

# Lecture 12: Working with Dates and Times

## DATA 351: Data Management with SQL

Lucas P. Cordova, Ph.D.

2026-03-19

This lecture covers working with dates, times, and time zones in PostgreSQL. We explore date/time data types, extraction functions, time zone handling, date math, and interval calculations. We then apply these concepts to real-world datasets: NYC yellow taxi trip data and Amtrak train schedules. Based on Chapter 12 of Practical SQL, 2nd Edition.

### Table of contents

|           |  |           |
|-----------|--|-----------|
| <b>1</b>  | <b>Part 1: Setting Up</b>                          | <b>2</b>  |
| <b>2</b>  | <b>Part 2: Date and Time Data Types</b>            | <b>4</b>  |
| <b>3</b>  | <b>Part 3: Extracting Date and Time Components</b> | <b>5</b>  |
| <b>4</b>  | <b>Part 4: Creating Dates and Times</b>            | <b>7</b>  |
| <b>5</b>  | <b>Part 5: Time Zones</b>                          | <b>8</b>  |
| <b>6</b>  | <b>Part 6: Date and Time Math</b>                  | <b>12</b> |
| <b>7</b>  | <b>Part 7: NYC Taxi Trip Analysis</b>              | <b>13</b> |
| <b>8</b>  | <b>Part 8: Amtrak Train Trips</b>                  | <b>16</b> |
| <b>9</b>  | <b>Part 9: Putting It All Together</b>             | <b>19</b> |
| <b>10</b> | <b>Part 10: What We Learned</b>                    | <b>21</b> |

# 1 Part 1: Setting Up

Before we start manipulating time itself, we need to load some data.

## 1.1 Loading the Chapter Database

### 1.1.1 Step 1: Create the Database

Open a terminal and create a fresh database for this lecture:

Using `psql`'s `createdb` command:

```
1 createdb -U postgres dates_and_times;
```

OR

Using Beekeeper Studio by connecting to your localhost server and clicking (+).

OR

Using pgAdmin by connecting to your localhost server and right-clicking on the server -> Create -> Database.

### 1.1.2 Step 2: Load the Taxi Data

We need to create the NYC taxi trips table and import the CSV. Run the following to create the table:

```
1 CREATE TABLE nyc_yellow_taxi_trips (  
2     trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,  
3     vendor_id text NOT NULL,  
4     tpep_pickup_datetime timestamptz NOT NULL,  
5     tpep_dropoff_datetime timestamptz NOT NULL,  
6     passenger_count integer NOT NULL,  
7     trip_distance numeric(8,2) NOT NULL,  
8     pickup_longitude numeric(18,15) NOT NULL,  
9     pickup_latitude numeric(18,15) NOT NULL,  
10    rate_code_id text NOT NULL,  
11    store_and_fwd_flag text NOT NULL,  
12    dropoff_longitude numeric(18,15) NOT NULL,  
13    dropoff_latitude numeric(18,15) NOT NULL,  
14    payment_type text NOT NULL,  
15    fare_amount numeric(9,2) NOT NULL,  
16    extra numeric(9,2) NOT NULL,
```

```

17     mta_tax numeric(5,2) NOT NULL,
18     tip_amount numeric(9,2) NOT NULL,
19     tolls_amount numeric(9,2) NOT NULL,
20     improvement_surcharge numeric(9,2) NOT NULL,
21     total_amount numeric(9,2) NOT NULL
22 );

```

### 1.1.3 Step 3: Import the CSV

Use whichever method you prefer:

#### Beekeeper Studio or pgAdmin:

*Change the path to the CSV file on your computer.*

```

1  copy nyc_yellow_taxi_trips (vendor_id, tpep_pickup_datetime,
2     tpep_dropoff_datetime, passenger_count, trip_distance,
3     pickup_longitude, pickup_latitude, rate_code_id,
4     store_and_fwd_flag, dropoff_longitude, dropoff_latitude,
5     payment_type, fare_amount, extra, mta_tax, tip_amount,
6     tolls_amount, improvement_surcharge, total_amount)
7  FROM '/path/to/nyc_yellow_taxi_trips.csv'
8  WITH (FORMAT CSV, HEADER);

```

**Beekeeper Studio:** Right-click the table, select Import CSV, browse to `nyc_yellow_taxi_trips.csv`.

Replace `/path/to/` with the actual path to your CSV. On macOS, drag the file into the terminal to paste the full path.

