Don't Do That

Ensuring data sanity with database constraints

Joshua Tolley, End Point Corporation

April 19, 2012

ACID

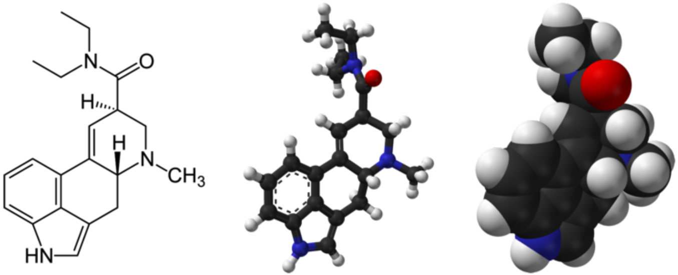

Figure: Lysergic acid diethylamide, public domain image courtesy of Benjah-bmm27, Wikipedia

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 simple verify a given expression

```
CREATE TABLE foo (
    i INTEGER,
    j INTEGER,
    p FLOAT CHECK (p > 0),
    CHECK ( (i IS NULL) OR (j IS NULL))
);

hal=# \d foo
          Table "public.foo"
 Column |       Type       | Modifiers
--------+------------------+-----------
 i      | integer          |
 j      | integer          |
 p      | double precision |
Check constraints:
    "foo_check" CHECK (i IS NULL OR j IS NULL)
    "foo_p_check" CHECK (p > 0::double precision)
```

Check constraints

```
hal=# insert into foo (p) values (-10);
ERROR:  new row for relation "foo" violates check constraint
    "foo_p_check"

hal=# insert into foo (p) values (10);
INSERT 0 1
hal=# select * from foo;
 i | j | p
---+---+----
   |   | 10
(1 row)

hal=# update foo set i = 1;
UPDATE 1
hal=# update foo set j = 1;
ERROR:  new row for relation "foo" violates check constraint
    "foo_check"
```

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)
- 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

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 foo (
    bar INTEGER REFERENCES baz (qux)
);
```

Note

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

FOREIGN KEY - Cascading

What happens when a value in a referenced table gets modified?

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.

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.

Triggers

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

Note

PostgreSQL has at least 18 different languages available for user-defined functions