**Utility Operators, Keywords, and Functions(Postgres)**

GREATEST

Gives the greatest value from the values provided

**SELECT GREATEST(200, 10, 30);**

Example

Compute the cost to ship each item

Shipping is the maximum of (weight \* $2) or $30

**SELECT name, weight, GREATEST(30, 2 \* weight)**

**FROM products;**

LEAST

This will give the least value from the list given

**SELECT LEAST(1, 20, 50, 100)**

Example

All products are on sale and the rule to calculate the sale price of a product is

Price is the least of the products price \* 0.5 or $400

**SELECT name, price, LEAST(price \* 0.5, 400)**

**FROM products;**

**The case keyword**

Example

Print each product and its price. Also, print a description of the price

If price > 600 then ‘high’, if price > 300 then ‘medium’ else print ‘cheap’

**SELECT**

**name,**

**price,**

**CASE**

**WHEN price > 600 THEN 'high'**

**WHEN price > 300 THEN 'medium'**

**ELSE 'cheap'**

**END**

**FROM products;**

**Else keyword isn’t compulsory (but if there is a row that doesn’t satisfy the two conditions, then we will end up with a default value of 1**

**PGAdmin**

* Tool to manage and inspect a Postgres database
* Can connect to local or remote databases
* Can view/change just about anything in PG

Postgres Server

* A PG Server can contain multiple databases
* All data for a single app lives in a single DB
* Having multiple DB’s is more about working with more than one app on your machine

**Data types**

* Integer
* Date/time
* Boolean
* Currency
* Character
* Range
* Xml
* Binary
* JSON
* Arrays
* UUID
* Geometric

**Numeric types**

* Smallint, integer and bigint -> Numbers without any decimal points
* Smallserial, serial and bigserial -> No decimal point, auto increment
* Decimal, numeric, real, double precision and float -> Number with decimal points

|  |  |
| --- | --- |
| **Numeric types** | **Range** |
| Smallint | -32768 to 32767 |
| Integer | -2147583648 to 2174483647 |
| Bigint | -922337203685775808 to 9223372036854775807 |
| smallserial | 1 to 32767 |
| serial | 1 to 2147483647 |
| bigserial | 1 to 9223372036854775807 |
| decimal | 131072 digits before decimal point, 16383 after |
| Numeric | 131072 before decimal point, 16383 after |
| Real | 1E-37 to 1E37 with at least 6 places precision |
| Double precision | 1E-307 to 1E308 with at least 15 place precision |
| float | Same as real or double precision |

**Fast Rules on Numeric Data Types**

|  |  |  |
| --- | --- | --- |
|  | ‘id’ column of any table | Mark the column as **serial** |
|  | Need to store a number without a decimal | Mark the column as **integer** |
| Bank balance, grams of gold, scientific calculations | Need to store a number **with a decimal and this data needs to be very accurate** | Mark the column as **numeric** |
| Kilograms of trash in a landfill, litres of water in a lake, air pressure in a tire | Need to store a number **with a decimal and the decimal doesn’t make a big difference** | Mark the column as **double precision** |

Floats are notoriously known for being inaccurate

SCRIPTS

SELECT 2 + 2; // returns a column with value 4

SELECT (2.0::INTEGER); // converts the datatype from numeric to integer

**Character Types**

CHAR(5) – Store some characters, length will always be 5 even if PG has to insert spaces

VARCHAR – Store any length of string

VARCHAR(40) – Store a string up to 40 characters, automatically remove extra characters

TEXT – Store any length of string

*NB: There is no performance difference in these character types unlike other types of databases*

**Boolean Types**

true, yes, on, 1, t, y - TRUE

false, no, off, 0, f, n - FALSE

null - NULL

Example

SELECT ('y'::BOOLEAN); // returns TRUE

SELECT ('n'::BOOLEAN); // returns FALSE

SELECT (NULL::BOOLEAN); // returns [null]

**DATE**

1980-11-20 -> 20 November 1980

Nov-20-1980 -> 20 November 1980

20-Nov-1980 -> 20 November 1980

1980-Novemberr-20 -> 20 November 1980

November 20, 1980 -> 20 November 1980

***NB: PostgreSQL converts any date format into a fixed date value as above***

***SELECT ('NOV-20-1980'::DATE);***

***SELECT ('NOV 20 1980'::DATE);***

***SELECT ('NOV,20,1980'::DATE);***

***SELECT ('NOV 20,1980'::DATE);***

*Gives –> 1980-11-20*

**TIME, Time without time zone**

01:23 AM -> 01:23, no time zone

05:23 PM -> 17:23, no time zone

20:34 -> 20:34, no time zone

**SELECT ('01:23 AM'::TIME); SELECT ('01:23 PM'::TIME WITHOUT TIME ZONE);**

Returns 01:23:00

**SELECT ('01:23 PM'::TIME);**

Returns 13:23:00

**SELECT ('02:30'::TIME);**

Returns 02:30:00

**TIME WITH TIME ZONE**

01:23 AM EST -> 01:23-05:00 (The minus 5 indicates, 5 hrs behind UTC time)

