Constraints and Schema Design: Let the Database Do the Work

The Philosophy: Constraints > Application Logic

Bad approach: Validate data in application code.

```
if (email.includes('@')) {
  await db.query('INSERT INTO users (email) VALUES ($1)', [email]);
}
```

Problem: Another app (or SQL client) can bypass validation.

Good approach: Enforce in the database.

```
CREATE TABLE users (
  email TEXT NOT NULL CHECK (email ~ '^.+@.+\..+$')
);
```

Benefits:

- **Single source of truth:** One place to enforce rules
- **Can't be bypassed:** Even raw SQL respects constraints
- **Self-documenting:** Schema describes invariants

Rule: If something **must** be true, enforce it in the schema.

NOT NULL: The Foundation

Default Behavior: Columns Are Nullable

```
CREATE TABLE users (
  name TEXT
);

INSERT INTO users (name) VALUES (NULL);  -- Allowed
```

Enforce Non-NULL

```
CREATE TABLE users (
  name TEXT NOT NULL
);

INSERT INTO users (name) VALUES (NULL);  -- ERROR: null value in column "name"
```

When to Use NOT NULL

✅ **Use on almost everything.** NULL complicates logic.

✅ **Especially:**

- Primary keys (always)
- Foreign keys (unless the relationship is optional)
- Business-critical fields (email, username, etc.)

❌ **Don't use when:**

- Value is genuinely optional (e.g., middle name)
- Foreign key is optional (e.g., manager_id for CEO)

Setting Defaults

```
CREATE TABLE users (
  active BOOLEAN NOT NULL DEFAULT true,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

INSERT without specifying values:

```
INSERT INTO users (name) VALUES ('Alice');
```

Result: `active = true`, `created_at = current timestamp`.

PRIMARY KEY: Unique Row Identifier

What It Does

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

Enforces:

1. **Uniqueness:** No two rows with the same `id`
2. **Non-NULL:** `id` cannot be NULL

Equivalent to:

```
CREATE TABLE users (
  id SERIAL UNIQUE NOT NULL,
  name TEXT NOT NULL
);
```

But `PRIMARY KEY` is clearer and has special meaning (e.g., for foreign keys).

Choosing Primary Keys

Option 1: Serial / Auto-Increment (Surrogate Key)

```sql
-- Postgres:
id SERIAL PRIMARY KEY

-- MySQL:
id INT AUTO_INCREMENT PRIMARY KEY
```

Benefits:

- Simple, guaranteed unique
- Small (integer)
- No business meaning (won't change)

Drawbacks:

- Sequential (predictable, can leak count)
- Not globally unique (per-table)

Option 2: UUID (Surrogate Key)

```sql
id UUID PRIMARY KEY DEFAULT gen_random_uuid()
```

Benefits:

- Globally unique (across tables, databases, servers)
- Non-sequential (less predictable)

Drawbacks:

- Larger (16 bytes vs 4/8 for INT/BIGINT)
- Slightly slower inserts (random B-tree inserts)

Option 3: Natural Key

```sql
CREATE TABLE countries (
  code CHAR(2) PRIMARY KEY,  -- 'US', 'CA', etc.
  name TEXT NOT NULL
);
```

Benefits:

- Self-explanatory (no need to join to get the code)

Drawbacks:

- Business keys can change (email, username, etc.)
- Composite keys are awkward

Best practice: Use surrogate keys (SERIAL or UUID) unless the natural key is immutable and stable.

Composite Primary Keys

```sql
CREATE TABLE order_items (
  order_id INT,
  product_id INT,
  quantity INT NOT NULL,
  PRIMARY KEY (order_id, product_id)
);
```

Enforces: Each (order_id, product_id) pair is unique.

Use case: Junction tables (many-to-many relationships).

UNIQUE Constraints: Enforce Uniqueness

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT NOT NULL UNIQUE
);
```

Equivalent to:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT NOT NULL,
  CONSTRAINT users_email_unique UNIQUE (email)
);
```

Multiple NULLs Are Allowed

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  phone TEXT UNIQUE
);

INSERT INTO users (phone) VALUES (NULL);
INSERT INTO users (phone) VALUES (NULL);  -- Allowed!
```

Why? `NULL != NULL`, so no uniqueness violation.

