Chapter 7 Table Design Best Practices (PostgreSQL 17)

Table of Contents

Introduction

Good database design is like having a well-organized closet - when you need something, you know exactly where to find it. Just as obsessively organized people appreciate knowing their keys will always be on the same hook, database analysts value a carefully structured database when they need to retrieve information from millions of rows across dozens of tables.

This chapter builds on basic table creation concepts by diving deeper into organizing and tuning PostgreSQL 17 databases for optimal performance and data integrity.

Naming Conventions

PostgreSQL developers follow different naming patterns for database objects (identifiers):

- **snake_case**: `berry_smoothie` (all lowercase with underscores separating words)
- **PascalCase**: `BerrySmoothie` (all words capitalized including the first)

While preferences may vary, consistent naming is crucial:

```
-- snake_case example
CREATE TABLE inventory.product_inventory (
    product_id bigint GENERATED ALWAYS AS IDENTITY,
    product_name varchar(100),
    unit_price numeric(10,2)
);

-- PascalCase example (requires quotes)
```

```
CREATE TABLE "Inventory"."ProductInventory" (
    "ProductId" bigint GENERATED ALWAYS AS IDENTITY,
    "ProductName" varchar(100),
    "UnitPrice" numeric(10,2)
);
```

Naming Best Practices:

1. **Choose a consistent style**: Either snake_case or PascalCase (with quotes)
2. **Choose meaningful names**: Avoid cryptic abbreviations - `arrival_time` is better than `arr_tm` (snake_case) or `ArrivalTime` is better than `ArrTm` (PascalCase)
3. **Use singular nouns for tables**: Since tables represent a definition for each instance, use singular names like `teacher`, `product`, or `department`
4. **Include schema qualifiers**: Use schema names to organize related tables (`inventory.product`, `accounting.invoice`)
5. **Be mindful of name length limits**: PostgreSQL limits identifiers to 63 characters (ANSI SQL standard is 128)
6. **Date-tag temporary tables**: For copies or backups, append date formats like `tire_size_2023_10_20` for easy sorting

Constraints for Data Integrity

Constraints are rules that help maintain data quality and relationships between tables. PostgreSQL supports several constraint types:

Primary Keys

Primary keys uniquely identify each row in a table and guarantee:

1. Uniqueness of values across rows
2. No NULL values are allowed

Natural vs. Surrogate Keys

Natural Keys: Use existing meaningful data as identifiers

```
-- snake_case
CREATE TABLE customer.license (
    license_id varchar(10) CONSTRAINT license_pky PRIMARY KEY,
    first_name varchar(50),
    last_name varchar(50)
);

-- PascalCase
CREATE TABLE "Customer"."License" (
```

```
    "LicenseId" varchar(10) CONSTRAINT "License_Pky" PRIMARY KEY,
    "FirstName" varchar(50),
    "LastName" varchar(50)
);
```

Surrogate Keys: Create artificial identifiers (recommended in PostgreSQL 17)

```
-- snake_case
CREATE TABLE sales.order (
    order_id bigint GENERATED ALWAYS AS IDENTITY,
    order_date date,
    customer_id bigint,
    CONSTRAINT order_pky PRIMARY KEY (order_id)
);

-- PascalCase
CREATE TABLE "Sales"."Order" (
    "OrderId" bigint GENERATED ALWAYS AS IDENTITY,
    "OrderDate" date,
    "CustomerId" bigint,
    CONSTRAINT "Order_Pky" PRIMARY KEY ("OrderId")
);
```

> 💡 **PostgreSQL 17 Tip**: Always use `GENERATED ALWAYS AS IDENTITY` instead of older `serial` types. This provides better standards compliance and integrity protection.