05:23 PM PST -> 17:23-08:00

05:23 PM UTC -> 17:23+00:00

05:23 PM UTC -> 17:23+00:00

**SELECT ('01:23:23 AM EST'::TIME WITH TIME ZONE);**

**SELECT ('01:23:23 AM Z'::TIME WITH TIME ZONE); (z is utc)**

**SELECT ('01:23:23 AM UTC'::TIME WITH TIME ZONE); (utc)**

**TIMESTAMP WITH TIME ZONE**

Nov-20-1980 05:23 PM PST -> 1980-11-20 18:23:00-07

**SELECT ('NOV-20-1980 1:23 AM PST'::TIMESTAMP WITH TIME ZONE);**

Returns -> 1980-11-20 19:23:00+10

**INTERVAL**

1 day -> 1 day

1 D -> 1 day

1 D 1M 1S -> 1 day 1 minute 1 second

Think of an interval as a duration of time

**SELECT ('1 day'::INTERVAL);**

**SELECT ('1 D'::INTERVAL);**

**SELECT ('1 D 20 H'::INTERVAL);**

Returns -> 1 day 20:00:00

**SELECT ('1 D 20 H 30 M 40 S'::INTERVAL);**

Returns -> 1 day 20:30:40

Can perform operations like:

**SELECT ('1 D 20 H 30 M 40 S'::INTERVAL) - ('1 D'::INTERVAL);**

Returns -> 20:30:40

**SELECT ('NOV-20-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE) - ('1 D'::INTERVAL);**

Returns -> 1980-11-19 16:23:00+10

**SELECT**

**('NOV-20-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE)**

**-**

**('NOV-10-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE);**

Returns -> 10 days

It can work with different time zones as well:

**SELECT**

**('NOV-20-1980 1:23 AM EST':: TIMESTAMP WITH TIME ZONE)**

**-**

**('NOV-10-1980 5:43 AM PST':: TIMESTAMP WITH TIME ZONE);**

Returns -> 9 days 16:40:00

**Database-Side Validation and Constraints**

**Applying a NULL Constraint**

**Row-Level Validation**

* Is a given value defined?
* Is a value unique in its column
* Is a value >, <, >=, <=, = some other value?

You can always add the constraint NOT NULL to prevent null values from being inserted into the table

When Creating the table, this is how you do it for price:

**CREATE TABLE products (**

**Id SERIAL PRIMARY KEY,**

**name VARCHAR(50),**

**department VARCHAR(50),**

**price INTEGER NOT NULL,**

**weight INTEGER**

**);**

When the Table has already been created:

**ALTER TABLE products**

**ALTER COLUMN price**

**SET NOT NULL;**

NB: The constraint cannot be set if there are already null values in the table. The null values have to be converted to an actual values for the constraint to be applied or the null value rows could be deleted

Example of converting a price column with null values

**UPDATE products**

**SET price = 9999**

**WHERE price IS NULL;**

Now that the price column has no null values in it, the constraint can be applied.

**ALTER TABLE products**

**ALTER COLUMN price**

**SET NOT NULL;**

**DEFAULT COLUMN VALUES**

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Department VARCHAR(50) NOT NULL,

Price INTEGER DEFAULT 999,

Weight INTEGER

);

After The Table Was Created

ALTER TABLE products

ALTER COLUMN price

SET DEFAULT 999;

**Applying a Unique Constraint to One Column**

A ***UNIQUE*** constraint ensures values in a particular column are not duplicated

In the case below, the UNIQUE constraint is used to ensure that the column name does not have products with the same name

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL UNIQUE,

Department VARCHAR(50) NOT NULL,

Price INTEGER DEFAULT 999,

Weight INTEGER

);

After The Table Was Created

ALTER TABLE products

ADD UNIQUE (name);

NB: You cannot add(alter) the UNIQUE constraint unless all values inside the table are already unique values

**Multi-Column Uniqueness**

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Department VARCHAR(50) NOT NULL,

Price INTEGER DEFAULT 999,

Weight INTEGER,

UNIQUE(name, department)

);

After The Table Was Created

ALTER TABLE products

ADD UNIQUE (name, department);

Dropping Constraints

ALTER TABLE products

DROP CONSTRAINT products\_name\_key;

ALTER TABLE products

DROP CONSTRAINT products\_name\_department\_key;

**Adding a Validation Check**

The CHECK below looks if the data being inserted or updated in the price column is greater than 0.

