04 DB Normalization - advanced

Database Normalization in SQL

Introduction to Normalization

Normalization is a systematic process of organizing database tables to minimize data redundancy and dependency. This process improves data integrity, reduces storage requirements, and helps prevent update anomalies (insertion, deletion, and modification anomalies).

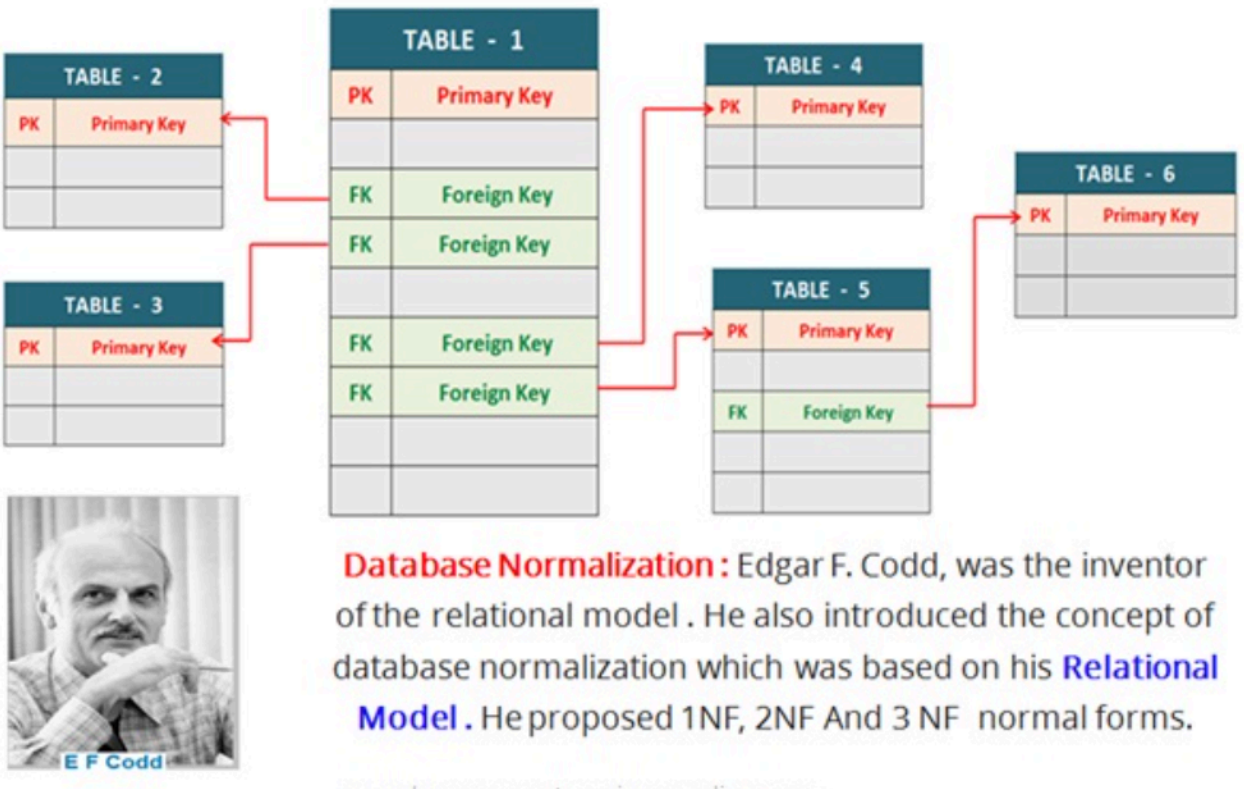
The normalization process involves breaking down large tables into smaller, related tables and defining relationships between them. This is done through a series of steps called normal forms.

