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

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:

```
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:

```
CREATE TABLE nyc_yellow_taxi_trips (
    trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    vendor_id text NOT NULL,
    tpep_pickup_datetime timestamptz NOT NULL,
    tpep_dropoff_datetime timestamptz NOT NULL,
    passenger_count integer NOT NULL,
    trip_distance numeric(8,2) NOT NULL,
    pickup_longitude numeric(18,15) NOT NULL,
    pickup_latitude numeric(18,15) NOT NULL,
    rate_code_id text NOT NULL,
    store_and_fwd_flag text NOT NULL,
    dropoff_longitude numeric(18,15) NOT NULL,
    dropoff_latitude numeric(18,15) NOT NULL,
    payment_type text NOT NULL,
    fare_amount numeric(9,2) NOT NULL,
    extra numeric(9,2) NOT NULL,
```

```
    mta_tax numeric(5,2) NOT NULL,
    tip_amount numeric(9,2) NOT NULL,
    tolls_amount numeric(9,2) NOT NULL,
    improvement_surcharge numeric(9,2) NOT NULL,
    total_amount numeric(9,2) NOT NULL
);
```

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.

```
copy nyc_yellow_taxi_trips (vendor_id, tpep_pickup_datetime,
    tpep_dropoff_datetime, passenger_count, trip_distance,
    pickup_longitude, pickup_latitude, rate_code_id,
    store_and_fwd_flag, dropoff_longitude, dropoff_latitude,
    payment_type, fare_amount, extra, mta_tax, tip_amount,
    tolls_amount, improvement_surcharge, total_amount)
FROM '/path/to/nyc_yellow_taxi_trips.csv'
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

```
CREATE INDEX tpep_pickup_idx
ON nyc_yellow_taxi_trips (tpep_pickup_datetime);

-- Verify: you should see 368,774 rows
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:

```
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
`timestamp with time zone` (`timestamptz`)	Date + time + time zone	`2022-12-01 18:37:12 EST`
`date`	Date only	`2022-12-01`
`time`	Time only	`18:37:12`
`interval`	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: `timestamptz`.

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. `timestamptz` (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:

```
SELECT
    date_part('year',   '2022-12-01 18:37:12 EST'::timestamptz) AS year,
    date_part('month',  '2022-12-01 18:37:12 EST'::timestamptz) AS month,
    date_part('day',    '2022-12-01 18:37:12 EST'::timestamptz) AS day,
    date_part('hour',   '2022-12-01 18:37:12 EST'::timestamptz) AS hour,
    date_part('minute', '2022-12-01 18:37:12 EST'::timestamptz) AS minute,
    date_part('seconds','2022-12-01 18:37:12 EST'::timestamptz) 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:

```
SELECT
    date_part('timezone_hour', '2022-12-01 18:37:12 EST'::timestamptz) AS tz,
    date_part('week',          '2022-12-01 18:37:12 EST'::timestamptz) AS week,
    date_part('quarter',       '2022-12-01 18:37:12 EST'::timestamptz) AS quarter,
    date_part('epoch',         '2022-12-01 18:37:12 EST'::timestamptz) AS epoch;
```

Component	What It Returns
`timezone_hour`	UTC offset in hours (e.g., -5 for EST)
`week`	ISO 8601 week number (1-53)
`quarter`	Quarter of the year (1-4)
`epoch`	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:

```
SELECT extract(year FROM '2022-12-01 18:37:12 EST'::timestamptz) 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.

```
-- Your query here
```

. . .

```
SELECT
    date_part('dow', current_timestamp) AS day_of_week,
    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:

```
-- Make a date
SELECT make_date(2022, 2, 22);

-- Make a time (no time zone)
SELECT make_time(18, 4, 30.3);

-- Make a full timestamp with time zone
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?":

```
SELECT
    current_timestamp,    -- timestamp with time zone (SQL standard)
    localtimestamp,       -- timestamp WITHOUT time zone (avoid this)
    current_date,         -- just the date
    current_time,         -- just the time with zone
    localtime,            -- just the time without zone
    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?