Composite Primary Keys: When uniqueness requires multiple columns

```
-- snake_case
CREATE TABLE school.attendance (
    student_id varchar(10),
    school_day date,
    present boolean,
    CONSTRAINT attendance_pky PRIMARY KEY (student_id, school_day)
);

-- PascalCase
CREATE TABLE "School"."Attendance" (
    "StudentId" varchar(10),
    "SchoolDay" date,
    "Present" boolean,
    CONSTRAINT "Attendance_Pky" PRIMARY KEY ("StudentId", "SchoolDay")
);
```

Foreign Keys

Foreign keys preserve referential integrity by ensuring that values in one table exist in another table's primary key (or unique constraint).

```
-- snake_case
CREATE TABLE vehicle.registration (
    registration_id varchar(10),
    registration_date date,
    license_id varchar(10) REFERENCES customer.license (license_id),
    CONSTRAINT registration_pky PRIMARY KEY (registration_id, license_id)
);

-- PascalCase
CREATE TABLE "Vehicle"."Registration" (
    "RegistrationId" varchar(10),
    "RegistrationDate" date,
    "LicenseId" varchar(10) REFERENCES "Customer"."License" ("LicenseId"),
    CONSTRAINT "Registration_Pky" PRIMARY KEY ("RegistrationId", "LicenseId")
);
```

CASCADE Deletion

To automatically delete related records when a parent record is deleted:

```
-- snake_case
CREATE TABLE vehicle.registration (
    registration_id varchar(10),
    registration_date date,
    license_id varchar(10) REFERENCES customer.license (license_id) ON DELETE
CASCADE,
    CONSTRAINT registration_pky PRIMARY KEY (registration_id, license_id)
);

-- PascalCase
CREATE TABLE "Vehicle"."Registration" (
    "RegistrationId" varchar(10),
    "RegistrationDate" date,
    "LicenseId" varchar(10) REFERENCES "Customer"."License" ("LicenseId") ON
DELETE CASCADE,
    CONSTRAINT "Registration_Pky" PRIMARY KEY ("RegistrationId", "LicenseId")
);
```

CHECK Constraints

A CHECK constraint evaluates whether data meets specified criteria:

```
-- snake_case
CREATE TABLE hr.employee (
    employee_id bigint GENERATED ALWAYS AS IDENTITY,
    role varchar(50),
    salary integer,
    CONSTRAINT employee_pky PRIMARY KEY (employee_id),
```

```
    CONSTRAINT valid_role CHECK (role IN('Admin', 'Manager', 'Staff')),
    CONSTRAINT positive_salary CHECK (salary > 0)
);

-- PascalCase
CREATE TABLE "Hr"."Employee" (
    "EmployeeId" bigint GENERATED ALWAYS AS IDENTITY,
    "Role" varchar(50),
    "Salary" integer,
    CONSTRAINT "Employee_Pky" PRIMARY KEY ("EmployeeId"),
    CONSTRAINT "ValidRole" CHECK ("Role" IN('Admin', 'Manager', 'Staff')),
    CONSTRAINT "PositiveSalary" CHECK ("Salary" > 0)
);
```

You can also perform cross-column checks:

```
-- snake_case
CREATE TABLE sales.pricing (
    product_id bigint GENERATED ALWAYS AS IDENTITY,
    retail_price numeric(10,2),
    sale_price numeric(10,2),
    CONSTRAINT price_pky PRIMARY KEY (product_id),
    CONSTRAINT valid_sale CHECK (sale_price < retail_price)
);

-- PascalCase
CREATE TABLE "Sales"."Pricing" (
    "ProductId" bigint GENERATED ALWAYS AS IDENTITY,
    "RetailPrice" numeric(10,2),
    "SalePrice" numeric(10,2),
    CONSTRAINT "Price_Pky" PRIMARY KEY ("ProductId"),
    CONSTRAINT "ValidSale" CHECK ("SalePrice" < "RetailPrice")
);
```