### 1.1.4 Step 4: Create an Index and Verify

```

1  CREATE INDEX tpep_pickup_idx
2  ON nyc_yellow_taxi_trips (tpep_pickup_datetime);
3
4  -- Verify: you should see 368,774 rows
5  SELECT count(*) FROM nyc_yellow_taxi_trips;

```

### 1.1.5 Step 5: Set the Time Zone

The taxi data is from New York City, so let's make sure we are all seeing the same timestamps:

```

1  SET TIME ZONE 'US/Eastern';

```

This is a **session-level** setting. It lasts until you disconnect. Your data on the server is unchanged.

## 2 Part 2: Date and Time Data Types

Time to talk about time. (That sentence hurt to write.)

### 2.1 The Four Types

#### 2.1.1 PostgreSQL Date/Time Types

PostgreSQL gives us four data types for temporal data:

| Type   | Stores                  | Example                 |
|--|-------------------------|-------------------------|
| <code>timestamp with time zone (timestampz)</code> | Date + time + time zone | 2022-12-01 18:37:12 EST |
| <code>date</code>                                  | Date only               | 2022-12-01              |
| <code>time</code>                                  | Time only               | 18:37:12                |
| <code>interval</code>                              | A duration              | 2 days 3 hours          |

The first three are **datetime types**. The fourth, `interval`, represents a length of time rather than a specific moment.

#### 2.1.2 Which One Should You Use?

Almost always: `timestampz`.

A timestamp without a time zone is like an address without a city. Sure, “123 Main Street” means something, but *where?*

`date` is fine when you genuinely only care about the calendar date (birthdays, holidays). `time` alone is almost never useful because a time without a date or zone is meaningless.

And `interval`? You do not create columns of type `interval` very often. It shows up as the *result* of subtracting two timestamps. Think of it as the answer to “how long?” rather than “when?”

### 2.1.3 Quick Quiz: Pick the Type

What data type would you use for each of these?

1. When a customer placed an order (from a global e-commerce site)
2. A person's date of birth
3. How long a movie is
4. The daily closing time of a store

...

**Answers:**

1. `timestampz` (you need to know *where* on Earth that order happened)
2. `date` (nobody was born at 3:47 PM UTC)
3. `interval` (e.g., '2 hours 23 minutes')
4. Trick question. `time` *seems* right, but without a date, you cannot handle daylight saving changes. In practice, store this as text or use application logic.

## 3 Part 3: Extracting Date and Time Components

Sometimes you do not want the whole timestamp. You just want the year, or the hour, or the day of the week.

### 3.1 `date_part()` and `extract()`

#### 3.1.1 Pulling Apart a Timestamp

The `date_part()` function extracts a single component from a date or timestamp:

```
1 SELECT
2     date_part('year',    '2022-12-01 18:37:12 EST'::timestampz) AS year,
3     date_part('month',  '2022-12-01 18:37:12 EST'::timestampz) AS month,
4     date_part('day',    '2022-12-01 18:37:12 EST'::timestampz) AS day,
5     date_part('hour',   '2022-12-01 18:37:12 EST'::timestampz) AS hour,
6     date_part('minute', '2022-12-01 18:37:12 EST'::timestampz) AS minute,
7     date_part('seconds', '2022-12-01 18:37:12 EST'::timestampz) AS seconds;
```

**Run this now.** Your hour value might differ from your neighbor's. Why?

...

Because your PostgreSQL server converts the timestamp to *your* session time zone. EST is UTC-5. If your server is set to Pacific time (UTC-8), the hour shows as 15 instead of 18. The underlying moment in time is identical.

### 3.1.2 More Components

`date_part()` can extract more than the obvious pieces:

```
1 SELECT
2     date_part('timezone_hour', '2022-12-01 18:37:12 EST'::timestampz) AS tz,
3     date_part('week',         '2022-12-01 18:37:12 EST'::timestampz) AS week,
4     date_part('quarter',      '2022-12-01 18:37:12 EST'::timestampz) AS quarter,
5     date_part('epoch',        '2022-12-01 18:37:12 EST'::timestampz) AS epoch;
```

| Component                  | What It Returns                            |
|----------------------------|--|
| <code>timezone_hour</code> | UTC offset in hours (e.g., -5 for EST)     |
| <code>week</code>          | ISO 8601 week number (1-53)                |
| <code>quarter</code>       | Quarter of the year (1-4)                  |
| <code>epoch</code>         | Seconds since January 1, 1970 00:00:00 UTC |

**Epoch** is how computers *actually* think about time. Every timestamp is secretly just a big number counting seconds from 1970. Everything else is a human convenience.