```
CREATE TABLE current_time_example (
    time_id integer GENERATED ALWAYS AS IDENTITY,
    current_timestamp_col timestamptz,
    clock_timestamp_col timestamptz
);

INSERT INTO current_time_example
            (current_timestamp_col, clock_timestamp_col)
    (SELECT current_timestamp,
            clock_timestamp()
     FROM generate_series(1,1000));

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

```
SHOW timezone;

-- Or equivalently:
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:

```
-- Abbreviations (short list)
SELECT * FROM pg_timezone_abbrevs ORDER BY abbrev;

-- Full named zones (long list)
SELECT * FROM pg_timezone_names ORDER BY name;
```

You can filter them:

```
SELECT * FROM pg_timezone_names
WHERE name LIKE 'US%'
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?

```
-- Your query here
```

. . .

```
SELECT count(*) FROM pg_timezone_names
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:

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

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

5.2.2 The Great Demonstration

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

```
CREATE TABLE time_zone_test (
    test_date timestamptz
);

INSERT INTO time_zone_test VALUES ('2023-01-01 4:00');
```

Now watch:

```
SET TIME ZONE 'US/Pacific';
SELECT test_date FROM time_zone_test;
-- Shows: 2023-01-01 04:00:00-08

SET TIME ZONE 'US/Eastern';
SELECT test_date FROM time_zone_test;
-- 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?

```
SELECT test_date AT TIME ZONE 'Asia/Seoul'
FROM time_zone_test;
-- 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 `timestamptz`, 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`)

```
-- Your query here
```

. . .

```
SELECT
    '2030-01-01 00:00:00 US/Eastern'::timestamptz
        AT TIME ZONE 'Europe/London' AS london,
    '2030-01-01 00:00:00 US/Eastern'::timestamptz
        AT TIME ZONE 'Asia/Tokyo' AS tokyo,
    '2030-01-01 00:00:00 US/Eastern'::timestamptz
        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):

```
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:

```
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:

```
SELECT
    '2026-03-19 17:00:00 US/Eastern'::timestamptz -
    '2026-03-19 09:00:00 US/Eastern'::timestamptz
    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.

```
-- Your query here
```

. . .

```
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:

```
SET TIME ZONE 'US/Eastern';

SELECT
    date_part('hour', tpep_pickup_datetime) AS trip_hour,
    count(*)
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
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.)

7.1.2 Try It: Busiest by Count

Write a query to find the **single busiest hour** and how many trips occurred in it.

```
-- Your query here
```

. . .

```
SELECT
    date_part('hour', tpep_pickup_datetime) AS trip_hour,
    count(*) AS num_trips
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
ORDER BY num_trips DESC
LIMIT 1;
```

7.1.3 The Afternoon Dip

If you look at the data carefully, there is a dip around 3-4 PM. Why would taxi ridership drop in the middle of the afternoon?

Could be shift changes for taxi drivers. Could be that people are still at work. Could be that everyone is stuck in traffic anyway so they walk. This is where data raises questions faster than it answers them.

7.2 Trip Duration Analysis

7.2.1 Median Trip Time by Hour

How long do taxi rides take at different times of day? We can calculate the **median** trip duration by subtracting pickup from dropoff:

```
SELECT
    date_part('hour', tpep_pickup_datetime) AS trip_hour,
    percentile_cont(.5)
        WITHIN GROUP (ORDER BY
            tpep_dropoff_datetime - tpep_pickup_datetime) AS median_trip
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
ORDER BY trip_hour;
```

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?

```
-- Your query here
```

. . .

```
SELECT
    tpep_pickup_datetime,
    tpep_dropoff_datetime,
    tpep_dropoff_datetime - tpep_pickup_datetime AS duration,
    trip_distance
FROM nyc_yellow_taxi_trips
ORDER BY duration DESC
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?

```
-- Your query here
```

. . .

```sql
SELECT
    date_part('hour', tpep_pickup_datetime) AS trip_hour,
    round(avg(total_amount), 2) AS avg_fare
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
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

```sql
CREATE TABLE train_rides (
    trip_id bigint GENERATED ALWAYS AS IDENTITY PRIMARY KEY,
    segment text NOT NULL,
    departure timestamptz NOT NULL,
    arrival timestamptz NOT NULL
);

INSERT INTO train_rides (segment, departure, arrival)
VALUES
    ('Chicago to New York',
        '2020-11-13 21:30 CST', '2020-11-14 18:23 EST'),
    ('New York to New Orleans',
        '2020-11-15 14:15 EST', '2020-11-16 19:32 CST'),
    ('New Orleans to Los Angeles',
        '2020-11-17 13:45 CST', '2020-11-18 9:00 PST'),
    ('Los Angeles to San Francisco',
        '2020-11-19 10:10 PST', '2020-11-19 21:24 PST'),
    ('San Francisco to Denver',
        '2020-11-20 9:10 PST',  '2020-11-21 18:38 MST'),
    ('Denver to Chicago',
        '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

```
SET TIME ZONE 'US/Central';
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?