UNIQUE Constraints

UNIQUE ensures column values are unique across rows, but unlike primary keys, allows NULL values:

```
-- snake_case
CREATE TABLE contacts.user_profile (
    user_id bigint GENERATED ALWAYS AS IDENTITY CONSTRAINT user_pky PRIMARY KEY,
    first_name varchar(50),
    last_name varchar(50),
    email varchar(200) CONSTRAINT email_unique UNIQUE
);

-- PascalCase
CREATE TABLE "Contacts"."UserProfile" (
    "UserId" bigint GENERATED ALWAYS AS IDENTITY CONSTRAINT "User_Pky" PRIMARY
```

```
KEY,
    "FirstName" varchar(50),
    "LastName" varchar(50),
    "Email" varchar(200) CONSTRAINT "EmailUnique" UNIQUE
);
```

NOT NULL Constraints

NOT NULL prevents columns from accepting NULL values:

```
-- snake_case
CREATE TABLE school.student (
    student_id bigint GENERATED ALWAYS AS IDENTITY,
    first_name varchar(50) NOT NULL,
    last_name varchar(50) NOT NULL,
    CONSTRAINT student_pky PRIMARY KEY (student_id)
);

-- PascalCase
CREATE TABLE "School"."Student" (
    "StudentId" bigint GENERATED ALWAYS AS IDENTITY,
    "FirstName" varchar(50) NOT NULL,
    "LastName" varchar(50) NOT NULL,
    CONSTRAINT "Student_Pky" PRIMARY KEY ("StudentId")
);
```

Adding or Removing Constraints

To modify constraints on existing tables:

```
-- snake_case
-- Remove a constraint
ALTER TABLE school.student DROP CONSTRAINT student_pky;

-- Add a constraint
ALTER TABLE school.student ADD CONSTRAINT student_pky PRIMARY KEY (student_id);

-- Remove NOT NULL
ALTER TABLE school.student ALTER COLUMN first_name DROP NOT NULL;

-- Add NOT NULL
ALTER TABLE school.student ALTER COLUMN first_name SET NOT NULL;

-- PascalCase
-- Remove a constraint
ALTER TABLE "School"."Student" DROP CONSTRAINT "Student_Pky";

-- Add a constraint
ALTER TABLE "School"."Student" ADD CONSTRAINT "Student_Pky" PRIMARY KEY
```

```
("StudentId");

-- Remove NOT NULL
ALTER TABLE "School"."Student" ALTER COLUMN "FirstName" DROP NOT NULL;

-- Add NOT NULL
ALTER TABLE "School"."Student" ALTER COLUMN "FirstName" SET NOT NULL;
```

📊 Understanding Indexes

B-Tree Indexes

B-Tree (Balanced Tree) is PostgreSQL's default index type. It organizes data hierarchically for efficient searches using equality and range operators (`<`, `<=`, `=`, `>=`, `>`).

PostgreSQL automatically creates indexes for primary keys and unique constraints. For other columns, create indexes with:

```
-- snake_case
CREATE INDEX street_idx ON location.address (street);

-- PascalCase
CREATE INDEX "StreetIdx" ON "Location"."Address" ("Street");
```

Index Performance Testing

Use `EXPLAIN ANALYZE` to measure query performance before and after adding indexes:

```
-- Before index (snake_case)
EXPLAIN ANALYZE SELECT * FROM location.address WHERE street = 'BROADWAY';

-- Create index
CREATE INDEX street_idx ON location.address (street);

-- After index
EXPLAIN ANALYZE SELECT * FROM location.address WHERE street = 'BROADWAY';

-- Before index (PascalCase)
EXPLAIN ANALYZE SELECT * FROM "Location"."Address" WHERE "Street" = 'BROADWAY';

-- Create index
CREATE INDEX "StreetIdx" ON "Location"."Address" ("Street");

-- After index
EXPLAIN ANALYZE SELECT * FROM "Location"."Address" WHERE "Street" = 'BROADWAY';
```

A typical performance improvement might drop query execution time from 300ms to 5ms.

