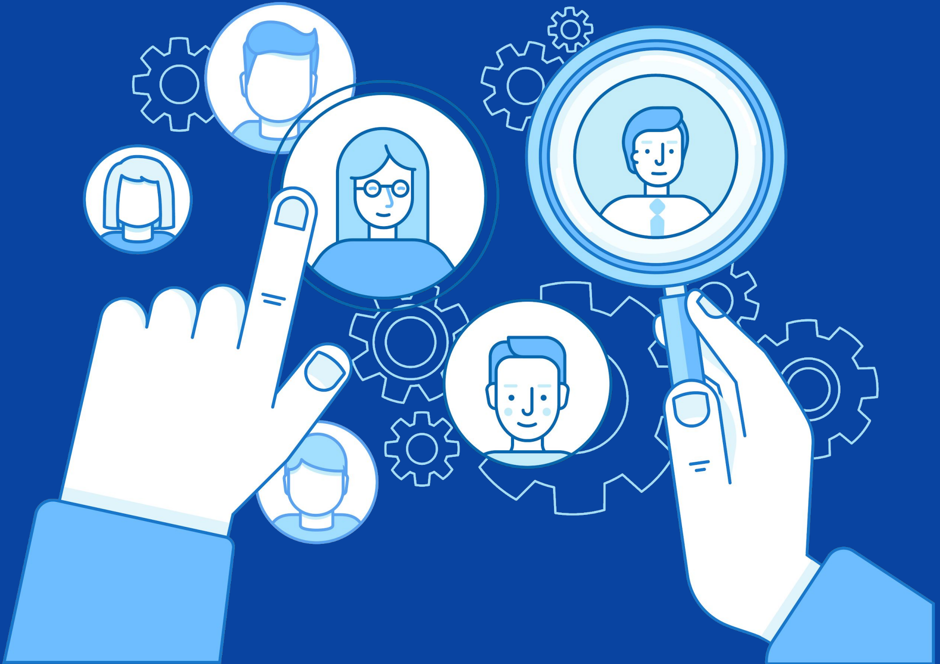
Digital Career Institute

Python Course - Database - Basic Usage



Digital Career Institute

Columns & Data Types

PostgreSQL Data Types

PostgreSQL has a variety of data types available.

- bigint
- bigserial
- bit
- bit varying
- boolean
- box
- bytea
- character
- character varying
- cidr
- circle
- date
- double precision
- inet
- integer
- interval
- json
- jsonb
- line
- lseg
- macaddr
- macaddr8
- money
- numeric
- path
- pg_lsn
- pg_snapshot
- point
- polygon
- real
- smallint
- smallserial
- serial
- text
- time
- time with time zone
- timestamp
- timestamp with time zone
- tsquery
- tsvector
- txid_snapshot
- uuid
- xml

PostgreSQL Data Types

In this submodule we will focus on:

- Boolean Type
- Numeric Types
- Text Types

Values vs. No-Values

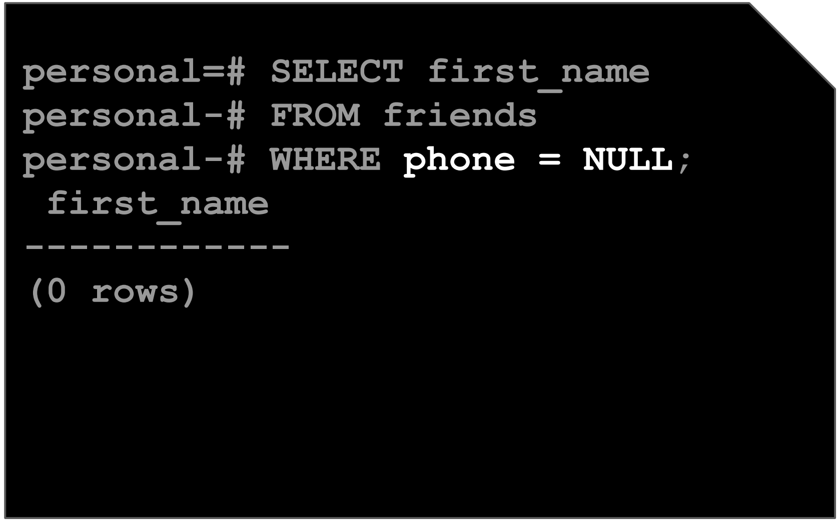
All types allow the data to be unset, with no value.

This state is named `NULL`.

