

# **Lesson 5: Multiple Tables and Relationships**

---

**Duration:** 20 minutes

**Deliverables:** `lesson5_schema.sql`, `lesson5_data.sql`

## Learning Objectives

---

By the end of this lesson, you will be able to:

- Understand database normalization and why it matters
- Design one-to-many relationships
- Create related tables with foreign keys
- Link tables together properly
- Avoid data duplication



## Why Use Multiple Tables?

---

Currently, our `characters` table stores homeworld as TEXT. This creates problems:

### Problems with Single Table Design:

1. **Duplication:** "Tatooine" is stored multiple times
2. **Inconsistency:** One entry might say "Tatooine", another "tatooine"
3. **Limited information:** We can't store climate, population, etc.
4. **Update issues:** Changing planet name requires updating many rows

**Solution:** Use **multiple related tables!**



# Understanding Database Relationships

---

## One-to-Many Relationship

**Definition:** One record in Table A relates to many records in Table B.

### Examples:

- One planet → Many characters (many characters from one planet)
- One character → Many vehicles (one character pilots many vehicles)
- One vehicle → Many characters (many characters pilot one vehicle - this is many-to-many!)

## Primary Key vs Foreign Key

Key Type	Purpose	Example
<b>Primary Key</b>	Uniquely identifies each row in a table	<code>id</code> in <code>planets</code> table
<b>Foreign Key</b>	References a primary key in another table	<code>homeworld_id</code> in <code>characters</code> table

## Part 1: Creating the Planets Table (5 minutes)

### Step 1: Create Schema File

1. Navigate to `lessons/` folder
2. Create: `lesson5_schema.sql`
3. Add header:

```
-- Lesson 5: Multiple Tables and Relationships (Schema)
-- Student Name: [Your Name]
-- Date: [Today's Date]
--
-- This script creates related tables with foreign keys
```

### Step 2: Create the Planets Table

```
-- Create planets table
CREATE TABLE IF NOT EXISTS planets (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL UNIQUE,
    climate TEXT,
    terrain TEXT,
    population INTEGER
);
```

**New Concept:** `UNIQUE` constraint ensures no duplicate planet names.

### Step 3: Create the Vehicles Table

```
-- Create vehicles table
CREATE TABLE IF NOT EXISTS vehicles (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name TEXT NOT NULL,
    model TEXT,
    vehicle_class TEXT,
    manufacturer TEXT
);
```

### Step 4: Create the Junction Table

Many-to-many relationships need a **junction table** (also called a linking table or associative table).

```
-- Create character_vehicles junction table
CREATE TABLE IF NOT EXISTS character_vehicles (
    character_id INTEGER NOT NULL,
    vehicle_id INTEGER NOT NULL,
    PRIMARY KEY (character_id, vehicle_id),
    FOREIGN KEY (character_id) REFERENCES characters(id),
    FOREIGN KEY (vehicle_id) REFERENCES vehicles(id)
);
```

### **Explanation:**

- character\_id references the characters table
- vehicle\_id references the vehicles table
- PRIMARY KEY (character\_id, vehicle\_id) ensures each pairing is unique
- This allows many characters to pilot many vehicles

### **Step 5: Modify Characters Table**

We need to change homeworld from TEXT to a foreign key:

```
-- Add homeworld_id column
ALTER TABLE characters ADD COLUMN homeworld_id INTEGER;

-- Add foreign key constraint (Note: SQLite has limited ALTER TABLE support)
-- In production, you'd recreate the table with FOREIGN KEY constraint
```

**Note:** SQLite doesn't support adding foreign key constraints to existing columns via ALTER TABLE. In practice, you'd recreate the table.