When to Use Indexes

Indexes improve query speed but require maintenance and storage. Consider these guidelines:

1. Add indexes to columns frequently used in `WHERE` clauses
2. Index foreign key columns to speed up joins
3. Consider specialized index types for full-text search (GIN, GiST)
4. Use `EXPLAIN ANALYZE` to test performance improvements
5. Remember that each insert/update requires index maintenance

PostgreSQL 17 Enhancements

PostgreSQL 17 brings several improvements to table design and constraints:

1. **Generated Columns**: Define columns whose values are calculated from other columns

```
-- snake_case
CREATE TABLE sales.order_item (
    item_id bigint GENERATED ALWAYS AS IDENTITY,
    quantity integer,
    unit_price numeric(10,2),
    total_price numeric(10,2) GENERATED ALWAYS AS (quantity * unit_price)
STORED,
    CONSTRAINT item_pky PRIMARY KEY (item_id)
);

-- PascalCase
CREATE TABLE "Sales"."OrderItem" (
    "ItemId" bigint GENERATED ALWAYS AS IDENTITY,
    "Quantity" integer,
    "UnitPrice" numeric(10,2),
    "TotalPrice" numeric(10,2) GENERATED ALWAYS AS ("Quantity" *
"UnitPrice") STORED,
    CONSTRAINT "Item_Pky" PRIMARY KEY ("ItemId")
);
```

2. **Identity Columns**: Always use `GENERATED ALWAYS AS IDENTITY` or `GENERATED BY DEFAULT AS IDENTITY` instead of `serial`

```
-- Recommended in PostgreSQL 17 (snake_case)
CREATE TABLE customer.user (
    user_id bigint GENERATED ALWAYS AS IDENTITY,
    username varchar(50)
);

-- Recommended in PostgreSQL 17 (PascalCase)
CREATE TABLE "Customer"."User" (
    "UserId" bigint GENERATED ALWAYS AS IDENTITY,
    "Username" varchar(50)
);
```

```
-- Avoid in new code (snake_case)
CREATE TABLE customer.user_old (
    user_id bigserial,
    username varchar(50)
);

-- Avoid in new code (PascalCase)
CREATE TABLE "Customer"."UserOld" (
    "UserId" bigserial,
    "Username" varchar(50)
);
```

3. **Enhanced CHECK Constraints**: More expressive validation capabilities

```
-- snake_case
CREATE TABLE hr.employee (
    id bigint GENERATED ALWAYS AS IDENTITY,
    hire_date date,
    salary numeric(10,2),
    CONSTRAINT valid_date CHECK (hire_date <= CURRENT_DATE),
    CONSTRAINT salary_range CHECK (salary BETWEEN 30000 AND 500000)
);

-- PascalCase
CREATE TABLE "Hr"."Employee" (
    "Id" bigint GENERATED ALWAYS AS IDENTITY,
    "HireDate" date,
    "Salary" numeric(10,2),
    CONSTRAINT "ValidDate" CHECK ("HireDate" <= CURRENT_DATE),
    CONSTRAINT "SalaryRange" CHECK ("Salary" BETWEEN 30000 AND 500000)
);
```

4. **Schema Organization**: More explicit schema usage for better organization

```
-- Create schemas for different areas of application (snake_case)
CREATE SCHEMA inventory;
CREATE SCHEMA sales;
CREATE SCHEMA customer;

-- Create tables in appropriate schemas
CREATE TABLE inventory.product (...);
CREATE TABLE sales.order (...);
CREATE TABLE customer.profile (...);

-- Create schemas for different areas of application (PascalCase)
CREATE SCHEMA "Inventory";
CREATE SCHEMA "Sales";
CREATE SCHEMA "Customer";
```

```
-- Create tables in appropriate schemas
CREATE TABLE "Inventory"."Product" (...);
CREATE TABLE "Sales"."Order" (...);
CREATE TABLE "Customer"."Profile" (...);
```