```
SELECT segment,
       to_char(departure, 'YYYY-MM-DD HH12:MI a.m. TZ') AS departure,
       arrival - departure AS segment_duration
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?

```
-- Your query here
```

. . .

```sql
SELECT segment,
       arrival - departure AS segment_duration
FROM train_rides
ORDER BY segment_duration DESC
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**:

```sql
SELECT segment,
       arrival - departure AS segment_duration,
       sum(arrival - departure) OVER (ORDER BY trip_id) AS cume_duration
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:

```sql
SELECT segment,
       arrival - departure AS segment_duration,
       justify_interval(sum(arrival - departure)
                        OVER (ORDER BY trip_id)) AS cume_duration
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'`.

```
-- Your query here
```

. . .

```
SELECT segment,
       to_char(departure, 'Mon DD, YYYY HH12:MI AM TZ') AS departure,
       arrival - departure AS segment_duration,
       justify_interval(sum(arrival - departure)
                        OVER (ORDER BY trip_id)) AS cume_duration
FROM train_rides;
```

9 Part 9: CTEs and Useful PostgreSQL Features

Before we hit the combined exercises, let's cover some tools that make complex temporal queries much easier to write and read.

9.1 Common Table Expressions (CTEs)

9.1.1 What Is a CTE?

A **Common Table Expression** (CTE) is a temporary, named result set that exists only for the duration of a single query. Think of it as a "named subquery" that you define at the top and reference below.

```
WITH daily_stats AS (
    SELECT
        date_part('hour', tpep_pickup_datetime) AS trip_hour,
        count(*) AS num_trips,
        round(avg(total_amount), 2) AS avg_fare
    FROM nyc_yellow_taxi_trips
    GROUP BY trip_hour
)
SELECT
    trip_hour,
    num_trips,
```

```
    avg_fare
FROM daily_stats
WHERE num_trips > 10000
ORDER BY trip_hour;
```

The `WITH` clause defines `daily_stats`. The main `SELECT` then queries it like a regular table. When the query finishes, `daily_stats` disappears.

9.1.2 Why Use CTEs?

Without a CTE, the same query would use a nested subquery:

```
SELECT trip_hour, num_trips, avg_fare
FROM (
    SELECT
        date_part('hour', tpep_pickup_datetime) AS trip_hour,
        count(*) AS num_trips,
        round(avg(total_amount), 2) AS avg_fare
    FROM nyc_yellow_taxi_trips
    GROUP BY trip_hour
) AS daily_stats
WHERE num_trips > 10000
ORDER BY trip_hour;
```

Both produce the same result. But CTEs are:

- **More readable.** The logic flows top to bottom instead of inside out.
- **Reusable.** You can reference the same CTE multiple times in one query.
- **Chainable.** You can define multiple CTEs that build on each other.

9.1.3 Chaining Multiple CTEs

You can define several CTEs separated by commas. Each one can reference the ones defined before it:

```
WITH hourly AS (
    SELECT
        date_part('hour', tpep_pickup_datetime) AS trip_hour,
        count(*) AS num_trips,
        round(avg(total_amount), 2) AS avg_fare,
        round(avg(trip_distance), 2) AS avg_distance
    FROM nyc_yellow_taxi_trips
    GROUP BY trip_hour
```

```
),
ranked AS (
    SELECT
        trip_hour,
        num_trips,
        avg_fare,
        avg_distance,
        rank() OVER (ORDER BY num_trips DESC) AS popularity_rank
    FROM hourly
)
SELECT *
FROM ranked
WHERE popularity_rank <= 5
ORDER BY popularity_rank;
```

Step 1 (`hourly`): aggregate by hour. Step 2 (`ranked`): rank the hours by trip count. Step 3 (main query): filter to the top 5.

Each step is simple. The whole query is complex. That is the power of CTEs.

9.1.4 Try It: Busiest vs. Quietest Hours

Using a CTE, find the 3 busiest and 3 quietest hours by trip count. Combine them into one result using `UNION ALL`.

Hint: Write two queries against the same CTE – one with `ORDER BY num_trips DESC LIMIT 3` and one with `ORDER BY num_trips ASC LIMIT 3` – and union them.