**Execute all schema queries** to create the tables.



## Part 2: Inserting Related Data (10 minutes)

### Step 6: Create Data File

1. Create: lesson5\_data.sql
2. Add header:

```
-- Lesson 5: Multiple Tables and Relationships (Data)
-- Student Name: [Your Name]
-- Date: [Today's Date]
--
-- This script inserts data into related tables
```

### Step 7: Insert Planets

```
-- Insert planets
INSERT INTO planets (name, climate, terrain, population) VALUES
    ('Tatooine', 'arid', 'desert', 200000),
    ('Alderaan', 'temperate', 'grasslands', 'mountains', 2000000000),
    ('Hoth', 'frozen', 'tundra', 'ice caves', NULL),
    ('Kashyyyk', 'tropical', 'jungle', 'forests', 45000000),
    ('Naboo', 'temperate', 'grassy hills', 'swamps', 4500000000),
    ('Corellia', 'temperate', 'plains', 'urban', 3000000000),
    ('Stewjon', 'temperate', 'grass', NULL),
    ('Unknown', NULL, NULL, NULL);
```

### Step 8: Update Characters with Planet IDs

Now we link characters to planets using foreign keys:

```
-- Update characters with homeworld_id
UPDATE characters SET homeworld_id = (SELECT id FROM planets WHERE name = 'Tatooine')
UPDATE characters SET homeworld_id = (SELECT id FROM planets WHERE name = 'Alderaan')
UPDATE characters SET homeworld_id = (SELECT id FROM planets WHERE name = 'Corellia')
UPDATE characters SET homeworld_id = (SELECT id FROM planets WHERE name = 'Kashyyyk')
UPDATE characters SET homeworld_id = (SELECT id FROM planets WHERE name = 'Stewjon')
UPDATE characters SET homeworld_id = (SELECT id FROM planets WHERE name = 'Naboo')
UPDATE characters SET homeworld_id = (SELECT id FROM planets WHERE name = 'Unknown')
```

**Explanation:** These subqueries find the planet's ID and update the character's homeworld\_id.

## Step 9: Insert Vehicles

```
-- Insert vehicles
INSERT INTO vehicles (name, model, vehicle_class, manufacturer) VALUES
    ('X-wing', 'T-65 X-wing', 'Starfighter', 'Incom Corporation'),
    ('Millennium Falcon', 'YT-1300 light freighter', 'Light freighter', 'Corellian En
('TIE Fighter', 'Twin Ion Engine Fighter', 'Starfighter', 'Sienar Fleet Systems')
('Imperial Speeder Bike', '74-Z speeder bike', 'Speeder', 'Aratech Repulsor Compa
('Snowspeeder', 'T-47 airspeeder', 'Airspeeder', 'Incom Corporation'),
    ('Lambda Shuttle', 'Lambda-class shuttle', 'Transport', 'Sienar Fleet Systems'),
    ('AT-AT', 'All Terrain Armoured Transport', 'Assault walker', 'Kuat Drive Yards')
    ('Jedi Starfighter', 'Delta-7 Aethersprite', 'Starfighter', 'Kuat Systems Enginee
```

## Step 10: Link Characters to Vehicles

```
-- Link characters to vehicles (many-to-many relationship)
INSERT INTO character_vehicles (character_id, vehicle_id) VALUES
    -- Luke flies X-wing, Snowspeeder
    (1, 1),
    (1, 5),
    -- Han flies Millennium Falcon
    (3, 2),
    -- Chewbacca also flies Millennium Falcon
    (4, 2),
    -- Obi-Wan flies Jedi Starfighter
    (5, 8),
    -- Darth Vader flies TIE Fighter, Lambda Shuttle
    (6, 3),
    (6, 6),
    -- Yoda flies... nothing (wise, he walks)
    -- R2-D2 is IN X-wing and Jedi Starfighter
    (8, 1),
    (8, 8);
```

**Note:** We use character `id` and vehicle `id` values. Check your IDs with `SELECT * FROM characters;` if unsure.

**Execute all data insertion queries.**



## Part 3: Verifying Relationships (5 minutes)

### Step 11: View All Tables

```
-- View all planets  
SELECT * FROM planets;  
  
-- View all vehicles  
SELECT * FROM vehicles;  
  
-- View character-vehicle links  
SELECT * FROM character_vehicles;  
  
-- View updated characters table  
SELECT id, name, homeworld, homeworld_id FROM characters;
```

### Step 12: Understanding Foreign Keys

A **foreign key** is a field that links to another table's primary key.

#### Benefits:

- **Data integrity:** Can't reference non-existent records
- **Consistency:** One source of truth for planet data
- **Efficiency:** Store planet data once, reference many times
- **Easy updates:** Change planet name in one place

#### Example:

```
characters table:  
id | name           | homeworld_id  
1  | Luke Skywalker| 1  
  
planets table:  
id | name           | climate | population  
1  | Tatooine       | arid    | 200000
```

Luke's `homeworld_id = 1` links to Tatooine (`id = 1`) in the planets table.