If you want only one NULL: Use a partial unique index (Postgres):

```
CREATE UNIQUE INDEX users_phone_unique ON users (phone) WHERE phone IS NOT NULL;
```

Composite UNIQUE Constraints

```
CREATE TABLE reservations (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL,
  date DATE NOT NULL,
  UNIQUE (user_id, date)
);
```

Enforces: Each user can have at most one reservation per date.

FOREIGN KEY: Referential Integrity

Basic Syntax

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL REFERENCES users(id)
);
```

Enforces:

1. `user_id` must exist in `users.id` (or be NULL, if nullable)
2. Can't delete a user if they have orders (unless cascade rules apply)

ON DELETE / ON UPDATE Actions

ON DELETE CASCADE

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL REFERENCES users(id) ON DELETE CASCADE
);
```

Behavior: If a user is deleted, all their orders are deleted too.

Use case: Dependent data that should be cleaned up (e.g., user sessions, logs).

Warning: Powerful, use carefully (accidental deletes cascade).

ON DELETE SET NULL

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT REFERENCES users(id) ON DELETE SET NULL
);
```

Behavior: If a user is deleted, `user_id` in orders becomes NULL.

Use case: Historical data where you want to preserve orders but anonymize the user.

ON DELETE RESTRICT (Default)

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL REFERENCES users(id) ON DELETE RESTRICT
);
```

Behavior: Can't delete a user if they have orders (error).

Use case: Prevent accidental deletion.

ON DELETE SET DEFAULT

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL DEFAULT 1 REFERENCES users(id) ON DELETE SET DEFAULT
);
```

Behavior: If a user is deleted, `user_id` becomes the default (e.g., "Unknown User").

Use case: Rare. Setting a sentinel value.

ON UPDATE: Same Options

Usually less important (primary keys rarely change), but `ON UPDATE CASCADE` can be useful:

```
REFERENCES users(id) ON UPDATE CASCADE
```

If `users.id` changes, `orders.user_id` updates automatically.

Composite Foreign Keys

```
CREATE TABLE order_items (
  order_id INT,
  product_id INT,
  quantity INT NOT NULL,
  FOREIGN KEY (order_id, product_id) REFERENCES order_product_combos(order_id,
product_id)
);
```

Rare, but useful for complex relationships.

Deferrable Constraints (Postgres)

By default, foreign keys are checked immediately (on each INSERT/UPDATE).

Problem: Circular dependencies.

```
-- Can't insert user without org, can't insert org without user!
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  org_id INT NOT NULL REFERENCES orgs(id)
);

CREATE TABLE orgs (
  id SERIAL PRIMARY KEY,
  owner_id INT NOT NULL REFERENCES users(id)
);
```

Solution: Deferrable constraints:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  org_id INT REFERENCES orgs(id) DEFERRABLE INITIALLY DEFERRED
);

CREATE TABLE orgs (
  id SERIAL PRIMARY KEY,
  owner_id INT REFERENCES users(id) DEFERRABLE INITIALLY DEFERRED
);
```

Behavior: Constraints are checked at **COMMIT** time, not immediately.

Use case: Rare. Mostly for complex migrations or circular dependencies.

CHECK Constraints: Custom Validation

Basic Syntax

```
CREATE TABLE users (
  age INT CHECK (age >= 18)
);
```

Enforces: Age must be at least 18.

```
INSERT INTO users (age) VALUES (17);  -- ERROR: check constraint violated
```

Named Constraints

```
CREATE TABLE products (
  price NUMERIC CHECK (price > 0),
  discount NUMERIC CHECK (discount >= 0 AND discount <= 1),
  CONSTRAINT price_after_discount CHECK (price * (1 - discount) > 0)
);
```

Naming constraints helps with error messages:

```
ERROR: new row violates check constraint "price_after_discount"
```

Multi-Column Checks

```
CREATE TABLE events (
  start_date DATE NOT NULL,
  end_date DATE,
  CHECK (end_date IS NULL OR end_date >= start_date)
);
```

Enforces: If `end_date` is set, it must be after `start_date`.

NULLs Pass CHECK Constraints

```
CREATE TABLE users (
  age INT CHECK (age >= 18)
);

INSERT INTO users (age) VALUES (NULL);  -- Allowed!
```

Why? `NULL >= 18` is UNKNOWN, and CHECK allows UNKNOWN.

