REFERENCES Authors(AuthorID)
);
```

Many-to-Many (N:M)

- **Definition**: Multiple records in one table are associated with multiple records in another table.
- **Example**:
  - Tables: `Students`, `Courses`, and `StudentCourses`
  - Schema:
    - `Students`: StudentID (PK), Name
    - `Courses`: CourseID (PK), Title
    - `StudentCourses`: StudentID (FK), CourseID (FK)
  - Implementation:

```sql
CREATE TABLE Students (
    StudentID INT PRIMARY KEY,
    Name VARCHAR(100)
);

CREATE TABLE Courses (
    CourseID INT PRIMARY KEY,
    Title VARCHAR(100)
);

CREATE TABLE StudentCourses (
    StudentID INT,
    CourseID INT,
    PRIMARY KEY (StudentID, CourseID),
    FOREIGN KEY (StudentID) REFERENCES Students(StudentID),
    FOREIGN KEY (CourseID) REFERENCES Courses(CourseID)
);
```

---

4. Advanced Querying

Understanding and Implementing JOINs

- **INNER JOIN**:
  - Combines rows from both tables where a match exists.
  - Example:

```sql
SELECT Users.UserName, Orders.OrderID
FROM Users
INNER JOIN Orders ON Users.UserID = Orders.UserID;
```

- **LEFT JOIN**:
  - Retrieves all rows from the left table and matching rows from the right table.
  - Example:

```sql
SELECT Users.UserName, Orders.OrderID
FROM Users
LEFT JOIN Orders ON Users.UserID = Orders.UserID;
```

- **RIGHT JOIN**:
  - Retrieves all rows from the right table and matching rows from the left table.
  - Example:

```sql
SELECT Users.UserName, Orders.OrderID
FROM Users
RIGHT JOIN Orders ON Users.UserID = Orders.UserID;
```

---

5. Indexing

Techniques to Optimize Query Performance

1. **What is an Index?**
   - A data structure that speeds up the retrieval of rows by pointing to their locations.
2. **Types of Indexes**:
   - **Single-Column Index**: Index on a single column.
   - **Composite Index**: Index on multiple columns.
3. **Creating an Index**:
   - Syntax:

```sql
CREATE INDEX idx_column ON TableName(ColumnName);
```

4. **Benefits**:
    - Faster SELECT queries.
    - Optimized JOIN operations.
5. **Trade-offs**:
    - Slower INSERT, UPDATE, and DELETE operations.
    - Requires additional storage.
6. **Example**:

```sql
CREATE INDEX idx_author_name ON Authors(Name);
```

7. **EXPLAIN Keyword**:
    - Analyze query performance.
    - Example:

```sql
EXPLAIN SELECT * FROM Books WHERE AuthorID = 1;
```