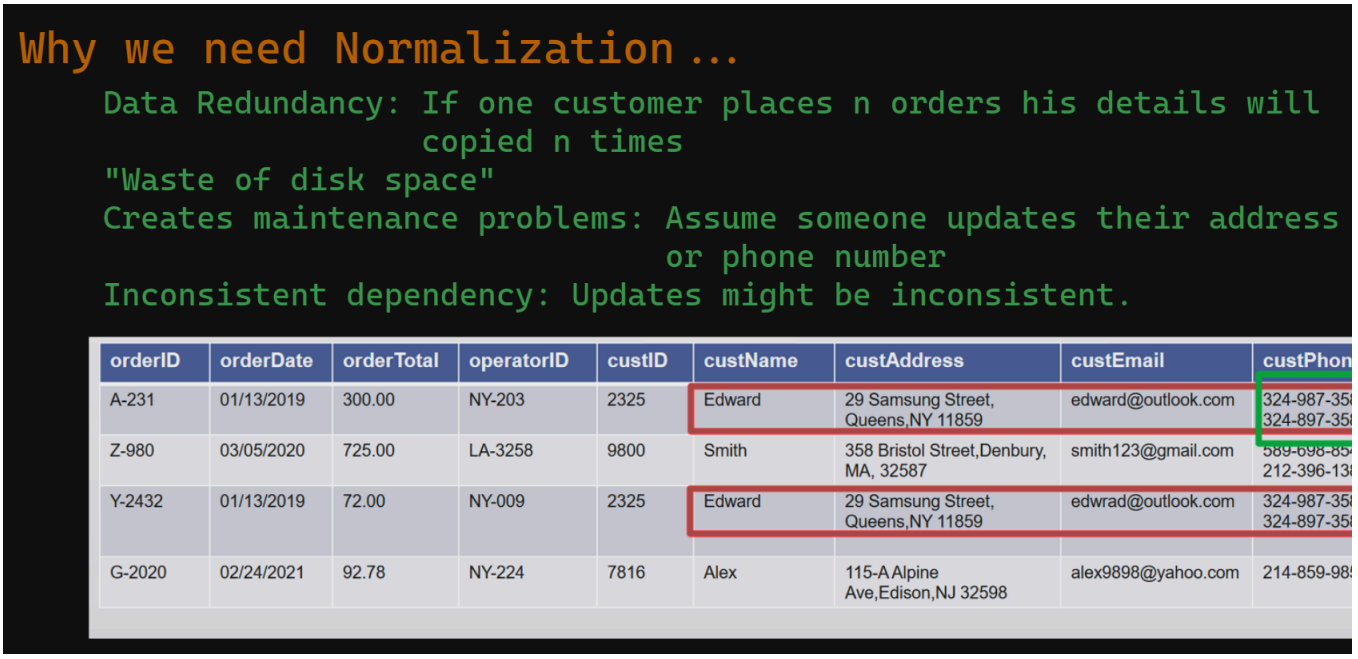
Why Normalize Databases?

Before diving into the normal forms, let's understand why normalization is important:

1. **Eliminate Redundant Data**: Reduce duplication of information
2. **Minimize Data Modification Issues**: Prevent anomalies when inserting, updating, or deleting data
3. **Simplify Queries**: Make database queries more straightforward and efficient
4. **Improve Database Structure**: Create a logical, modular design that's easier to understand and maintain

---





orderID	orderDate	orderTotal	operatorID	custID	custName	custAddress	custEmail	custPhon
A-231	01/13/2019	300.00	NY-203	2325	Edward	29 Samsung Street, Queens,NY 11859	edward@outlook.com	324-987-35 324-897-35
Z-980	03/05/2020	725.00	LA-3258	9800	Smith	358 Bristol Street,Denbury, MA, 32587	smith123@gmail.com	589-698-85 212-396-13
Y-2432	01/13/2019	72.00	NY-009	2325	Edward	29 Samsung Street, Queens,NY 11859	edwrad@outlook.com	324-987-35 324-897-35
G-2020	02/24/2021	92.78	NY-224	7816	Alex	115-A Alpine Ave,Edison,NJ 32598	alex9898@yahoo.com	214-859-98

Database Anomalies

The database anomalies are the problems caused due to the presence of duplicate data (redundant data) in the large tables. The database anomalies is the root cause of inconsistent state of the database . The database queries and operations fail to produce the correct results due the inconsistent
state of the database caused by various database anomalies.

The redundant data creates many anomalies that include:
a. Insert Anomaly
b. Update Anomaly
c. Delete Anomaly

How to remove db anomalies?

- Remove Redundant data.
- Normalize all tables within the database.