### 3.1.3 The SQL Standard Alternative: `extract()`

The SQL standard offers `extract()`, which does the same thing with slightly different syntax:

```
1 SELECT extract(year FROM '2022-12-01 18:37:12 EST'::timestampz) AS year;
```

Note: no quotes around the component name, and `FROM` instead of a comma. Both `date_part()` and `extract()` return the same result. Use whichever your team prefers.

### 3.1.4 Try It: Your Turn

Write a query that extracts the **day of the week** (`dow`) and the **hour** from the current timestamp.

Hint: `date_part('dow', ...)` returns 0 for Sunday through 6 for Saturday.

```
1 -- Your query here
```

```
...
1 SELECT
2     date_part('dow', current_timestamp) AS day_of_week,
3     date_part('hour', current_timestamp) AS current_hour;
```

## 4 Part 4: Creating Dates and Times

Sometimes your data arrives in pieces: year in one column, month in another, day in a third. PostgreSQL can reassemble them.

### 4.1 Making Datetimes

#### 4.1.1 Building from Components

PostgreSQL provides three constructor functions:

```
1 -- Make a date
2 SELECT make_date(2022, 2, 22);
3
4 -- Make a time (no time zone)
5 SELECT make_time(18, 4, 30.3);
6
7 -- Make a full timestamp with time zone
8 SELECT make_timestamptz(2022, 2, 22, 18, 4, 30.3, 'Europe/Lisbon');
```

That last one is the most useful. Notice the time zone argument is a string. The output will be converted to your session's time zone.

**Run the `make_timestamptz` query now.** What time does it show?

```
...
```

If your session is set to `US/Eastern`, you should see `2022-02-22 13:04:30.3-05`. Lisbon is at UTC+0, Eastern in February is UTC-5, so 6:04 PM in Lisbon is 1:04 PM in New York.

## 4.1.2 Getting the Current Date and Time

PostgreSQL gives you several ways to ask “what time is it?”:

```
1 SELECT
2     current_timestamp,    -- timestamp with time zone (SQL standard)
3     localtimestamp,      -- timestamp WITHOUT time zone (avoid this)
4     current_date,        -- just the date
5     current_time,        -- just the time with zone
6     localtime,          -- just the time without zone
7     now();               -- PostgreSQL shorthand for current_timestamp
```

**Important:** All of these return the time at the *start* of the query. If your query takes 10 seconds, every row gets the *same* timestamp.

## 4.1.3 current\_timestamp vs. clock\_timestamp()

What if you want the *actual* clock time as each row is processed?

```
1 CREATE TABLE current_time_example (
2     time_id integer GENERATED ALWAYS AS IDENTITY,
3     current_timestamp_col timestamptz,
4     clock_timestamp_col timestamptz
5 );
6
7 INSERT INTO current_time_example
8     (current_timestamp_col, clock_timestamp_col)
9     (SELECT current_timestamp,
10          clock_timestamp()
11     FROM generate_series(1,1000));
12
13 SELECT * FROM current_time_example;
```

**Run this and look at the results.** The `current_timestamp_col` is identical for all 1,000 rows. The `clock_timestamp_col` increases slightly with each row.

When would this matter? Imagine logging the exact processing time of each row during a large data migration. `current_timestamp` would lie to you. `clock_timestamp()` tells the truth.

## 5 Part 5: Time Zones

This is where dates and times get *spicy*.

## 5.1 Understanding Time Zones

### 5.1.1 Why Time Zones Matter

Quick thought experiment: a server in Oregon logs an event at 2026-03-15 14:00:00. A server in New York logs an event at 2026-03-15 17:00:00. Which happened first?

...

They happened at the **exact same moment**. Oregon is UTC-7 in March, New York is UTC-4. Both are 21:00 UTC.

Without time zone information, you would think the Oregon event happened 3 hours earlier. This is how bugs are born, and they are the worst kind of bugs because they look correct until they do not.

### 5.1.2 Checking Your Time Zone

```
1 SHOW timezone;
2
3 -- Or equivalently:
4 SELECT current_setting('timezone');
```

Run this. What does yours say? If you followed the setup instructions, it should say **US/Eastern** because we set it earlier.