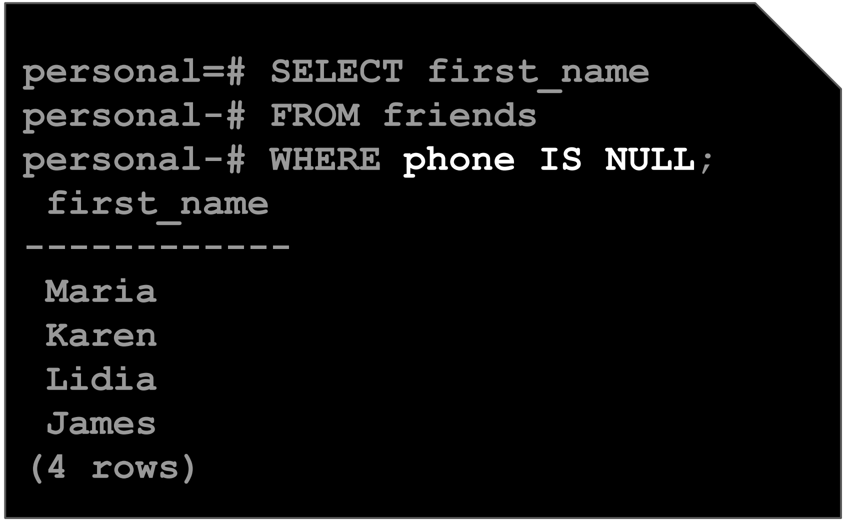
Sometimes it is called *NULL value*,
but it is technically not a value.

`NULL` represents the absence of a value.

Retrieve No-Values



```
personal=# SELECT first_name
personal-# FROM friends
personal-# WHERE phone = NULL;
 first_name
------------
(0 rows)
```



```
personal=# SELECT first_name
personal-# FROM friends
personal-# WHERE phone IS NULL;
 first_name
------------
 Maria
 Karen
 Lidia
 James
(4 rows)
```

To check if a row has no value we cannot do `column = NULL` because the `=` operator works only with values.

Instead, the query must be defined as `column IS NULL`.

Define Columns Without No-Values

```sql
CREATE TABLE private.friends (
  first_name   varchar(20) NOT NULL,
  last_name    varchar(50),
  phone        varchar(12),
  age          integer
);
```

The `NOT NULL` construct will not allow
NULL values in the column.

The Boolean Type

```
CREATE TABLE friends (
  first_name   varchar(20),
  last_name    varchar(50),
  age          integer,
  from_school  boolean
);
```

A boolean column will accept any of the following states:

- TRUE
- FALSE
- NULL

A **`boolean`** column may contain a boolean value, or no value at all. Therefore, it is a **three-state switch**.

The Boolean Type

```sql
UPDATE friends
SET from_school = TRUE;
UPDATE friends
SET from_school = 'yes';
UPDATE friends
SET from_school = 'on';
UPDATE friends
SET from_school = 1;
```

A boolean column may be set to **`TRUE`** with any of these values:

- TRUE
- yes
- on
- 1

The Boolean Type

```
UPDATE friends
SET from_school = FALSE;
UPDATE friends
SET from_school = 'no';
UPDATE friends
SET from_school = 'off';
UPDATE friends
SET from_school = 0;
```

A boolean column may be set to **`FALSE`** with any of these values:

- FALSE
- no
- off
- 0

The Numeric Types

There is a variety of numeric types
that can be grouped into:

Integer Types

Decimal Types

The Numeric Types: Integers

Different integer types are provided
to optimize the database.

	SMALLINT	INTEGER	BIGINT
STORAGE	2 bytes	4 bytes	8 bytes
MIN. VALUE	-32768	-2147483648	-9223372036854775808
MAX. VALUE	+32767	+2147483647	+9223372036854775807

The Numeric Types: Integers

PostgreSQL validates against each type.

```
CREATE TABLE friends (
  first_name    varchar(20),
  last_name     varchar(50),
  age           smallint
);
```

```
=# INSERT INTO friends(age)
-# VALUES(50000);
ERROR:  smallint out of range
```

The Numeric Types: Serial Integers

Serial types are
auto-incrementing integers.

