Don't Do That

Ensuring data sanity with database constraints

Joshua Tolley

End Point Corporation

April 24, 2012

ACID

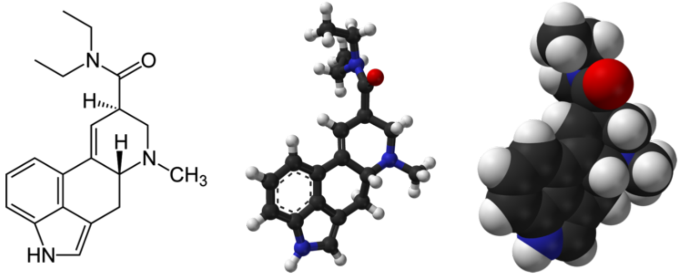

FIGURE: Lysergic acid diethylamide, public domain image courtesy of Benjah-bmm27, Wikipedia

ACID

We've all heard about ACID:

- **Atomicity**: Operations are grouped into transactions; each transaction either succeeds or fails in its entirety
- **Consistency**: At the end of each transaction, the data meet all applicable constraints
- **Isolation**: Data from uncommitted transactions are invisible to all but the transaction that created them
- **Durability**: Kicking the power cord doesn't destroy your data

Most developers ignore atomicity and consistency, don't understand isolation, and take durability for granted.

Just consistency, please

If you expect to hear about Atomicity, Isolation, or Durability, you're in the wrong room.

Database constraints

You get consistency from database constraints. Constraints ensure your data remain sane, meaningful, and unambiguous. If you do them right.

Why?

If you've ever dealt with "bad" data in the database, you'll understand why constraints are important

Database constraints

SQL databases maintain constraints in several ways:

- Check constraints
- Data type, including custom data types and domains
- UNIQUE, NOT NULL, DEFAULT (kinda)
- Primary and foreign keys
- Triggers
- Exclusion constraints

Check constraints

Check constraints simply verify a given expression

```
CREATE TABLE employee (
    manager_id INTEGER,  -- Employees with a manager
    team_id INTEGER,     -- must be assigned to a team
    salary FLOAT CHECK (salary > 0), -- Must be positive!
    CHECK ((manager_id IS NULL AND team_id IS NULL ) OR
       (manager_id IS NOT NULL AND team_id IS NOT NULL))
);
```

Check constraints

```
hal=# \d employee
          Table "public.employee"
   Column   |       Type       | Modifiers
------------+------------------+-----------
 manager_id | integer          |
 team_id    | integer          |
 salary     | double precision |
Check constraints:
    "employee_check" CHECK (manager_id IS NULL AND
      team_id IS NULL OR manager_id IS NOT NULL AND
      team_id IS NOT NULL)
    "employee_salary_check" CHECK
      (salary > 0::double precision)
```

Check constraints

```
hal=# insert into employee (salary) values (-10);
ERROR:  new row for relation "employee" violates check
  constraint "employee_salary_check"
hal=# insert into employee (salary) values (10);
INSERT 0 1
hal=# select * from employee;
 manager_id | team_id | salary
------------+---------+--------
            |         |     10
(1 row)
```

Check constraints

```
hal=# update employee set manager_id = 10;
ERROR:  new row for relation "employee" violates check
  constraint "employee_check"
hal=# update employee set team_id = 100;
ERROR:  new row for relation "employee" violates check
  constraint "employee_check"
hal=# update employee set team_id = 100,
  manager_id = 10;
UPDATE 1
```

Data types

Many data types include parameters of one sort or another. Most have inherent limitations that act as constraints.

- Integers are limited to a specific range, and have no fractional part
- VARCHAR() fields are often limited in length
- ENUM types can contain only values from a defined set
- Date and time types can contain only **VALID* dates**
- **Geometric, network, and other more complex types are also constrained**

** MySQL, are you listening?*

Custom data types

Many databases allow users to define their own data types

Composite data type

```
CREATE TYPE complex AS (
    r       double precision,
    i       double precision
);
```

User-defined data type

```
CREATE TYPE box (
    INTERNALLENGTH = 16,
    INPUT = my_box_in_function,
    OUTPUT = my_box_out_function,
    ELEMENT = float4
);
```

DOMAINS

The SQL standard includes *domains*, which combine a given data type with one or more check constraints (more on check constraints later).

```
CREATE DOMAIN us_postal_code AS TEXT
    CHECK(
        VALUE ~ '^\d{5}$'
        OR VALUE ~ '^\d{5}-\d{4}$'
);
```

Column constraints

Column definitions can include constraints:

- UNIQUE
- NULL / NOT NULL
  - DEFAULT (not really a constraint, but common with NOT NULL fields, and worth pointing out here)
- PRIMARY KEY
- REFERENCES ... (foreign keys)
- CHECK

UNIQUE, [NOT] NULL

- Unique columns must contain unqiue values
- "Unique" depends on the data type's definition of equality
- NULL means "unknown", so NULL != NULL, so UNIQUE columns can contain multiple NULLs
    - ...so you might consider including a NOT NULL
    - ...and perhaps a DEFAULT

Note

In PostgreSQL, UNIQUE is implemented with an index. Often, these are fields you'd likely want to index anyway. Users can declare the field "UNIQUE"; the index will be created and named automatically.

PRIMARY KEY

Primary keys are UNIQUE and NOT NULL. Tables may have only one primary key, but many UNIQUE + NOT NULL columns.