### 5.1.3 Browsing Available Time Zones

PostgreSQL ships with two system tables full of time zone information:

```
1 -- Abbreviations (short list)
2 SELECT * FROM pg_timezone_abbrevs ORDER BY abbrev;
3
4 -- Full named zones (long list)
5 SELECT * FROM pg_timezone_names ORDER BY name;
```

You can filter them:

```
1 SELECT * FROM pg_timezone_names
2 WHERE name LIKE 'US%'
3 ORDER BY name;
```

### 5.1.4 Try It: Find a Time Zone

Write a query to find all time zones in Asia. How many are there?

```
1 -- Your query here
. . .
1 SELECT count(*) FROM pg_timezone_names
2 WHERE name LIKE 'Asia/%';
```

(There are a lot. Asia is a big continent with a complicated relationship with time zones.)

## 5.2 Changing and Converting Time Zones

### 5.2.1 SET TIME ZONE

You can change your session's time zone at any time:

```
1 SET TIME ZONE 'US/Pacific';
```

This changes how timestamps are *displayed*, not how they are *stored*. Internally, PostgreSQL always stores `timestampz` values as UTC.

### 5.2.2 The Great Demonstration

Let's watch the same moment in time change its clothes:

```
1 CREATE TABLE time_zone_test (
2     test_date timestampz
3 );
4
5 INSERT INTO time_zone_test VALUES ('2023-01-01 4:00');
```

Now watch:

```
1 SET TIME ZONE 'US/Pacific';
2 SELECT test_date FROM time_zone_test;
3 -- Shows: 2023-01-01 04:00:00-08
4
5 SET TIME ZONE 'US/Eastern';
6 SELECT test_date FROM time_zone_test;
7 -- Shows: 2023-01-01 07:00:00-05
```

Same row, same data on disk, different display. Pacific is 3 hours behind Eastern, so 4 AM Pacific = 7 AM Eastern. PostgreSQL handles this automatically.

### 5.2.3 AT TIME ZONE: One-Off Conversions

What if you do not want to change your whole session, just peek at one value in a different time zone?

```
1 SELECT test_date AT TIME ZONE 'Asia/Seoul'
2 FROM time_zone_test;
3 -- Shows: 2023-01-01 21:00:00
```

4 AM Pacific on January 1st is 9 PM the same day in Seoul. New Year's Eve in Portland, New Year's Day in Korea.

**Quirk alert:** When you use `AT TIME ZONE` on a `timestamp`, the output is a `timestamp without time zone` (and vice versa). PostgreSQL strips the zone because you already specified which zone you want.

### 5.2.4 Try It: World Clocks

It is midnight on New Year's Day 2030 in New York ('2030-01-01 00:00:00 US/Eastern').

Write a query that shows what time it is at that exact moment in:

- London (Europe/London)
- Tokyo (Asia/Tokyo)
- Sydney (Australia/Sydney)

```
1 -- Your query here
...
1 SELECT
2     '2030-01-01 00:00:00 US/Eastern'::timestampz
3     AT TIME ZONE 'Europe/London' AS london,
4     '2030-01-01 00:00:00 US/Eastern'::timestampz
5     AT TIME ZONE 'Asia/Tokyo' AS tokyo,
6     '2030-01-01 00:00:00 US/Eastern'::timestampz
7     AT TIME ZONE 'Australia/Sydney' AS sydney;
```

London: 5 AM. Tokyo: 2 PM. Sydney: 4 PM. When New Yorkers are watching the ball drop, Sydney already had their fireworks 16 hours ago.

## 6 Part 6: Date and Time Math

You can do arithmetic with dates and times, and it actually makes sense.

### 6.1 Calculations with Dates

#### 6.1.1 Subtracting Dates

Subtract one date from another and you get an integer (days between them):

```
1 SELECT '1929-09-30'::date - '1929-09-27'::date;
```

Result: 3. Three days between September 27 and September 30, 1929. (That was right before the stock market crash, incidentally. Bad week.)

#### 6.1.2 Adding Intervals to Dates

Add an interval to a date and you get a new timestamp:

```
1 SELECT '1929-09-30'::date + '5 years'::interval;
```

Result: 1934-09-30 00:00:00. PostgreSQL knows about leap years, month lengths, and all the calendar quirks. You do not have to think about whether February has 28 or 29 days.