	SMALLSERIAL	SERIAL	BIGSERIAL
STORAGE	2 bytes	4 bytes	8 bytes
MIN. VALUE	1	1	1
MAX. VALUE	32767	2147483647	9223372036854775807

The Numeric Types: Serial Integers

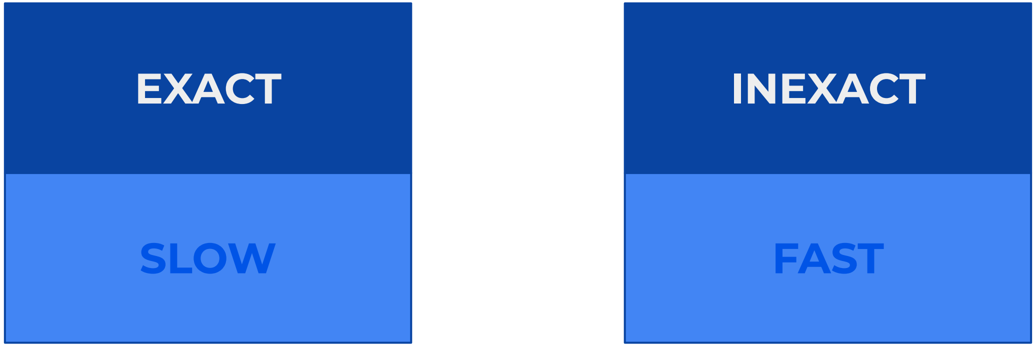
Inserting data will auto populate the serial column.

```
CREATE TABLE tasks (
  id        serial,
  name      varchar(30)
);
```

```
=# INSERT INTO tasks(name)
-# VALUES('Iron'),('Clean'),
-#       ('Study'),('Cook');
INSERT 0 4
=# SELECT * FROM tasks;
 id | name
----+-------
  1 | Iron
  2 | Clean
  3 | Study
  4 | Cook
(4 rows)
```

The Numeric Types: Decimals

Decimal types can be divided into **exact** and **inexact** decimals.



EXACT

SLOW

INEXACT

FAST

Exact types produce exact results when used in calculations.

The Numeric Types: Exact Decimals

There are two exact types, but they are equivalent.

DECIMAL = **NUMERIC**

The Numeric Types: Exact Decimals

The numeric type has two parameters:

```
NUMERIC(<precision>, <scale>);
```

`<precision>` is the total amount of digits (to both the right and left of the comma) that can be stored for each value.

`<scale>` is the total amount of decimal digits the column may store for each value. That is, the amount of digits to the right of the comma.

The Numeric Types: Exact Decimals

```
CREATE TABLE people (
  id      serial,
  height  numeric(3, 2)
);
```

Valid values:

- 1.62
- 2.32
- 9.99
- 0.01
- 1.00
- -3.50

Invalid values:

- 21.29
- 1.12345

The Numeric Types: Exact Decimals

The numeric type can also be used
with only one parameter:

```
NUMERIC(<precision>);
```

The `<scale>` will be set to 0. So the field will only accept integer values.

The Numeric Types: Exact Decimals

The numeric type can even be used without any parameter:

```
NUMERIC;
```

The column will accept any value of any **`<precision>`** and **`<scale>`**.

There will be no limitation to the amount of digits that can be stored.

The Numeric Types: Inexact Decimals

There are two inexact types.

	REAL	DOUBLE PRECISION
STORAGE	4 bytes	8 bytes
PRECISION	6	15

The Text Types

There are 3 types of text columns:

	CHARACTER	CHARACTER VARYING	TEXT
LENGTH	FIXED*	VARIABLE	VARIABLE
LIMIT	YES	YES	NO
ALIAS	CHAR	VARCHAR	-

* The fixed-length type will fill up the remaining characters with white spaces.

The Text Types

```
CREATE TABLE people (
  id           serial,
  name         varchar(50),
  id_card      char(10),
  description  text
);
```

Different situations may require different text types.