When Creating The Table

CREATE TABLE products (

Id SERIAL PRIMARY KEY,

Name VARCHAR(50) NOT NULL,

Department VARCHAR(50) NOT NULL,

Price INTEGER CHECK (price > 0),

Weight INTEGER,

UNIQUE(name, department)

);

After The Table Was Created

ALTER TABLE products

ADD CHECK (price > 0);

NB: A check can only work on the row we are adding/updating. We cannot apply a check if all the rows inside our existing table don’t already satisfy the check.

**Checks Over Multiple Columns**

The CHECK below ensures that an order is delivered after it is created

CREATE TABLE orders (

id SERIAL PRIMARY KEY,

name VARCHAR(50) NOT NULL,

created\_at TIMESTAMP NOT NULL,

est\_delivery TIMESTAMP NOT NULL,

CHECK (created\_at < est\_delivery)

);

The check ensures only the colors red, green and blue are added into the database

CREATE TABLE cars (

id SERIAL PRIMARY KEY,

name VARCHAR(20),

color VARCHAR(20) CHECK (color IN ('red', 'green', 'blue'))

);

NB: We can only make use of columns inside the check statement inside the row that we are trying to insert

**Where to Apply Validation**

|  |  |
| --- | --- |
| **Web Server** | **Database** |
| Easier to express more complex validation | Validation still applied even if you connect with a different client |
| Fare easier to apply new validation rules | Guaranteed that validation is always applied |
| Many libraries to handle validation automatically | Can only apply new validation rules if all existing rows satisfy it |

NB: Better if you put the bulk of your validation at the web server level and only put some very critical ones in the database

**SQL Schema Designers**

1. Dbdiagram.io
2. Drawsql.app
3. Sqldbm.com
4. Quickdatabasediagrams.com
5. Ondras.zarovi.cz/sql/demo

Check on draw.io (can create good visual diagrams)

**A screenshot of a computer

Description automatically generated**

**How to build a like system**

Rules of likes

* Each user can like a specific post a single time
* A user should be able to ‘unlike’ a post
* Need to be able to figure out how many users like a post
* Need to be able to list which users like a post
* Something besides a post might need to be liked (comments)
* We might want to think about ‘dislikes’ or other kinds of reactions

How To Not Design A Like System

Do not add a ‘likes’ column to posts

* No way to make sure a user likes a post only once
* No way to make sure a user can only ‘unlike’ a post they have liked
* No way to figure out which users like a particular post
* No way to remove a like if a user gets deleted

Designing a Like system

Create a likes table with user\_id and post\_id

Assign a unique constraint to user\_id and post\_id such that a user cannot like a post twice -> UNIQUE(user\_id, post\_id)

Disadvantage of the system above – Can’t really track or know the exact reaction of the like of a post or track the dislikes

**Making a Reaction System Instead**

Make a reactions table with these columns: id, user\_id, post\_id, type

**Polymorphic Associations**

A like can be a ‘post like’ or a ‘comment like’. (polymorphic association)

Requires your app to figure out the meaning of each like

Can’t use foreign key columns – ‘liked\_id’ is a plain integer

Not recommended, but may still see it in use

Best Solution (use this)

Create a likes table with the following columns: id, user\_id, post\_id, comment\_id

|  |  |  |  |
| --- | --- | --- | --- |
| Likes | | | |
| Id | User\_id | Post\_id | Comment\_id |
| 1 | 3 | 1 | NULL |
| 2 | 1 | NULL | 3 |

The table above allows to associate a user with posts or comments and still get to use foreign key columns.

First row indicates a like for a post with an ID of one.

Second row indicates a like for a comment with an id of 3.

We need to make sure we don’t have a scenario where there are two values of post\_id and comment\_id defined or both the post\_id and comment\_id are null

Use the following Constraint:

**ADD CHECK of**

**(COALESCE((post\_id)::BOOLEAN::INTEGER,0)**

**+**

**COALESCE((comment\_id)::BOOLEAN::INTEGER, 0)) = 1**

*COALESCE returns the first value that is not null*

e.g.

