Topic 04: Keys and Joins - Relational Database Design

Overview

This topic introduces **relational database design** - the fundamental principle that makes databases powerful for modeling real-world relationships. You'll learn to separate data into multiple related tables using **primary keys** and **foreign keys**, enforce **referential integrity**, and combine data using **SQL joins**. This represents a crucial shift from storing redundant data to organizing data efficiently with relationships.

The Problem with Previous Designs

In Topics 1-3, the pets table stored redundant data:

```
id | name    | type | age | owner
---|---------|------|-----|------
1  | Dorothy | dog  | 9   | greg
2  | Suzy    | dog  | 9   | greg
3  | Sandy   | cat  | 2   | steve
```

Problems:

- Pet type ("dog", "cat") repeated for every pet
- No place to store type-specific information (food, sound)
- Typos create inconsistency ("dog" vs "Dog" vs "dogs")
- Changing type information requires updating many rows
- Can't enforce that valid types exist

The Solution: Normalized Design

Split into two tables with a **relationship**:

kind table (lookup/reference table):

```
id | name | food     | sound
---|------|----------|------
1  | dog  | dogfood  | bark
2  | cat  | catfood  | meow
```

pet table (main data table):

```
id | name    | kind_id | age | owner
---|---------|---------|-----|------
1  | Dorothy | 1       | 9   | greg
2  | Suzy    | 1       | 9   | greg
3  | Sandy   | 2       | 2   | steve
```

Benefits:

- Each kind defined once (single source of truth)
- Kind-specific data (food, sound) stored with the kind
- Consistency enforced through foreign key
- Easy to add new kinds without touching pet records
- Can enforce that pets must reference valid kinds

Primary Keys

A **primary key** uniquely identifies each row in a table.

```
create table kind (
    id integer primary key autoincrement,
    name text not null,
    food text,
    sound text
);
```

Key properties:

- **Unique**: No two rows have the same primary key value
- **Not null**: Every row must have a primary key
- **Autoincrement**: Database automatically generates sequential IDs (1, 2, 3...)
- **Immutable**: Primary keys shouldn't change once assigned

Why use integer IDs instead of names?

[!] **Using name as primary key:**

```
create table kind (
    name text primary key,  -- Fragile!
    food text,
    sound text
);
```

Problems:

- What if you want to change "dog" to "Dog"? Must update all pets
- Names might be long or contain spaces
- What if you need two entries with same name? (e.g., "mixed breed")

☑ **Using integer ID:**

- IDs never change
- Compact storage

- Fast comparisons
- Relationships remain stable

Foreign Keys

A **foreign key** creates a relationship by referencing another table's primary key.

```
create table pet (
    id integer primary key autoincrement,
    name text not null,
    kind_id integer not null,
    age integer,
    owner text,
    foreign key (kind_id) references kind(id)
      on delete RESTRICT
      on update CASCADE
);
```

Breaking it down:

- `kind_id integer not null` - Column that stores the reference
- `foreign key (kind_id)` - Declares this column is a foreign key
- `references kind(id)` - Points to the `id` column in the `kind` table
- `on delete RESTRICT` - Cannot delete a kind if pets reference it
- `on update CASCADE` - If kind ID changes, update all pets automatically

Referential Integrity

Referential integrity ensures relationships remain valid:

Without foreign keys:

```
-- Nothing stops this invalid data:
insert into pet(name, kind_id, age, owner) values ('Spot', 999, 5,
'alice');
-- kind_id 999 doesn't exist!
```

With foreign keys:

```
-- This will fail with an error:
insert into pet(name, kind_id, age, owner) values ('Spot', 999, 5,
'alice');
-- Error: FOREIGN KEY constraint failed
```

Enabling Foreign Keys in SQLite

SQLite requires explicit activation of foreign key constraints:

```
def initialize(database_file):
    global connection
    connection = sqlite3.connect(database_file, check_same_thread=False)
    connection.execute("PRAGMA foreign_keys = 1")  # Critical!
    connection.row_factory = sqlite3.Row
```

Without this pragma:

- Foreign key constraints are defined but **not enforced**
- Invalid references can be inserted
- Referential integrity is not guaranteed

With this pragma:

- Database enforces foreign key constraints
- Invalid inserts are rejected
- Delete restrictions are enforced

Cascade Options

Foreign keys support different behaviors when referenced data changes:

ON DELETE RESTRICT

```
on delete RESTRICT
```

Behavior: Cannot delete a kind if any pets reference it.

Example:

```
-- Try to delete kind with id=1 (dog)
delete from kind where id = 1;
-- Error: FOREIGN KEY constraint failed
-- (Because pets with kind_id=1 exist)
```

Use case: Prevents accidental data loss. Must explicitly handle dependent records first.