First Normal Form (1NF)

Definition

A table is in 1NF if:

- It has a primary key
- All columns contain atomic (indivisible) values
- No repeating groups or arrays
- All entries in a column are of the same data type

Problem Example

Consider an un-normalized table storing customer orders:

```SQL
CREATE TABLE customer_orders_unnormalized (
    customer_id INT,
    customer_name VARCHAR(100),
    order_date DATE,
    product_list VARCHAR(255),
    PRIMARY KEY (customer_id, order_date)
);

INSERT INTO customer_orders_unnormalized VALUES
(1, 'John Smith', '2025-01-15', 'Laptop, Mouse, Keyboard'),
(2, 'Mary Johnson', '2025-01-16', 'Headphones, Phone Case'),
(1, 'John Smith', '2025-01-20', 'Monitor, HDMI Cable');
```

Issues with this design:

- `product_list` contains multiple values (non-atomic)
- Cannot efficiently search for specific products
- Difficult to handle individual products

1NF Solution

```sql
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100)
);

CREATE TABLE orders (
    order_id INT PRIMARY KEY AUTO_INCREMENT,
    customer_id INT,
    order_date DATE,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

CREATE TABLE order_items (
    order_id INT,
    product_name VARCHAR(100),
    PRIMARY KEY (order_id, product_name),
    FOREIGN KEY (order_id) REFERENCES orders(order_id)
);

-- Insert data
INSERT INTO customers VALUES
(1, 'John Smith'),
(2, 'Mary Johnson');

INSERT INTO orders (customer_id, order_date) VALUES
(1, '2025-01-15'),
(2, '2025-01-16'),
(1, '2025-01-20');

-- Assuming order_id 1, 2, 3 were generated
INSERT INTO order_items VALUES
(1, 'Laptop'),
(1, 'Mouse'),
(1, 'Keyboard'),
(2, 'Headphones'),
(2, 'Phone Case'),
(3, 'Monitor'),
(3, 'HDMI Cable');
```

Second Normal Form (2NF)

Definition

A table is in 2NF if:

- It is in 1NF
- All non-key attributes are fully functionally dependent on the primary key
- No partial dependencies (where a non-key attribute depends on only part of the primary key)

Problem Example

Consider a table tracking student course enrollment:

```SQL
CREATE TABLE enrollments_not_2nf (
    student_id INT,
    course_id INT,
    student_name VARCHAR(100),
    course_name VARCHAR(100),
    instructor_name VARCHAR(100),
    grade CHAR(1),
    PRIMARY KEY (student_id, course_id)
);

INSERT INTO enrollments_not_2nf VALUES
(101, 301, 'Alice Cooper', 'Database Systems', 'Dr. Smith', 'A'),
(101, 302, 'Alice Cooper', 'Web Development', 'Dr. Jones', 'B'),
(102, 301, 'Bob Miller', 'Database Systems', 'Dr. Smith', 'B');
```

Issues with this design:

- `student_name` depends only on `student_id` (partial dependency)
- `course_name` and `instructor_name` depend only on `course_id` (partial dependency)
- Only `grade` depends on the full composite key

2NF Solution

```SQL
CREATE TABLE students (
    student_id INT PRIMARY KEY,
    student_name VARCHAR(100)
);

CREATE TABLE courses (
    course_id INT PRIMARY KEY,
    course_name VARCHAR(100),
    instructor_name VARCHAR(100)
);

CREATE TABLE enrollments (
    student_id INT,
    course_id INT,
    grade CHAR(1),
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY (student_id) REFERENCES students(student_id),
    FOREIGN KEY (course_id) REFERENCES courses(course_id)
);

-- Insert data
INSERT INTO students VALUES
(101, 'Alice Cooper'),
(102, 'Bob Miller');

INSERT INTO courses VALUES
(301, 'Database Systems', 'Dr. Smith'),
(302, 'Web Development', 'Dr. Jones');

INSERT INTO enrollments VALUES
(101, 301, 'A'),
(101, 302, 'B'),
(102, 301, 'B');
```

Third Normal Form (3NF)

Definition

A table is in 3NF if:

- It is in 2NF
- No transitive dependencies (non-primary key attributes depending on other non-primary key attributes)

- All attributes depend directly on the primary key

Problem Example

Consider an employee table:

```SQL
CREATE TABLE employees_not_3nf (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    department_id INT,
    department_name VARCHAR(100),
    department_location VARCHAR(100)
);

INSERT INTO employees_not_3nf VALUES
(1001, 'John Doe', 10, 'Engineering', 'Building A'),
(1002, 'Jane Smith', 20, 'Marketing', 'Building B'),
(1003, 'Mike Johnson', 10, 'Engineering', 'Building A');
```

Issues with this design:

- `department_name` and `department_location` depend on `department_id`, not directly on the primary key `employee_id` (transitive dependency)
- Department information is duplicated for employees in the same department

