Web Programming
Databases 3: Constraints, Normalization, Indexing & Transactions

Leander Jehl | University of Stavanger

Part III
Constraint

Learning Objectives

- Understand and apply SQL constraints.
- Learn about foreign keys and their role in data integrity.
- Gain an understanding of database normalization.
- Learn to connect multiple tables using joins.
- Briefly discuss indexing and transactions.

What are Constraints?

Constraints enforce rules to maintain data integrity.

Types of Constraints:

- **NOT NULL**: Prevents null values in a column.
- **UNIQUE**: Ensures unique values.
- **DEFAULT**: Assigns default values.
- **CHECK**: Validates data based on a condition.
- **PRIMARY KEY**: Uniquely identifies a record.
- **FOREIGN KEY**: Links tables together.

What are Constraints?

Constraints enforce rules to maintain data integrity.

Types of Constraints:

- **NOT NULL**: Prevents null values in a column.

```
CREATE TABLE students (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);
```

What are Constraints?

Constraints enforce rules to maintain data integrity.

Types of Constraints:

- **UNIQUE**: Ensures unique values.

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    email TEXT UNIQUE
);
```

What are Constraints?

Constraints enforce rules to maintain data integrity.

Types of Constraints:

- **DEFAULT**: Assigns default values.

```
CREATE TABLE orders (
    id INTEGER PRIMARY KEY,
    status TEXT DEFAULT ‘Pending’
);
```

What are Constraints?

Constraints enforce rules to maintain data integrity.

Types of Constraints:

- **CHECK**: Validates data based on a condition.

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    age INTEGER CHECK(age >= 18)
);
```

What are Constraints?

Constraints enforce rules to maintain data integrity.

Types of Constraints:

- **PRIMARY KEY**: Uniquely identifies a record.

```
CREATE TABLE departments (
    id INTEGER PRIMARY KEY,
    name TEXT NOT NULL
);
```

What are Constraints?

Constraints enforce rules to maintain data integrity.

Types of Constraints:

- **FOREIGN KEY**: Links tables together.

```
CREATE TABLE orders (
    id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    FOREIGN KEY (customer_id ) REFERENCES costumers(id)
);
```

Column Constraints in SQLite

SQLite does not strictly enforce data types but has type affinity.

Example:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY,
    username TEXT UNIQUE NOT NULL,
    age INTEGER CHECK(age>=18)
);
```

Exercise: Insert invalid data and analyze errors.

```
INSERT INTO users (id, username, age) VALUES (1, 'JohnDoe', 17); -- Should fail due to CHECK constraint
```

Foreign Keys and Referential Integrity

SQLite does not enforce foreign keys by default.

To enable:

```
INSERT INTO users (id, username, age) VALUES (1, 'JohnDoe', 17); -- Should fail due to CHECK constraint
```

Example of a Foreign key

```
CREATE TABLE orders (
    id INTEGER PRIMARY KEY,
    customer_id INTEGER,
    FOREIGN KEY (customer_id) REFERENCES customers(id)
);
```

Cascading Actions

What happens if we UPDATE or DELETE an entry from one of the linked tables.

To enable:

```
ON DELETE CASCADE -- (Delete related records automatically)

ON UPDATE SET NULL
```

Example

```
CREATE TABLE enrollments (
    student_id INTEGER,
    course_id INTEGER,
    PRIMARY KEY (student_id, course_id),
    FOREIGN KEY(student_id) REFERENCES students(id) ON DELETE CASCADE
);
```

Part III
Normalization

Database Normalization

Why Normalize?

- Reduces redundancy and improves consistency.
- Improves query performance and maintains data integrity.

Not-Normalized Database Table

STUDENT

StudentID	StudentName	MajorName	NoOfCreditHours
111	Kirsten	Accounting	152
222	Eve	IS	138
333	Zoe	IS	138
444	Ben	Accounting	152

Normalized Database Tables

MAJOR

MajorName	NoOfCreditHours
Accounting	152
IS	138

STUDENT

StudentID	StudentName	MajorName
111	Kirsten	Accounting
222	Eve	IS
333	Zoe	IS
444	Ben	Accounting

First Normal Form (1NF)

Each column contains atomic values (no lists or arrays).

- **Example**

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    skill TEXT  - - Bad Design (should be separate tables)
);
```

- **Better**

```
CREATE TABLE employee_skills (
    employee_id INTEGER,
    skill TEXT,
    FOREIGN KEY (employee_id) REFERENCES employees(id)
);
```

Second Normal Form (2NF)

Removes partial dependencies.

- **Example**: Split employee table into employee and department tables.

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department_id INTEGER,
    FOREIGN KEY (department_id) REFERENCES departments(id)
);
```

Third Normal Form (3NF)

No transitive dependencies (remove indirect relationships).

- **Example**: Separate city and country from an address table.