ON DELETE CASCADE

```
on delete CASCADE
```

Behavior: Deleting a kind automatically deletes all pets of that kind.

Example:

```
-- Delete kind with id=1 (dog)
delete from kind where id = 1;
-- Success: kind deleted AND all pets with kind_id=1 deleted
```

Use case: When child records have no meaning without parent (e.g., order items without an order).

[!] **Dangerous**: Easy to accidentally delete large amounts of data.

ON UPDATE CASCADE

```
on update CASCADE
```

Behavior: If a kind's ID changes, all pets' kind_id values update automatically.

Example:

```
-- Change kind id from 1 to 100
update kind set id = 100 where id = 1;
-- All pets with kind_id=1 now have kind_id=100
```

Use case: Maintains consistency when primary keys change (rare with autoincrement).

ON DELETE SET NULL

```
on delete SET NULL
```

Behavior: Deleting a kind sets pets' kind_id to NULL.

Use case: When relationship is optional and you want to preserve the child records.

SQL Joins

Joins combine data from multiple tables based on relationships.

The Problem

With separate tables, querying is tedious:

```
# Get pet information
cursor.execute("select * from pet where id = 1")
pet = cursor.fetchone()  # (1, 'Dorothy', 1, 9, 'greg')
```

```
# Look up kind information separately
cursor.execute("select * from kind where id = ?", (pet['kind_id'],))
kind = cursor.fetchone()  # (1, 'dog', 'dogfood', 'bark')

# Manually combine
result = {
    'name': pet['name'],
    'kind_name': kind['name'],
    'food': kind['food'],
    'sound': kind['sound']
}
```

The Solution: JOIN

```
SELECT pet.id, pet.name, pet.age, pet.owner,
       kind.name as kind_name, kind.food, kind.sound
FROM pet
JOIN kind ON pet.kind_id = kind.id
```

Benefits:

- Single query retrieves all data
- Database optimizes the operation
- Less code, fewer database round-trips
- Results include combined information

Implementation in database.py

```
def get_pets():
    cursor = connection.cursor()
    cursor.execute("""
        SELECT pet.id, pet.name, pet.age, pet.owner,
               kind.name as kind_name, kind.food, kind.sound
        FROM pet
        JOIN kind ON pet.kind_id = kind.id
    """)
    pets = cursor.fetchall()
    pets = [dict(pet) for pet in pets]
    return pets
```

Result:

```
[
    {
        'id': 1,
        'name': 'Dorothy',
```

```
        'age': 9,
        'owner': 'greg',
        'kind_name': 'dog',
        'food': 'dogfood',
        'sound': 'bark'
    },
    # ... more pets
]
```

JOIN Syntax Breakdown

```
SELECT pet.id, pet.name, kind.name as kind_name
FROM pet
JOIN kind ON pet.kind_id = kind.id
```

- `SELECT` - Specify columns to retrieve
- `pet.id` - Qualify column with table name (avoids ambiguity)
- `kind.name as kind_name` - Rename column in results
- `FROM pet` - Main table (left side of join)
- `JOIN kind` - Table to join with (right side)
- `ON pet.kind_id = kind.id` - Join condition (how tables relate)

Types of Joins (Overview)

INNER JOIN (default JOIN):

- Returns only rows with matches in both tables
- Used in this topic

LEFT JOIN:

- Returns all rows from left table, even without matches
- Unmatched fields are NULL

RIGHT JOIN:

- Returns all rows from right table
- Less common (not standard in SQLite)

CROSS JOIN:

- Every row from first table paired with every row from second
- Rarely used (Cartesian product)

Two-Table CRUD Operations

The application now manages two related entities:

Managing Kinds (Reference Data)

List all kinds:

```
def get_kinds():
    cursor = connection.cursor()
    cursor.execute("""select * from kind""")
    kinds = cursor.fetchall()
    kinds = [dict(kind) for kind in kinds]
    return kinds
```

Create a kind:

```
def create_kind(data):
    cursor = connection.cursor()
    cursor.execute(
        """insert into kind(name, food, sound) values (?,?,?)""",
        (data["name"], data["food"], data["sound"])
    )
    connection.commit()
```

Update a kind:

```
def update_kind(id, data):
    cursor = connection.cursor()
    cursor.execute(
        """update kind set name=?, food=?, sound=? where id=?""",
        (data["name"], data["food"], data["sound"], id)
    )
    connection.commit()
```

Delete a kind (with error handling):

```
def delete_kind(id):
    cursor = connection.cursor()
    cursor.execute("""delete from kind where id = ?""", (id,))
    connection.commit()
    # Will raise exception if pets reference this kind (RESTRICT)
```

Managing Pets (Dependent Data)

Create pet with kind reference:

```
def create_pet(data):
    try:
        data["age"] = int(data["age"])
```

```
    except:
        data["age"] = 0
    cursor = connection.cursor()
    cursor.execute(
        """insert into pet(name, age, kind_id, owner) values (?,?,?,?)""",
        (data["name"], data["age"], data["kind_id"], data["owner"])
    )
    connection.commit()
```

Note: `kind_id` comes from the form's dropdown selection.