#### 6.1.3 Subtracting Timestamps

Subtract two timestamps and you get an interval:

```
1 SELECT
2     '2026-03-19 17:00:00 US/Eastern'::timestampz -
3     '2026-03-19 09:00:00 US/Eastern'::timestampz
4     AS work_day_length;
```

Result: 08:00:00. An eight-hour workday. (In theory.)

### 6.1.4 Try It: How Old Is PostgreSQL?

PostgreSQL's first release was on July 8, 1996. Write a query that calculates how many days ago that was.

```
1 -- Your query here
. . .
1 SELECT current_date - '1996-07-08'::date AS days_since_postgres;
```

That is a lot of days. PostgreSQL has been around longer than some of you have been alive.

## 7 Part 7: NYC Taxi Trip Analysis

Time to put all of this to work on real data. We have 368,774 taxi rides from June 1, 2016, in New York City.

### 7.1 Patterns in Pickup Times

#### 7.1.1 The Busiest Hours

When do New Yorkers take the most taxi rides? Let's find out:

```
1 SET TIME ZONE 'US/Eastern';
2
3 SELECT
4     date_part('hour', tpep_pickup_datetime) AS trip_hour,
5     count(*)
6 FROM nyc_yellow_taxi_trips
7 GROUP BY trip_hour
8 ORDER BY trip_hour;
```

**Run this now.** You should get 24 rows, one per hour.

Look at the results. When is the busiest time? When is the slowest?

. . .

The busiest hours are in the evening (6 PM to 10 PM). The slowest are in the early morning (2 AM to 5 AM). No surprises there: nobody is hailing a cab at 3 AM on a Wednesday. (Well, almost nobody.)



**Run this.** Which hour has the longest median trip time? Which has the shortest?

...

The longest median trips are midday (around 1 PM: ~15 minutes). The shortest are early morning (~5 AM: under 8 minutes). Less traffic at 5 AM means you get where you are going faster. At 1 PM, you are fighting for every block.

### 7.2.2 Try It: The Longest Rides

Write a query to find the 10 longest taxi rides by duration. Include the pickup time, dropoff time, the calculated duration, and the trip distance.

Do any of the results look suspicious?

```
1 -- Your query here
```

...

```
1 SELECT
2     tpep_pickup_datetime,
3     tpep_dropoff_datetime,
4     tpep_dropoff_datetime - tpep_pickup_datetime AS duration,
5     trip_distance
6 FROM nyc_yellow_taxi_trips
7 ORDER BY duration DESC
8 LIMIT 10;
```

You will probably see some rides lasting 20+ hours with a trip distance of 0. Those are almost certainly errors: the meter was left running, or the data was entered incorrectly. Real-world data is messy. Always sanity-check your results.

### 7.2.3 Try It: Average Fare by Hour

Write a query that shows the average `total_amount` (fare) for each hour of the day. Which hour has the highest average fare?

```
1 -- Your query here
```

...

```

1 SELECT
2     date_part('hour', tpep_pickup_datetime) AS trip_hour,
3     round(avg(total_amount), 2) AS avg_fare
4 FROM nyc_yellow_taxi_trips
5 GROUP BY trip_hour
6 ORDER BY avg_fare DESC;

```

The early morning hours (4-5 AM) often have the highest average fares. Fewer but longer rides: probably airport runs and longer-distance trips when there is no traffic.

## 8 Part 8: Amtrak Train Trips

Let's switch from taxis to trains. We are going to track a cross-country Amtrak trip through four time zones.

### 8.1 Building the Train Data

#### 8.1.1 Creating and Loading the Table

```

1 CREATE TABLE train_rides (
2     trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
3     segment text NOT NULL,
4     departure timestampz NOT NULL,
5     arrival timestampz NOT NULL
6 );
7
8 INSERT INTO train_rides (segment, departure, arrival)
9 VALUES
10    ('Chicago to New York',
11     '2020-11-13 21:30 CST', '2020-11-14 18:23 EST'),
12    ('New York to New Orleans',
13     '2020-11-15 14:15 EST', '2020-11-16 19:32 CST'),
14    ('New Orleans to Los Angeles',
15     '2020-11-17 13:45 CST', '2020-11-18 9:00 PST'),
16    ('Los Angeles to San Francisco',
17     '2020-11-19 10:10 PST', '2020-11-19 21:24 PST'),
18    ('San Francisco to Denver',
19     '2020-11-20 9:10 PST', '2020-11-21 18:38 MST'),
20    ('Denver to Chicago',
21     '2020-11-22 19:10 MST', '2020-11-23 14:50 CST');

```

**Run this now.**

Notice how each timestamp includes the time zone of the city. Chicago departs in CST, arrives in New York in EST. This is critical for accurate duration calculations.