FOREIGN KEY

```
CREATE TABLE employee (
  id SERIAL PRIMARY KEY,
  manager_id INTEGER REFERENCES employee (id),
  team_id INTEGER REFERENCES team (id),
  salary FLOAT CHECK (salary > 0),
  CHECK ((manager_id IS NULL AND team_id IS NULL ) OR
    (manager_id IS NOT NULL AND team_id IS NOT NULL))
 );
```

Note

In PostgreSQL, columns referenced in a foreign key must be declared UNIQUE

FOREIGN KEY - Cascading

What happens when one of the teams or managers gets deleted?

```
hal=# delete from team where id = 100;
ERROR:  update or delete on table "team" violates
  foreign key constraint "team_fkey" on table
  "employee"
DETAIL:  Key (id)=(100) is still referenced from
  table "employee".
```

FOREIGN KEY - Cascading

`ON UPDATE` *`action`* and `ON DELETE` *`action`*

- **NO ACTION**: Throw an error saying that the action would break consistency
- **RESTRICT**: Same as "NO ACTION", but not deferrable
- **CASCADE**: Delete or update all rows referencing this row
- **SET NULL**: Set referencing columns to NULL
- **SET DEFAULT**: Set the referencing columns to their default values.

Cascading

```
... team_id REFERENCES team (id) ON UPDATE CASCADE
```

```
hal=# select * from employee;
 id | manager_id | team_id | salary
----+------------+---------+--------
  1 |          1 |     100 |     10
(1 row)

hal=# update team set id = 101 where id = 100;
UPDATE 1
hal=# select * from employee;
 id | manager_id | team_id | salary
----+------------+---------+--------
  1 |          1 |     101 |     10
(1 row)
```

Deferred constraints

```
... team_id REFERENCES team (id) DEFERRABLE
hal=# begin;
hal=# set constraints employee_team_id_fkey deferred;
hal=# update team set id = 101;
hal=# update employee set team_id = 101 where
   team_id = 100;
hal=# commit;
COMMIT
hal=# select * from employee;
 id | manager_id | team_id | salary
----+------------+---------+--------
  1 |          1 |     101 |     10
(1 row)
```

Multi-column constraints

These constraints can apply to multiple columns

- CREATE UNIQUE INDEX foo ON bar (baz, qux)
- CREATE TABLE CONSTRAINT foo_fkey FOREIGN KEY (bar, baz) REFERENCES alpha (bar, baz)

Note

You can even set multi-column foreign keys so some of the columns can be NULL, but it's rarely used. Google "foreign key match clause" for more.

Advanced index-based constraints

Index-based constraints can become more flexible when used with functional or partial indexes

- CREATE UNIQUE INDEX ix1 ON foo (lower(bar));
- CREATE UNQIUE INDEX ix2 ON employee (name) WHERE (team_id = 100);

Triggers

Triggers run user-defined functions (UDFs) when various things happen. Details of UDF programming are beyond this talk, but here's an example

```
CREATE FUNCTION sample() RETURNS TRIGGER AS $$
DECLARE
  msg TEXT; i INTEGER;
BEGIN
  IF NEW.jurisdiction_id IS NOT NULL AND NOT EXISTS (
    SELECT 1 FROM places p
      JOIN places_types pt ON (pt.place_id = p.id)
      JOIN codes c ON (c.the_code = 'J' AND c.id = pt.type_id)
    WHERE p.id = NEW.jurisdiction_id
  ) THEN
    RAISE EXCEPTION 'Error. Place % is not a jurisdiction.',
        NEW.jurisdiction_id;
    RETURN NULL;
  END IF;
  RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER sample_trigger BEFORE INSERT OR UPDATE
    ON some_table FOR EACH ROW EXECUTE PROCEDURE sample();
```

Triggers

Note

PostgreSQL has at least 18 different languages available for user-defined functions. These include PL/pgSQL (like Oracle's PL/SQL), Perl, Python, Tcl, Javascript, Lua, Java, Ruby, and LOLCODE. Not all these languages support triggers.

Note that defining a trigger to validate data will *not* automatically validate the data already in the table.

Triggers

Triggers can do all kinds of neat things:

- Validate new and modified data
- Log users' behavior
- Calculate hidden fields
- Make views that work like tables
- Launch the missiles

Why PostgreSQL Rocks: Exclusion Constraints

A UNIQUE constraint can be generalized. Unique constraints say "don't allow data where the equality operator for this data type returns true for a new row and some existing row". What if we weren't limited to the equality operator?

Why PostgreSQL Rocks: Exclusion Constraints

```
hal=# CREATE TABLE circles (
    my_circle circle,
    name text,
    EXCLUDE USING gist ( my_circle WITH && )
);
```

&& is the "overlaps" operator for PostgreSQL's native circle type. So this says "don't allow circles to overlap."

Note

Only PostgreSQL allows you to do this.

Exclusion Constraints

```
josh=# insert into circles values
  ('0, 0, 10', 'first);
INSERT 0 1
hal=# select * from circles ;
my_circle   | name
------------+-------
<(0,0),10> | first
(1 row)

hal=# insert into circles values
  ('5,0,10', 'second');
ERROR:  conflicting key value violates exclusion
  constraint "circles_my_circle_excl"
DETAIL:  Key (my_circle)=(<(5,0),10>) conflicts
  with existing key (my_circle)=(<(0,0),10>).
```

Refutation and Rebuttal

Some application frameworks* claim they handle all data validation in the application, so the database doesn't need to worry about it. What could possibly go wrong?

** Rails, I'm talking to you*

Refutation and Rebuttal

What if...

- some other process access the database outside your application?
- the database were smart enough to optimize queries based on the constraints you declared, but you didn't declare them?
- the database could process the constraints faster than the application can?
- the constraints could be cleaner, simpler, and more straightforward in SQL?

Please...

Don't let your data run wild. Who knows what it might do...