Web Interface Changes

Kind Selection Dropdown

Before (Topic 03): Text input for type

```
<p>Type:<input name="type"/></p>
```

After (Topic 04): Dropdown with valid kinds

```
<p>Kind_ID:<select name="kind_id">
    {% for kind in kinds %}
    <option value="{{kind['id']}}">{{kind['name']}}</option>
    {% endfor %}
</select></p>
```

Benefits:

- Users can only select valid kinds
- No typos or inconsistency
- Shows available options
- Automatically uses correct kind_id

Displaying Combined Data

Template (list.html):

```
<tr>
    <td>{{ pet['name'] }}</td>
    <td>{{ pet['kind_name'] }}</td>
    <td>{{ pet['food'] }}</td>
    <td>{{ pet['sound'] }}</td>
    <td>{{ pet['age'] }}</td>
    <td>{{ pet['owner'] }}</td>
</tr>
```

Template doesn't need to know about the join - it receives complete data.

Kind Management Routes

New routes for managing the kinds table:

```
@app.route("/kind/list", methods=["GET"])
def get_kind_list():
    kinds = database.get_kinds()
    return render_template("kind_list.html", kinds=kinds)

@app.route("/kind/create", methods=["GET", "POST"])
def get_kind_create():
    # ... handle form display and submission

@app.route("/kind/update/<id>", methods=["GET", "POST"])
def get_kind_update(id):
    # ... handle editing kinds

@app.route("/kind/delete/<id>", methods=["GET"])
def get_kind_delete(id):
    try:
        database.delete_kind(id)
    except Exception as e:
        return render_template("error.html", error_text=str(e))
    return redirect(url_for("get_kind_list"))
```

Key feature: Delete error handling catches foreign key violations.

Error Handling for Referential Integrity

```
@app.route("/kind/delete/<id>", methods=["GET"])
def get_kind_delete(id):
    try:
        database.delete_kind(id)
    except Exception as e:
        return render_template("error.html", error_text=str(e))
    return redirect(url_for("get_kind_list"))
```

User experience:

1. User tries to delete "dog" kind
2. Database raises foreign key constraint error (pets still reference it)
3. Application catches exception
4. Displays friendly error page: "FOREIGN KEY constraint failed"
5. User must delete or reassign pets before deleting the kind

Error template (error.html):

```
<html>
    <H1>Error occured:</H1>
    <H3>{{error_text}}</H3>
    <a href="/kind">Pet Kinds</a>
</html>
```

Database Setup Script

The SQL setup script demonstrates proper table creation order:

```
pragma foreign_keys = 1;  -- Enable enforcement

-- Create parent table first
create table kind (
    id integer primary key autoincrement,
    name text not null,
    food text,
    sound text
);

-- Insert reference data
insert into kind(name, food, sound) values ('dog','dogfood','bark');
insert into kind(name, food, sound) values ('cat','catfood','meow');

-- Create child table second (references parent)
create table pet (
    id integer primary key autoincrement,
    name text not null,
    kind_id integer not null,
    age integer,
    owner text,
    foreign key (kind_id) references kind(id)
      on delete RESTRICT
      on update CASCADE
);

-- Insert child data (with valid foreign keys)
insert into pet(name, kind_id, age, owner) values ('dorothy', 1, 9,
'greg');
```

Critical order:

1. Enable foreign keys
2. Create parent table (kind)
3. Populate parent table
4. Create child table (pet) with foreign key
5. Populate child table with valid references

Testing with Related Tables

```
def setup_test_database():
    initialize("test_pets.db")
    cursor = connection.cursor()

    # Create kind table first
    cursor.execute("""
        create table if not exists kind (
            id integer primary key autoincrement,
            name text not null,
            food text,
            sound text
        )
    """)
    connection.commit()

    # Insert kinds
    cursor.execute("insert into kind(name, food, sound) values (?,?,?)",
                   ("dog", "dogfood", "bark"))
    cursor.execute("insert into kind(name, food, sound) values (?,?,?)",
                   ("cat", "catfood", "meow"))
    connection.commit()

    # Create pet table (references kind)
    cursor.execute("""
        create table if not exists pet (
            id integer primary key autoincrement,
            name text not null,
            kind_id integer,
            age integer,
            owner text
        )
    """)
    connection.commit()

    # Insert pets with valid kind_id
    pets = [
        {"name": "dorothy", "kind_id": 1, "age": 9, "owner": "greg"},
        {"name": "suzy", "kind_id": 1, "age": 9, "owner": "greg"},
    ]
    for pet in pets:
        create_pet(pet)
```

Testing joins:

```
def test_get_pets():
    pets = get_pets()
    # Verify joined data is present
    assert 'kind_name' in pets[0]
```

```
    assert 'food' in pets[0]
    assert 'sound' in pets[0]
```

Database Normalization Principles

This design follows **Third Normal Form (3NF)**:

First Normal Form (1NF)

- Each column contains atomic (indivisible) values
- Each row is unique
- No repeating groups

☑ Achieved: Each cell has single value, rows identified by ID.