```
CREATE TABLE addresses (
    id INTEGER PRIMARY KEY,
    city TEXT,
    country TEXT
);
```

- **Exercise:** Given a denormalized table, normalize it step by step.

Exercise #1

Given a denormalized table, normalize it step by step.

1. **Denormalized Table**

```
CREATE TABLE employee_details (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department TEXT,
    manager TEXT,
    skills TEXT
);
```

2. **Step 1 (1NF - Remove Multi-Valued Columns)**

```
CREATE TABLE employees (
    id INTEGER PRIMARY KEY,
    name TEXT,
    department TEXT,
    manager TEXT
);

CREATE TABLE employee_skills (
    employee_id INTEGER,
    skill TEXT,
    FOREIGN KEY (employee_id) REFERENCES employees(id)
);
```

Exercise #1

Given a denormalized table, normalize it step by step.

3. **Step 2 (2NF - Remove Partial Dependencies)**:

```
CREATE TABLE departments (
   id INTEGER PRIMARY KEY,
   name TEXT,
   manager TEXT
);
ALTER TABLE employees ADD COLUMN department_id INTEGER;
UPDATE employees SET department_id = (SELECT id FROM departments WHERE name = employees.department);
ALTER TABLE employees DROP COLUMN department;
```

4. **Step 3 (3NF - Remove Transitive Dependencies)**:

```
CREATE TABLE managers (
    id INTEGER PRIMARY KEY,
    name TEXT
);
ALTER TABLE departments ADD COLUMN manager_id INTEGER;
UPDATE departments SET manager_id = (SELECT id FROM managers WHERE name = departments.manager);
ALTER TABLE departments DROP COLUMN manager;
```

Exercise #1

- Insert sample data into denormalized table and normalize it step by step.

```
INSERT INTO departments (id, name) VALUES (1, 'Engineering'), (2, 'HR’);
INSERT INTO employees (id, name, department_id) VALUES (1, 'Alice', 1), (2, 'Bob', 2);
INSERT INTO managers (id, name) VALUES (1, 'Charlie'), (2, 'Dana’);
UPDATE departments SET manager_id = 1 WHERE id = 1;
UPDATE departments SET manager_id = 2 WHERE id = 2;
INSERT INTO employee_skills (employee_id, skill) VALUES (1, 'Python'), (1, 'SQL'), (2, 'HR Management');
```

- Write queries using INNER and LEFT JOIN to retrieve meaningful insights.
- Compare query performance before and after indexing.
- Simulate a transaction with rollback and commit.

Part III
Indexing in SQL

Indexing

What is Indexing?

- Indexing improves query performance by allowing the database to find rows faster.
- Works like an index in a book: instead of scanning the entire table, the database uses the index to jump directly to the data.

Types of Indexes

- **Primary Index**: Automatically created for primary keys.
- **Unique Index**: Enforces unique values.

```
CREATE UNIQUE INDEX idx_users_email ON users(email);
```

- **Composite Index**: Index on multiple columns.

```
CREATE INDEX idx_employee_dept ON employees(department_id, name);
```

- **Full-Text Index**: Used for searching text fields (not supported in SQLite).

Trade-Offs of Indexing

Pros:

1. Speeds up searches and queries.
2. Enhances efficiency for large datasets.

Cons:

1. Slows down INSERT, UPDATE, and DELETE operations.
2. Takes up additional storage space.

Part III
Additional SQL Topics

Auto-Increment in SQL

Used to automatically generate unique values for a primary key.

Example:

```
CREATE TABLE users (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL
);
```

When inserting a row, SQLite automatically assigns the next available ID.

Handling Dates in SQL

SQLite does not have a dedicated DATE type but supports storing dates as:

- *TEXT* (ISO 8601 format YYYY-MM-DD HH:MM:SS)
- *INTEGER* (Unix timestamp)
- *REAL* (Julian day number)
- **Example**

```
CREATE TABLE events (
    id INTEGER PRIMARY KEY,
    event_name TEXT,
    event_date TEXT DEFAULT CURRENT_TIMESTAMP
);
```

- Extracting Date Parts

```
SELECT strftime('%Y', event_date) AS year FROM events;
```

SQL Views

Views are virtual tables that simplify complex queries.

- **Example**

```
CREATE VIEW employee_details AS
SELECT employees.name, departments.name AS department
FROM employees
JOIN departments ON employees.department_id = departments.id;
```

- **Querying a View**

```
SELECT * FROM employee_details;
```

SQL Injection and Security

What is SQL Injection?

A technique where malicious SQL statements are inserted into an input field.

- **Example** of a vulnerable query:

```
SELECT * FROM users WHERE username = 'admin' AND password = ‘  ' OR '1' = '1';
```

Preventing SQL Injection

- Use parameterized queries:

```
SELECT * FROM users WHERE username = ? AND password = ?;
```

Hosting a Database

SQLite is serverless, but other databases like PostgreSQL and MySQL require a database server. Common hosting solutions:

- Local Development: SQLite, MySQL, PostgreSQL.
- Cloud-Based Solutions: AWS RDS, Google Cloud SQL, Azure SQL Database.