### 8.1.2 Viewing the Data

```
1 SET TIME ZONE 'US/Central';
2 SELECT * FROM train_rides;
```

All timestamps now display in Central time. The underlying data has not changed. PostgreSQL just shows everything through a Central time lens.

## 8.2 Calculating Trip Durations

### 8.2.1 Segment Duration

How long is each leg of the journey?

```
1 SELECT segment,
2        to_char(departure, 'YYYY-MM-DD HH12:MI a.m. TZ') AS departure,
3        arrival - departure AS segment_duration
4 FROM train_rides;
```

Two things to notice here:

1. `to_char()` formats the timestamp into a human-friendly string. `HH12` gives 12-hour time, `a.m.` adds the AM/PM indicator, `TZ` shows the time zone abbreviation.
2. **Subtracting timestamps** gives us an `interval`. PostgreSQL correctly handles the time zone differences. Chicago to New York crosses a time zone boundary, and the math accounts for it.

### 8.2.2 Try It: Longest Segment

Which segment of the trip takes the longest?

```
1 -- Your query here
...

```

```

1 SELECT segment,
2     arrival - departure AS segment_duration
3 FROM train_rides
4 ORDER BY segment_duration DESC
5 LIMIT 1;

```

San Francisco to Denver: over 32 hours. That is a long time to be on a train. Bring a book. Or three.

## 8.3 Cumulative Trip Duration

### 8.3.1 Running Total with Window Functions

How long has the entire trip taken so far after each segment? This is a job for a **window function**:

```

1 SELECT segment,
2     arrival - departure AS segment_duration,
3     sum(arrival - departure) OVER (ORDER BY trip_id) AS cume_duration
4 FROM train_rides;

```

**Run this.** Look at the `cume_duration` column. Something looks weird.

...

The output says things like `2 days 85:47:00`. That is technically correct (85 hours is, in fact, 85 hours) but confusing. PostgreSQL sums the days and the hours separately and does not roll them up.

### 8.3.2 Fixing It with `justify_interval()`

The `justify_interval()` function converts overflow hours into days and overflow days into months:

```

1 SELECT segment,
2     arrival - departure AS segment_duration,
3     justify_interval(sum(arrival - departure)
4                     OVER (ORDER BY trip_id)) AS cume_duration
5 FROM train_rides;

```

Now the final row shows `5 days 13:47:00`. That is how long the entire cross-country trip takes. Five and a half days on a train. The scenery better be incredible.

### 8.3.3 Try It: Format the Output

Rewrite the cumulative duration query so that the departure column is formatted as Mon DD, YYYY HH12:MI AM TZ using `to_char()`.

Hint: The format string is 'Mon DD, YYYY HH12:MI AM TZ'.

```
1 -- Your query here
2
3 . . .
4
5 1 SELECT segment,
6     to_char(departure, 'Mon DD, YYYY HH12:MI AM TZ') AS departure,
7     arrival - departure AS segment_duration,
8     justify_interval(sum(arrival - departure)
9                       OVER (ORDER BY trip_id)) AS cume_duration
10 FROM train_rides;
```

## 9 Part 9: Putting It All Together

Let's combine several concepts from today into more complex queries.

### 9.1 Combined Exercises

#### 9.1.1 Try It: Taxi Trips by Day of Week

Using the NYC taxi data, write a query that counts trips by **day of the week**. Use `date_part('dow', ...)` and a CASE statement to show day names instead of numbers.

Remember: `dow` returns 0 = Sunday through 6 = Saturday.