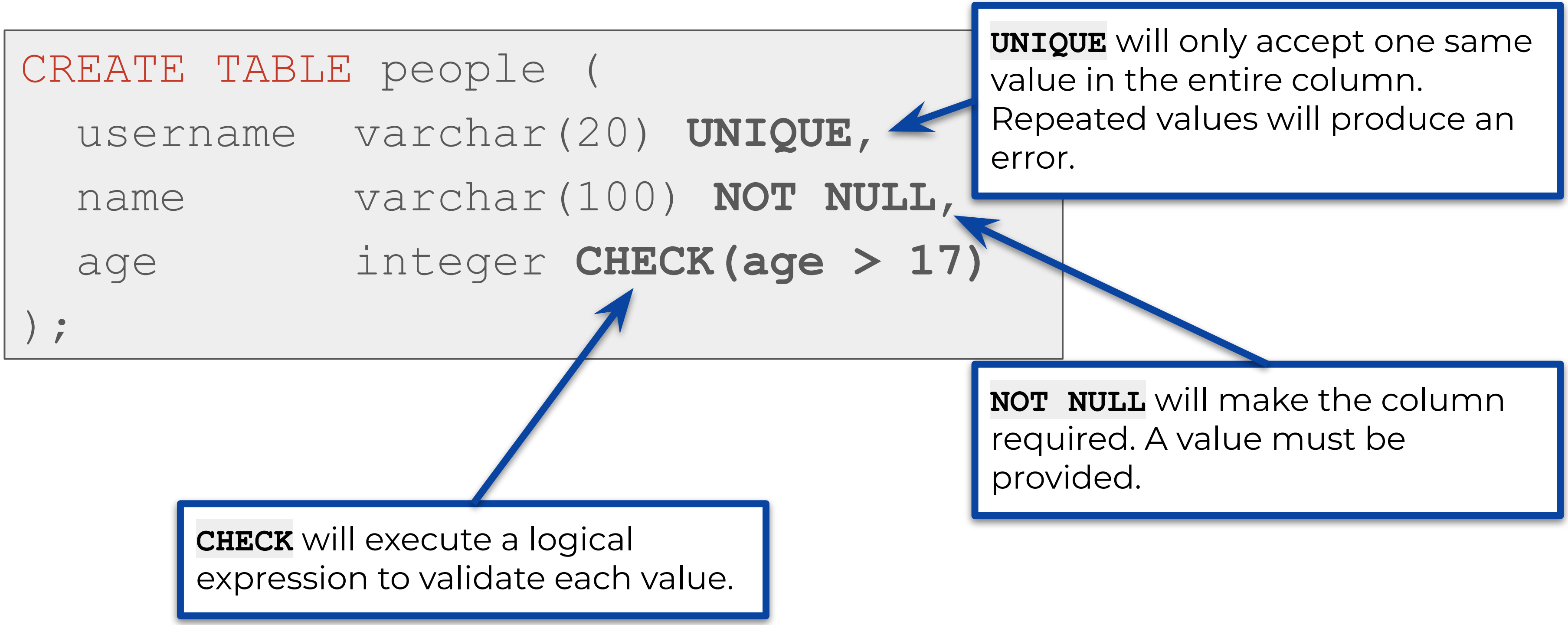
Column Constraints

Constraints are a basic form of validation.

They are used to define some rules any value in a column should follow.

If the value that is being inserted does not match the rules of the column, the engine produces an error.

Column Constraints



```
CREATE TABLE people (
   username   varchar(20) UNIQUE,
   name       varchar(100) NOT NULL,
   age        integer CHECK(age > 17)
);
```

UNIQUE will only accept one same value in the entire column. Repeated values will produce an error.

NOT NULL will make the column required. A value must be provided.

CHECK will execute a logical expression to validate each value.

We learned ...

- That PostgreSQL has a variety of types, including booleans and a variety of integer and text types.
- That booleans can be defined in many ways: true/false, yes/no, on/off and 1/0.
- That there are three types of integers that will use more or less storage space.
- That there are exact and inexact decimal types .
- That exact types are slow in performance as compared to inexact types.
- That all data types allow, by default, an additional state named **`NULL`**, which means it holds no value.
- That we can enforce different constraints on the columns.

Keys

What are Keys?

Keys are columns in a table whose values can be used to **uniquely identify** a row in the same or another table.

One may need to do an operation on any single row in a table, so there has to be a way to identify that row.

Primary Keys

- They are the columns in a table that can be used to uniquely identify any record **on that same table**.
- The values in that column **must be unique**. No two different rows may have the same value in that column.
- Although PostgreSQL does not enforce it, almost all tables should have a primary key.

Primary Keys

Any type can be set as a primary key.

```
CREATE TABLE people (
  full_name    varchar(150) PRIMARY KEY,
  description  text
);
```

This example assumes no two people in the database will have the same full name.

If that is true, this is called a **natural primary key**.

Natural vs. Artificial Primary Keys

Natural primary keys are those attributes in our user data set that can be used to identify a row (for instance, the social security number).

Often, the data does not have such combination of fields, then we have to create a **surrogate primary key**.

```
CREATE TABLE people (
  id            serial PRIMARY KEY,
  ...
);
```

Multi-Column Primary Keys

Primary keys can be declared
on multiple columns at once.

```
CREATE TABLE city (
  name           varchar(30),
  region         varchar(30),
  country        varchar(30),
  PRIMARY KEY(name, region, country)
);
```

Foreign Keys

- They are the columns in a table that can be used to uniquely identify any record **on a different table**.
- The values in that column **are not unique**. They should refer to a column in a different table where values are unique, usually the primary key in that table.
- These keys are used to define relationships between tables.