Second Normal Form (2NF)

- In 1NF
- All non-key columns fully depend on the primary key

☑ Achieved: All pet attributes depend on pet.id, all kind attributes depend on kind.id.

Third Normal Form (3NF)

- In 2NF
- No transitive dependencies (non-key columns don't depend on other non-key columns)

☑ Achieved: Kind data (food, sound) moved to kind table, not stored with pets.

Running the Application

1. Initialize Database

```
# Using SQL script
sqlite3 pets.db < setup_database.sql

# Or using the notebook
# Open db-api.ipynb and run cells
```

2. Run Application

```
flask --app app run --debug
```

3. Access Features

- http://localhost:5000/list - View pets with kind information
- http://localhost:5000/kind/list - Manage pet kinds

- http://localhost:5000/create - Add new pet (with kind dropdown)
- http://localhost:5000/kind/create - Add new kind

4. Test Foreign Key Constraints

Try to delete a kind that has pets:

1. Go to kind list
2. Click delete on "dog"
3. See error message (can't delete - pets reference it)

Common Patterns and Best Practices

1. Always Enable Foreign Keys

```
connection.execute("PRAGMA foreign_keys = 1")
```

2. Create Tables in Dependency Order

- Parent tables first
- Child tables second

3. Populate Tables in Dependency Order

- Reference data first
- Dependent data second

4. Use Meaningful Names

- `kind_id` clearly indicates it's a foreign key to kind table
- Better than `type_id` or just `type`

5. Handle Constraint Violations

```
try:
    database.delete_kind(id)
except Exception as e:
    # Show user-friendly error
    return render_template("error.html", error_text=str(e))
```

6. Use JOIN for Display

- Don't make separate queries
- Let database optimize the join

7. Provide Dropdowns for Foreign Keys

- Users select from valid options
- Prevents invalid references
- Better UX than typing IDs

Key Concepts and Terminology

- **Primary Key**: Unique identifier for each row in a table
- **Foreign Key**: Column that references another table's primary key
- **Referential Integrity**: Ensuring foreign keys always reference valid rows
- **Normalization**: Organizing data to reduce redundancy
- **JOIN**: Combining rows from multiple tables based on relationships
- **CASCADE**: Automatic propagation of changes
- **RESTRICT**: Prevent operations that would violate integrity
- **Parent Table**: Table being referenced (kind)
- **Child Table**: Table with foreign key (pet)
- **Lookup Table**: Small table of reference values (kind)

When to Use Related Tables

Use separate tables when:

- Data would be repeated across many rows
- The repeated data has its own attributes
- You need to enforce consistency
- You want to manage reference data separately

Example scenarios:

- Products and Categories
- Orders and Customers
- Books and Authors
- Students and Courses
- Employees and Departments

Don't over-normalize:

- Not every column needs its own table
- Simple attributes (name, age) stay in main table
- Only separate data that's truly independent

Advantages Over Previous Approach

Aspect	Topic 03	Topic 04
Data redundancy	High (type repeated)	Low (kind defined once)
Consistency	Manual (typos possible)	Enforced (foreign keys)
Extensibility	Hard (add type attrs everywhere)	Easy (add columns to kind)
Validation	Application-level only	Database-level constraints

Aspect	Topic 03	Topic 04
Queries	Simple but incomplete	Joins provide full data
Maintenance	Update many rows	Update reference table

Key Takeaways

1. **Primary keys uniquely identify rows** - use autoincrement integers
2. **Foreign keys create relationships** - reference other tables' primary keys
3. **Referential integrity prevents orphaned data** - must be explicitly enabled in SQLite
4. **Cascade options control behavior** - RESTRICT, CASCADE, SET NULL
5. **Joins combine related data** - single query for complete information
6. **Normalization reduces redundancy** - separate independent entities
7. **Create parent tables first** - dependencies matter for foreign keys
8. **Handle constraint violations gracefully** - show users meaningful errors
9. **Dropdowns prevent invalid references** - better UX and data quality
10. **This is fundamental relational design** - foundation for complex data models

Topic 04 establishes the relational database patterns that power most production applications. Understanding keys, joins, and referential integrity is essential for designing scalable, maintainable database schemas.