(This dataset only has one day, but the query pattern works on multi-day datasets.)

```
1 -- Your query here
2
3 . . .
4
5 1 SELECT
6     CASE date_part('dow', tpep_pickup_datetime)
7     WHEN 0 THEN 'Sunday'
8     WHEN 1 THEN 'Monday'
9     WHEN 2 THEN 'Tuesday'
10    WHEN 3 THEN 'Wednesday'
11    WHEN 4 THEN 'Thursday'
```

```

8         WHEN 5 THEN 'Friday'
9         WHEN 6 THEN 'Saturday'
10    END AS day_name,
11    count(*) AS num_trips
12 FROM nyc_yellow_taxi_trips
13 GROUP BY day_name
14 ORDER BY num_trips DESC;

```

Since this dataset is from June 1, 2016 (a Wednesday), you will only see one row. But now you know the pattern for when you have months of data.

### 9.1.2 Try It: Time Zone Conversion

Write a query that shows the taxi pickup times converted to Asia/Tokyo time. Just show the first 5 rows.

What day is it in Tokyo when it is Wednesday evening in New York?

```

1  -- Your query here
   ...
1  SELECT
2     tpep_pickup_datetime AS nyc_time,
3     tpep_pickup_datetime AT TIME ZONE 'Asia/Tokyo' AS tokyo_time
4 FROM nyc_yellow_taxi_trips
5 ORDER BY tpep_pickup_datetime
6 LIMIT 5;

```

Tokyo is 13 hours ahead of New York in summer. A 9 PM Wednesday pickup in NYC is 10 AM Thursday in Tokyo.

### 9.1.3 Try It: Trip Duration Percentiles

Write a query that calculates the **25th**, **50th (median)**, and **75th percentile** of trip duration for the entire dataset. Use `percentile_cont()`.

```

1  -- Your query here
   ...

```

```

1 SELECT
2     percentile_cont(0.25)
3         WITHIN GROUP (ORDER BY tpep_dropoff_datetime - tpep_pickup_datetime)
4         AS p25,
5     percentile_cont(0.50)
6         WITHIN GROUP (ORDER BY tpep_dropoff_datetime - tpep_pickup_datetime)
7         AS median,
8     percentile_cont(0.75)
9         WITHIN GROUP (ORDER BY tpep_dropoff_datetime - tpep_pickup_datetime)
10        AS p75
11 FROM nyc_yellow_taxi_trips;

```

The interquartile range (P75 - P25) tells you how spread out “typical” trip durations are.

## 10 Part 10: What We Learned

### 10.1 Summary

#### 10.1.1 Key Concepts

| Concept  | What It Does                                  |
|--|---|
| <code>timestampz</code>                                      | The go-to type for storing moments in time    |
| <code>date</code>  | Date only (no time, no zone)                  |
| <code>interval</code>  | A duration, not a point in time               |
| <code>date_part()</code> / <code>extract()</code>            | Pull year, month, hour, etc. from a timestamp |
| <code>make_date()</code> /<br><code>make_timestampz()</code> | Build dates from components                   |
| <code>current_timestamp</code> / <code>now()</code>          | The time when the query started               |
| <code>clock_timestamp()</code>                               | The actual clock time (changes per row)       |
| <code>SET TIME ZONE</code>                                   | Change display for your session               |
| <code>AT TIME ZONE</code>                                    | One-off conversion to another zone            |
| <code>to_char()</code>                                       | Format timestamps as readable strings         |
| <code>justify_interval()</code>                              | Clean up messy interval output                |

#### 10.1.2 The Big Ideas

1. **Always use `timestampz`.** A timestamp without a zone is a bug waiting to happen.
2. **Time zones change the display, not the data.** PostgreSQL stores everything as UTC internally.

3. **Subtracting timestamps gives you intervals.** That is how you calculate durations.
4. **Real-world data is messy.** Always sanity-check your results. A 24-hour taxi ride with zero distance is not a real ride.
5. **date\_part() is your Swiss Army knife.** Need to group by hour? By month? By quarter? It handles all of them.

### 10.1.3 References

1. DeBarros, A. (2022). *Practical SQL: A Beginner's Guide to Storytelling with Data* (2nd ed.). No Starch Press. Chapter 12.
2. PostgreSQL Date/Time Functions Documentation: <https://www.postgresql.org/docs/current/functions-datetime.html>
3. PostgreSQL Formatting Functions: <https://www.postgresql.org/docs/current/functions-formatting.html>
4. NYC Taxi and Limousine Commission Trip Data: <https://www.nyc.gov/site/tlc/about/tlc-trip-record-data.page>
5. ISO 8601 Date and Time Standard: [https://en.wikipedia.org/wiki/ISO\\_8601](https://en.wikipedia.org/wiki/ISO_8601)