To enforce non-NULL:

```
age INT NOT NULL CHECK (age >= 18)
```

Limitations of CHECK

What you can't do:

- Reference other tables (use triggers or application logic)
- Use subqueries (Postgres allows it but it's not efficient)

What you can do:

- Column-level or table-level checks
- Use functions (e.g., `CHECK (email ~ '^.+@.+')`)

EXCLUSION Constraints (Postgres Only)

Use case: Ensure no overlapping ranges (e.g., room reservations).

```
CREATE TABLE reservations (
  room_id INT,
  reserved_at TSTZRANGE,
  EXCLUDE USING GIST (room_id WITH =, reserved_at WITH &&)
);
```

Enforces: No two reservations for the same room can overlap in time.

Example:

```
INSERT INTO reservations VALUES (1, '[2024-01-01 10:00, 2024-01-01 12:00)');
INSERT INTO reservations VALUES (1, '[2024-01-01 11:00, 2024-01-01 13:00)');  --
ERROR
```

Why it's powerful: Prevents double-booking at the database level.

Requires: `btree_gist` extension for `=` operator.

Default Values

Static Defaults

```sql
CREATE TABLE users (
  status TEXT NOT NULL DEFAULT 'active',
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW()
);
```

Dynamic Defaults (Functions)

```sql
CREATE TABLE users (
  id UUID PRIMARY KEY DEFAULT gen_random_uuid(),
  created_at TIMESTAMPTZ DEFAULT NOW()
);
```

Note: `NOW()` evaluates once per transaction, not per row.

For per-row timestamps (Postgres):

```sql
created_at TIMESTAMPTZ DEFAULT CLOCK_TIMESTAMP()
```

Or use triggers.

Generated Columns (Postgres 12+, MySQL 5.7+)

Computed automatically:

```sql
CREATE TABLE products (
  price NUMERIC NOT NULL,
  tax_rate NUMERIC NOT NULL,
  price_with_tax NUMERIC GENERATED ALWAYS AS (price * (1 + tax_rate)) STORED
);
```

Behavior:

- `price_with_tax` is computed from `price` and `tax_rate`
- Updated automatically when `price` or `tax_rate` changes
- Can be indexed

STORED vs VIRTUAL:

- **STORED:** Computed on write, stored physically (faster reads)
- **VIRTUAL:** Computed on read (saves space, slower reads; not in Postgres 15-, only MySQL)

Naming Conventions

Good constraint names help debugging:

```sql
-- Bad:
CONSTRAINT users_check CHECK (age >= 18)

-- Good:
CONSTRAINT users_age_min_18 CHECK (age >= 18)
```

Error message:

```
ERROR: new row violates check constraint "users_age_min_18"
```

Clear what failed.

Standard Naming Pattern

```
{table}_{column}_{type}
```

Examples:

- `users_email_unique`
- `orders_user_id_fkey`
- `products_price_check`

Constraints vs Application Logic

Use Constraints When:

✅ **Invariants that must never be violated:**

- Email must be unique
- Price must be positive
- Foreign keys must exist

✅ **Data integrity:**

- Referential integrity (foreign keys)
- Uniqueness
- Not null

Use Application Logic When:

✅ **Business rules that change frequently:**

- Password strength requirements
- Coupon discount rules

✅ **Complex validation requiring external data:**

- "User must have valid credit card" (external API)

✅ **Performance-sensitive checks:**

- "User must not have >100 orders" (expensive query)

Rule of thumb: If it's a **hard invariant**, use a constraint. If it's a **soft rule**, use application logic.

Schema Design Patterns

Pattern 1: Soft Deletes

Instead of DELETE, mark as deleted:

```
CREATE TABLE users (
  id SERIAL PRIMARY KEY,
  email TEXT NOT NULL UNIQUE,
  deleted_at TIMESTAMPTZ
);
```

"Delete" user:

```
UPDATE users SET deleted_at = NOW() WHERE id = 123;
```

Query active users:

```
SELECT * FROM users WHERE deleted_at IS NULL;
```

Benefits:

- Preserve history
- Undo deletions
- Audit trail

Drawbacks:

- Complicates queries (always filter `deleted_at IS NULL`)
- UNIQUE constraints don't respect soft deletes (email collision)

Fix unique constraint issue:

```
CREATE UNIQUE INDEX users_email_unique ON users (email) WHERE deleted_at IS NULL;
```

Pattern 2: Audit Columns

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL REFERENCES users(id),
  total NUMERIC NOT NULL,
  created_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  updated_at TIMESTAMPTZ NOT NULL DEFAULT NOW(),
  created_by INT REFERENCES users(id),
  updated_by INT REFERENCES users(id)
);
```

Update `updated_at` with a trigger:

```
CREATE OR REPLACE FUNCTION update_updated_at()
RETURNS TRIGGER AS $$
BEGIN
  NEW.updated_at = NOW();
  RETURN NEW;
END;
```

```
$$ LANGUAGE plpgsql;

CREATE TRIGGER orders_updated_at
BEFORE UPDATE ON orders
FOR EACH ROW
EXECUTE FUNCTION update_updated_at();
```

Pattern 3: Enum Types (Postgres)

Instead of CHECK constraints:

```
CREATE TYPE order_status AS ENUM ('pending', 'processing', 'shipped', 'delivered',
'cancelled');

CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  status order_status NOT NULL DEFAULT 'pending'
);
```

Benefits:

- Type-safe (typos are errors)
- Self-documenting
- Can't insert invalid values

Drawbacks:

- Hard to change (adding enum values requires ALTER TYPE)
- Postgres-specific (MySQL has ENUM, but it's less strict)

Alternative: Lookup table:

```
CREATE TABLE order_statuses (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL UNIQUE
);

CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  status_id INT NOT NULL REFERENCES order_statuses(id)
);
```

More flexible, but requires joins.

Pattern 4: JSONB for Semi-Structured Data

Use case: Flexible attributes.

```
CREATE TABLE products (
  id SERIAL PRIMARY KEY,
  name TEXT NOT NULL,
  metadata JSONB NOT NULL DEFAULT '{}'
);
```

Insert:

```
INSERT INTO products (name, metadata) VALUES
  ('Laptop', '{"brand": "Dell", "ram_gb": 16}');
```

Query:

```
SELECT * FROM products WHERE metadata->>'brand' = 'Dell';
```

Index JSONB:

```
CREATE INDEX idx_products_metadata ON products USING GIN(metadata);
```

When to use:

- Attributes vary by product type
- Frequent schema changes

When NOT to use:

- Core business data (use proper columns)
- Need strong typing

Database Portability vs Features

Postgres-Specific Features

- `EXCLUDE` constraints
- `ENUM` types
- `DEFERRABLE` constraints
- `ARRAY` types
- `JSONB` with GIN indexes

If you need portability: Stick to ANSI SQL (PRIMARY KEY, FOREIGN KEY, UNIQUE, CHECK, NOT NULL).

If you're committed to Postgres: Use these features—they're powerful.

Practical Guidelines

Always Index Foreign Keys

```
CREATE TABLE orders (
  id SERIAL PRIMARY KEY,
  user_id INT NOT NULL REFERENCES users(id)
);

CREATE INDEX idx_orders_user_id ON orders(user_id);
```

Why? JOINs on foreign keys are common. Indexes make them fast.

Use NOT NULL Liberally

Default to NOT NULL unless the field is truly optional.

Validate in Both App and Database

Layered defense:

1. **Client-side:** Immediate feedback (UX)
2. **Server-side:** Security (don't trust client)
3. **Database:** Final enforcement (can't be bypassed)

Use Migrations, Not Manual DDL

Good:

```
// Prisma migration
await prisma.$executeRaw`
  ALTER TABLE users ADD CONSTRAINT users_age_min_18 CHECK (age >= 18);
`;
```

Bad:

```
-- Manual SQL (no version control, no rollback)
ALTER TABLE users ADD CONSTRAINT ...;
```

Use tools: Prisma Migrate, Drizzle Kit, Flyway, Liquibase.

Key Takeaways

1. **NOT NULL + defaults eliminate most NULL issues.**
2. **PRIMARY KEY enforces uniqueness + NOT NULL.** Use surrogate keys (SERIAL/UUID).
3. **FOREIGN KEY enforces referential integrity.** Use ON DELETE/UPDATE actions wisely.
4. **CHECK constraints enforce custom rules.** Use for simple validations.
5. **UNIQUE constraints prevent duplicates.** Multiple NULLs are allowed (usually fine).
6. **Naming constraints helps debugging.** Follow a consistent pattern.
7. **Constraints > application logic** for hard invariants.
8. **Always index foreign keys** for JOIN performance.
9. **Use migrations for schema changes.** Never manual DDL in production.
10. **Postgres-specific features are powerful.** Use them if you're committed to Postgres.

Next up: Postgres-specific features you should know about.