3NF Solution

```sql
CREATE TABLE departments (
    department_id INT PRIMARY KEY,
    department_name VARCHAR(100),
    department_location VARCHAR(100)
);

CREATE TABLE employees (
    employee_id INT PRIMARY KEY,
    employee_name VARCHAR(100),
    department_id INT,
    FOREIGN KEY (department_id) REFERENCES departments(department_id)
);

-- Insert data
INSERT INTO departments VALUES
(10, 'Engineering', 'Building A'),
(20, 'Marketing', 'Building B');

INSERT INTO employees VALUES
(1001, 'John Doe', 10),
(1002, 'Jane Smith', 20),
(1003, 'Mike Johnson', 10);
```

Boyce-Codd Normal Form (BCNF)

Definition

A table is in BCNF if:

- It is in 3NF
- For every dependency X → Y, X must be a superkey
- In simpler terms, every determinant must be a candidate key

Problem Example

Consider a table representing student advisors for different subjects:

```sql
CREATE TABLE student_advisors_not_bcnf (
    student_id INT,
    subject VARCHAR(50),
    advisor_id INT,
    PRIMARY KEY (student_id, subject),
    UNIQUE (student_id, advisor_id)
);

-- Each student has only one advisor per subject
-- But also, each student-advisor pair works on only one subject
INSERT INTO student_advisors_not_bcnf VALUES
(101, 'Math', 201),
(101, 'Physics', 202),
(102, 'Math', 203),
(103, 'Chemistry', 201);
```

Issues with this design:

- There's a functional dependency: `(student_id, advisor_id) → subject`
- But `(student_id, advisor_id)` is not a superkey (though it has a unique constraint)
- This creates a situation where updating one row might require updating multiple rows to maintain consistency

BCNF Solution

```sql
CREATE TABLE student_advisor_assignments (
    student_id INT,
    advisor_id INT,
    PRIMARY KEY (student_id, advisor_id)
);

CREATE TABLE advisor_subjects (
    student_id INT,
    advisor_id INT,
    subject VARCHAR(50),
    PRIMARY KEY (student_id, subject),
    FOREIGN KEY (student_id, advisor_id) REFERENCES
student_advisor_assignments(student_id, advisor_id)
);

-- Insert data
INSERT INTO student_advisor_assignments VALUES
(101, 201),
(101, 202),
(102, 203),
(103, 201);

INSERT INTO advisor_subjects VALUES
(101, 201, 'Math'),
(101, 202, 'Physics'),
(102, 203, 'Math'),
(103, 201, 'Chemistry');
```

Real-World Example: E-commerce Database

Let's build a simplified e-commerce database that demonstrates normalization principles:

Un-normalized Version (For Illustration)

```SQL
CREATE TABLE ecommerce_unnormalized (
    order_id INT,
    order_date DATE,
    customer_id INT,
    customer_name VARCHAR(100),
    customer_email VARCHAR(100),
    customer_address VARCHAR(255),
    product_id INT,
    product_name VARCHAR(100),
    product_description TEXT,
    category_id INT,
    category_name VARCHAR(50),
    quantity INT,
    unit_price DECIMAL(10,2),
    PRIMARY KEY (order_id, product_id)
);
```

Normalized E-commerce Database (3NF)

SQL

```sql
-- Customers table
CREATE TABLE customers (
    customer_id INT PRIMARY KEY,
    customer_name VARCHAR(100),
    customer_email VARCHAR(100),
    customer_address VARCHAR(255)
);

-- Product categories
CREATE TABLE categories (
    category_id INT PRIMARY KEY,
    category_name VARCHAR(50)
);

-- Products table
CREATE TABLE products (
    product_id INT PRIMARY KEY,
    product_name VARCHAR(100),
    product_description TEXT,
    category_id INT,
    unit_price DECIMAL(10,2),
    FOREIGN KEY (category_id) REFERENCES categories(category_id)
);

-- Orders table
CREATE TABLE orders (
    order_id INT PRIMARY KEY,
    order_date DATE,
    customer_id INT,
    FOREIGN KEY (customer_id) REFERENCES customers(customer_id)
);

-- Order details table
CREATE TABLE order_details (
    order_id INT,
    product_id INT,
    quantity INT,
    PRIMARY KEY (order_id, product_id),
    FOREIGN KEY (order_id) REFERENCES orders(order_id),
    FOREIGN KEY (product_id) REFERENCES products(product_id)
);
```