Best Practices Summary

1. **Naming Conventions**
   - Use snake_case or PascalCase consistently (with quotes for PascalCase)
   - Include schema qualifiers
   - Make names meaningful and clear
   - Use singular nouns for table names
2. **Surrogate Keys**
   - Use `GENERATED ALWAYS AS IDENTITY` for new tables
   - Consider UUID for distributed systems
3. **Constraints**
   - Apply PRIMARY KEY constraints on every table
   - Use FOREIGN KEY constraints to maintain relationships
   - Add CHECK constraints to validate data
   - Mark required columns as NOT NULL
4. **Indexing**
   - Index columns used in WHERE clauses and joins
   - Test performance with EXPLAIN ANALYZE
   - Don't over-index (balance query speed with insert/update performance)
5. **Schema Organization**
   - Group related tables under schemas
   - Use fully qualified names (schema.table)

Hands-On Exercises

Exercise 1: Improve Album Collection Schema

Starting with these basic tables:

```
CREATE TABLE album (
    album_id bigserial,
    album_catalog_code varchar(100),
    album_title text,
    album_artist text,
    album_release_date date,
```

```
    album_genre varchar(40),
    album_description text
);

CREATE TABLE song (
    song_id bigserial,
    song_title text,
    song_artist text,
    album_id bigint
);
```

Enhance them with best practices:

```
-- snake_case example
-- Create schema
CREATE SCHEMA music;

-- Improved album table
CREATE TABLE music.album (
    album_id bigint GENERATED ALWAYS AS IDENTITY,
    album_catalog_code varchar(100) UNIQUE NOT NULL,
    album_title text NOT NULL,
    album_artist text NOT NULL,
    album_release_date date NOT NULL,
    album_genre varchar(40),
    album_description text,
    created_at timestamptz DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT album_pky PRIMARY KEY (album_id)
);

-- Improved song table
CREATE TABLE music.song (
    song_id bigint GENERATED ALWAYS AS IDENTITY,
    song_title text NOT NULL,
    song_artist text NOT NULL,
    album_id bigint NOT NULL,
    track_number integer,
    duration interval,
    CONSTRAINT song_pky PRIMARY KEY (song_id),
    CONSTRAINT album_fky FOREIGN KEY (album_id)
        REFERENCES music.album (album_id) ON DELETE CASCADE,
    CONSTRAINT valid_track CHECK (track_number > 0)
);

-- Create indexes for common queries
CREATE INDEX song_album_idx ON music.song (album_id);
CREATE INDEX album_artist_idx ON music.album (album_artist);
CREATE INDEX album_genre_idx ON music.album (album_genre);

-- PascalCase example
-- Create schema
CREATE SCHEMA "Music";
```

```
-- Improved album table
CREATE TABLE "Music"."Album" (
    "AlbumId" bigint GENERATED ALWAYS AS IDENTITY,
    "AlbumCatalogCode" varchar(100) UNIQUE NOT NULL,
    "AlbumTitle" text NOT NULL,
    "AlbumArtist" text NOT NULL,
    "AlbumReleaseDate" date NOT NULL,
    "AlbumGenre" varchar(40),
    "AlbumDescription" text,
    "CreatedAt" timestamptz DEFAULT CURRENT_TIMESTAMP,
    CONSTRAINT "Album_Pky" PRIMARY KEY ("AlbumId")
);

-- Improved song table
CREATE TABLE "Music"."Song" (
    "SongId" bigint GENERATED ALWAYS AS IDENTITY,
    "SongTitle" text NOT NULL,
    "SongArtist" text NOT NULL,
    "AlbumId" bigint NOT NULL,
    "TrackNumber" integer,
    "Duration" interval,
    CONSTRAINT "Song_Pky" PRIMARY KEY ("SongId"),
    CONSTRAINT "Album_Fky" FOREIGN KEY ("AlbumId")
        REFERENCES "Music"."Album" ("AlbumId") ON DELETE CASCADE,
    CONSTRAINT "ValidTrack" CHECK ("TrackNumber" > 0)
);

-- Create indexes for common queries
CREATE INDEX "SongAlbumIdx" ON "Music"."Song" ("AlbumId");
CREATE INDEX "AlbumArtistIdx" ON "Music"."Album" ("AlbumArtist");
CREATE INDEX "AlbumGenreIdx" ON "Music"."Album" ("AlbumGenre");
```

Back to TOC