SELECT COALESCE(NULL, 5); // returns 5

SELECT COALESCE(10, 5); // returns 10

SELECT (NULL)::BOOLEAN::INTEGER; // returns null(If put in a Coalesce function, coalesce is not going to return any value for that)

SELECT (534)::BOOLEAN::INTEGER; //returns 1

**The Simplest Alternative**

Each type of like gets its own table

Still want to write queries that will count up all likes? *You can use a Union or a View*

Creating a Posts\_likes and Comments\_likes table

The only downside is if a user has to like different kinds of things, we may end up creating many different kinds of tables

**How To Build A Mention System**

**Additional Features around Posts**

REAL datatype goes up to 6 decimals

Photo Mentions vs Caption Mentions

* Need to show a list of posts a user was mentioned in?
* Need to show a list of the most-often mentioned users?
* Need to notify a user when they’ve been mentioned?

Considerations on Photo Tags vs Caption Tags

Tag Solution 1

We could have a single table of tags with the following columns: id, user\_id, post\_id, x and y coordinates.

If the X and Y coordinates are NULL, then it means it’s a tag in a Caption.

Tag Solution 2

Create 2 tag tables: photo\_tags and caption tags.

Their difference being that the photo\_tags have the x and y coordinate and the caption tags have only the user\_id and post\_id

Which ones better?

Do you expect to query for caption\_tags and photo\_tags at different rates?

Will the meaning of a photo\_tag change at some point?

If you may expect any changes, solution 2 is the best, either way solution 2 is very flexible and convenient for any future changes.

Table users {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  updated\_at TIMESTAMP

  username VARCHAR(30)

}

Table posts {

  id SERIAL [pk, increment]

  title VARCHAR

  url VARCHAR(200)

  user\_id INTEGER [ref: > users.id]

  caption VARCHAR(240)

  lat REAL

  lng REAL

}

Table comments {

  id SERIAL [pk, increment]

  contents VARCHAR(240)

  user\_id INTEGER [ref: > users.id]

  post\_id INTEGER [ref: > posts.id]

}

Table likes {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  user\_id INTEGER [ref: > users.id]

  post\_id INTEGER [ref: > posts.id]

  comment\_id INTEGER [ref: > comments.id]

}

Table photo\_tags {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  updated\_at TIMESTAMP

  post\_id INTEGER [ref: > posts.id]

  user\_id INTEGER [ref: > users.id]

  x iNTEGER

  y INTEGER

}

Table caption\_tags {

  id SERIAL [pk, increment]

  created\_at TIMESTAMP

  post\_id INTEGER [ref: > posts.id]

  user\_id INTEGER [ref: > users.id]

}

**How to Build a ‘Hashtag’ System**

Seeing hashtags used in posts, comments, and user bio’s might make you think we need some kind of relationship for each.

We only have to model resources in the DB if we expect to query for them at some point!

Do we expect to run a query to see what posts/comments/users contain a given hashtag?

Can search for posts that contain a hashtag – implies that hashtags in a post’s caption are modelled in the db!

We can’t search for comments or user with a hashtag – implies they are not modelled! (or that we don’t have to)

**Tables for hashtags**

Solution 1

A table of hashtags with columns: id, title and post\_id

This would work but for some performance reasons try using this below:

A table of hashtags with columns: id, title

A table of hashtag\_posts with columns: id, hashtag\_id, post\_id

The reason for the performance reasons is that there were a lot of words being duplicated and any time there is duplication of strings, there may be a concern of space and may be using up a little more storage than what we would want to use.

The second table stores and references the hashtags(hashtag\_id) through an integer, which is less storage consuming.

**A few more user columns**

 bio VARCHAR(400)

 avatar VARCHAR(200)

phone VARCHAR(25)

email VARCHAR(40)

password VARCHAR(50)

 status VARCHAR(15)

**Why No Number of Followers or Posts?**

# posts and # followers can be calculated by running a query on data that already exists in our DB. We call this **‘derived data’.** We generally don’t want to store derived data

**How to design a ‘Follower’ System**

Create a table of followers with columns: id, leader\_id, follower\_id

CHECK (user\_id<>follower\_id)

UNIQUE (leader\_id, follower\_id) # We can never have two rows with an identical leader\_id and follower\_id

**Implementing Database Design Patterns**

1. Create a new db using PGAdmin
2. Convert our design to a series of CREATE TABLE statements
3. Insert data into the database
4. Write some queries
5. Realize that a few things could have been designed better! Make some changes

NOT NULL (A value must be provided (empty strings are values!)

DEFAULT (Provide a default value if an INSERT statement doesn’t give one)

NOT NULL + DEFAULT (We always want a value, but it should be optional)

Latitude values range from -90 < lat < 90

Longitude values range from -180 < long < 180