Foreign Keys

```
CREATE TABLE friends (
  id    serial,
  name  varchar(100)
);

CREATE TABLE message (
  id         serial PRIMARY KEY,
  friend_id  integer REFERENCES friends(id),
  text       text
);
```

Foreign Keys

```
CREATE TABLE friends (
  id     serial PRIMARY KEY,
  name   varchar(100)
);

CREATE TABLE message (
  id          serial PRIMARY KEY,
  friend_id   integer REFERENCES friends~~(id)~~,
  text        text
);
```

If the target column is declared as primary key of the table, that column is not required in the foreign key definition.

Populating Foreign Keys

```
INSERT INTO message(friend_id, text)
VALUES(10, 'How are you doing?');
```

```
=# INSERT INTO message(friend_id, text) VALUES(10, 'How are you doing?');
ERROR:  insert or update on table "message" violates foreign key constraint
"message_friend_id_fkey"
DETAIL:  Key (friend_id)=(10) is not present in table "friends".

=# INSERT INTO message(friend_id, text) VALUES(1, 'How are you doing?');
INSERT 0 1
```

Querying Related Tables

```sql
SELECT friends.name, message.text
FROM friends, message
WHERE friends.id = message.friend_id;
```

```
=# SELECT friends.name, message.text FROM friends, message WHERE friends.id =
message.friend_id;
    name     |          text
-------------+----------------------
 Lisa Klepp | How are you doing?
(1 row)
```

Deleting Related Rows

```
DELETE FROM friends WHERE id = 1;
```

```
=# DELETE FROM friends WHERE id = 1;
ERROR:  update or delete on table "friends" violates foreign key constraint
"message_friend_id_fkey" on table "message"
DETAIL:  Key (id)=(1) is still referenced from table "message".
```

Deleting Related Rows: On Delete

```
CREATE TABLE message (
  id          serial    PRIMARY KEY,
  friend_id   integer   REFERENCES friends
                        ON DELETE SET NULL,
  text        text
);
```

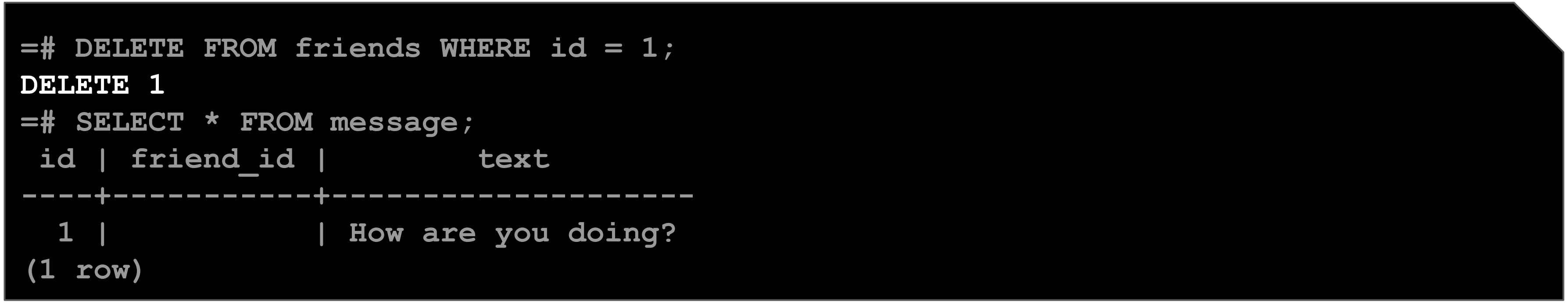
The two most common modes for **`ON DELETE`** are **`SET NULL`** and **`CASCADE`**.

`SET NULL` will set the referencing value to **`NULL`**.

`CASCADE` will delete the referencing row.

Deleting Related Rows with SET NULL

```
DELETE FROM friends WHERE id = 1;
```



```
=# DELETE FROM friends WHERE id = 1;
DELETE 1
=# SELECT * FROM message;
 id | friend_id |        text
----+-----------+----------------------
  1 |           | How are you doing?
(1 row)
```

Deleting Related Rows with CASCADE

```
DELETE FROM friends WHERE id = 1;
```

```
=# DELETE FROM friends WHERE id = 1;
DELETE 1
=# SELECT * FROM message;
 id | friend_id | text
----+-----------+------
(0 rows)
```

We learned …

- That every table must have a combination of columns that can be used to uniquely identify a row.
- That primary keys are unique columns to identify each row.
- That foreign keys are used to reference the primary keys in different tables.
- That these keys are used to define relationships between tables in the database.
- That we can control what happens when a row in a table is deleted and there are rows in another table referring to the missing primary key.

THANK YOU

Contact Details
DCI Digital Career Institute gGmbH

Digital Career Institute