## Part 4: Normalization Explained

**Normalization** is the process of organizing data to reduce redundancy.

### Before (One Table):

id	name	homeworld	climate	population
1	Luke Skywalker	Tatooine	arid	200000
2	Darth Vader	Tatooine	arid	200000

- "Tatooine", "arid", "200000" stored twice (duplication!)

### After (Two Tables):

characters:

id	name	homeworld_id
1	Luke Skywalker	1
2	Darth Vader	1

planets:

id	name	climate	population
1	Tatooine	arid	200000

- Planet data stored once, referenced multiple times!

## Practice Exercise

Add 2 more planets and update some characters to be from those planets.  
Add 2 more vehicles and link them to characters.

-- Practice: Add more planets

```
INSERT INTO planets (name, climate, terrain, population) VALUES
    ('Your Planet 1', 'climate', 'terrain', population_number),
    ('Your Planet 2', 'climate', 'terrain', population_number);
```

-- Practice: Add more vehicles

```
INSERT INTO vehicles (name, model, vehicle_class, manufacturer) VALUES
    ('Your Vehicle 1', 'model', 'class', 'manufacturer'),
    ('Your Vehicle 2', 'model', 'class', 'manufacturer');
```

-- Practice: Link vehicles to characters

```
INSERT INTO character_vehicles (character_id, vehicle_id) VALUES
    (character_id, vehicle_id),
    (character_id, vehicle_id);
```



## Common Errors & Troubleshooting

---

### Error: "FOREIGN KEY constraint failed"

**Problem:** Trying to insert a character\_id or vehicle\_id that doesn't exist.

**Solution:** Verify the IDs exist:

```
SELECT id, name FROM characters;  
SELECT id, name FROM vehicles;
```

### Error: "UNIQUE constraint failed: planets.name"

**Problem:** Trying to insert a planet that already exists.

**Solution:** Check existing planets first:

```
SELECT name FROM planets;
```

### Error: "no such table: planets"

**Problem:** Haven't created the planets table yet.

**Solution:** Run the CREATE TABLE statements from `lesson5_schema.sql` first.

### Wrong Foreign Key Values

**Problem:** homeworld\_id doesn't match actual planet IDs.

**Solution:** Use subqueries to find correct IDs:

```
SELECT id FROM planets WHERE name = 'Tatooine';
```

### Many-to-Many Confusion

#### Remember:

- One-to-many: Use foreign key in the "many" table
- Many-to-many: Use junction table with two foreign keys

## Checkpoint: What You've Learnt

---

Before moving on, make sure you can:

-  Explain database normalization
-  Understand one-to-many relationships
-  Create tables with foreign keys
-  Use junction tables for many-to-many relationships
-  Insert data maintaining referential integrity
-  Link records across tables using IDs

## Challenge Problem (Optional)

---

**Task:** Design and create a `missions` table that tracks Star Wars missions. Each mission should have:

- A unique ID
- A name
- A location (foreign key to planets)
- A date
- A description

Then create a `character_missions` junction table to track which characters participated in which missions. Insert at least 3 missions and link characters to them.

## Save Your Work with Git

---

```
git status  
git add lessons/lesson5_schema.sql lessons/lesson5_data.sql database/starwars.db  
git commit -m "Completed Lesson 5: Created related tables with foreign keys"  
git push
```



## Key Concepts Learnt

Concept	Meaning
<b>Normalization</b>	Organizing data to reduce duplication
<b>Primary Key</b>	Unique identifier for rows in a table
<b>Foreign Key</b>	Field linking to another table's primary key
<b>One-to-Many</b>	One record relates to many records
<b>Many-to-Many</b>	Multiple records relate to multiple records
<b>Junction Table</b>	Links two tables in many-to-many relationship
<b>Referential Integrity</b>	Ensuring foreign keys reference valid records



## Great Progress!

You've now built a proper relational database! In the next lesson, you'll learn how to retrieve data from multiple tables using JOINs.

**Ready to continue?** Move on to `lesson6_instructions.md`

### Need Help?

- Draw diagrams of table relationships
- Verify IDs before creating foreign key links
- Check constraints when inserts fail
- Ask your instructor
- Compare with the solution files!