Sample Data and Queries for the E-commerce Database

Sample Data Insertion

```SQL
-- Insert categories
INSERT INTO categories VALUES
(1, 'Electronics'),
(2, 'Books'),
(3, 'Clothing');

-- Insert products
INSERT INTO products VALUES
(101, 'Smartphone', 'High-end smartphone with 128GB storage', 1, 699.99),
(102, 'Laptop', '15-inch laptop with SSD', 1, 1299.99),
(103, 'SQL for Beginners', 'Learn SQL from scratch', 2, 29.99),
(104, 'T-shirt', 'Cotton t-shirt, available in multiple colors', 3, 19.99);

-- Insert customers
INSERT INTO customers VALUES
(1, 'Alice Johnson', 'alice@example.com', '123 Main St, Anytown'),
(2, 'Bob Smith', 'bob@example.com', '456 Oak Ave, Somewhere');

-- Insert orders
INSERT INTO orders VALUES
(1001, '2025-02-01', 1),
(1002, '2025-02-05', 2),
(1003, '2025-02-10', 1);

-- Insert order details
INSERT INTO order_details VALUES
(1001, 101, 1),
(1001, 103, 2),
(1002, 102, 1),
(1003, 104, 3),
(1003, 103, 1);
```

Sample Queries That Benefit from Normalization

Get complete order information with customer and product details:

```sql
SELECT
    o.order_id,
    o.order_date,
    c.customer_name,
    p.product_name,
    od.quantity,
    p.unit_price,
    (od.quantity * p.unit_price) AS subtotal
FROM
    orders o
JOIN
    customers c ON o.customer_id = c.customer_id
JOIN
    order_details od ON o.order_id = od.order_id
JOIN
    products p ON od.product_id = p.product_id
ORDER BY
    o.order_id, p.product_name;
```

Calculate total revenue per category:

```sql
SELECT
    cat.category_name,
    SUM(od.quantity * p.unit_price) AS total_revenue
FROM
    categories cat
JOIN
    products p ON cat.category_id = p.category_id
JOIN
    order_details od ON p.product_id = od.product_id
GROUP BY
    cat.category_name
ORDER BY
    total_revenue DESC;
```

Find top customers by purchase amount:

```sql
SELECT
    c.customer_id,
    c.customer_name,
    SUM(od.quantity * p.unit_price) AS total_spent
FROM
    customers c
JOIN
    orders o ON c.customer_id = o.customer_id
JOIN
    order_details od ON o.order_id = od.order_id
JOIN
    products p ON od.product_id = p.product_id
GROUP BY
    c.customer_id, c.customer_name
ORDER BY
    total_spent DESC;
```

Benefits of Normalization

1. **Reduced Data Redundancy**: Each piece of information is stored in only one place.
2. **Improved Data Integrity**: Reduces the chance of inconsistent data.
3. **Smaller Database Size**: Less duplication means less storage required.
4. **Better Performance for Writes**: Smaller, more focused tables can make inserts, updates, and deletes faster.
5. **More Flexible Database Design**: Easier to extend and modify the database structure.
6. **Better Data Relationships**: Clear relationships between entities.

When to Consider Denormalization

While normalization provides many benefits, there are situations where some level of denormalization might be appropriate:

1. **Read-heavy applications**: When performance of complex read queries is critical
2. **Reporting databases**: For analytical purposes and complex reports
3. **Data warehouses**: For historical data analysis
4. **When joins become too expensive**: If the performance cost of joins outweighs the benefits of normalization

Practical Normalization Process

1. Identify the entities and attributes in your data
2. Create an initial table design
3. Apply 1NF by eliminating repeating groups
4. Apply 2NF by removing partial dependencies
5. Apply 3NF by removing transitive dependencies

6. Consider BCNF for complex scenarios
7. Test the design with real-world queries and data

Conclusion

Database normalization is a fundamental concept in database design that helps create efficient, maintainable, and consistent databases. By following the normalization process, you can structure your data in a way that minimizes redundancy and dependency issues while maintaining data integrity.

Remember that while higher normal forms offer better theoretical data integrity, the practical needs of your application should guide the level of normalization you implement. In some cases, a balance between normalization and performance considerations might be necessary.

Additional Resources

For more information on database normalization and MySQL best practices:

- MySQL Documentation: https://dev.mysql.com/doc/
- Database Design Books: "Database Design for Mere Mortals" by Michael J. Hernandez