```
-- Your query here
```

. . .

```
WITH hourly AS (
    SELECT
        date_part('hour', tpep_pickup_datetime) AS trip_hour,
        count(*) AS num_trips
    FROM nyc_yellow_taxi_trips
    GROUP BY trip_hour
)
(SELECT trip_hour, num_trips, 'Busiest' AS category
 FROM hourly ORDER BY num_trips DESC LIMIT 3)
UNION ALL
(SELECT trip_hour, num_trips, 'Quietest' AS category
```

```
 FROM hourly ORDER BY num_trips ASC LIMIT 3)
ORDER BY category, num_trips DESC;
```

Notice how the CTE is defined once but used twice. Without a CTE, you would have to write the same `GROUP BY` query in both halves of the `UNION ALL`.

9.2 Useful PostgreSQL Functions

9.2.1 COALESCE: Handling NULLs

`COALESCE` returns the first non-NULL value from its arguments. It is essential whenever you deal with optional data or outer joins:

```
SELECT
    coalesce(passenger_count, 0) AS passengers,
    coalesce(tip_amount, 0) AS tip
FROM nyc_yellow_taxi_trips
LIMIT 5;
```

This is especially useful when combining `generate_series` with `LEFT JOIN` – hours with no trips produce NULL counts, and `COALESCE(count, 0)` turns those into zeros.

9.2.2 ROUND and Numeric Formatting

You have seen `ROUND` already, but here is a reminder of its importance:

```
SELECT
    round(avg(total_amount), 2) AS avg_fare,
    round(avg(trip_distance), 2) AS avg_distance,
    round(avg(
        extract(epoch FROM
            (tpep_dropoff_datetime - tpep_pickup_datetime)
        ) / 60
    ), 1) AS avg_duration_min
FROM nyc_yellow_taxi_trips;
```

Always round your output. Showing `12.345678901234` when `12.35` will do is noise, not precision.

9.2.3 HAVING: Filtering Groups

`WHERE` filters rows *before* aggregation. `HAVING` filters groups *after* aggregation:

```
SELECT
    date_part('hour', tpep_pickup_datetime) AS trip_hour,
    count(*) AS num_trips,
    round(avg(total_amount), 2) AS avg_fare
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
HAVING count(*) > 15000
ORDER BY trip_hour;
```

This returns only hours with more than 15,000 trips. You cannot do this with `WHERE` because the count does not exist until after the `GROUP BY` runs.

9.2.4 Try It: High-Fare Hours

Write a query that finds all hours where the average `total_amount` exceeds $20. Show the hour and the average fare.

```
-- Your query here
```

. . .

```
SELECT
    date_part('hour', tpep_pickup_datetime) AS trip_hour,
    round(avg(total_amount), 2) AS avg_fare
FROM nyc_yellow_taxi_trips
GROUP BY trip_hour
HAVING avg(total_amount) > 20
ORDER BY avg_fare DESC;
```

9.2.5 CTE + Date Functions: A Real Pattern

Here is a practical example combining CTEs with date/time functions. We calculate each hour's trip count, then compare it to the overall average:

```
WITH hourly_counts AS (
    SELECT
        date_part('hour', tpep_pickup_datetime) AS trip_hour,
        count(*) AS num_trips
    FROM nyc_yellow_taxi_trips
```

```
    GROUP BY trip_hour
),
overall AS (
    SELECT round(avg(num_trips), 0) AS avg_hourly_trips
    FROM hourly_counts
)
SELECT
    h.trip_hour,
    h.num_trips,
    o.avg_hourly_trips,
    CASE
        WHEN h.num_trips > o.avg_hourly_trips THEN 'Above Average'
        WHEN h.num_trips < o.avg_hourly_trips THEN 'Below Average'
        ELSE 'Average'
    END AS comparison
FROM hourly_counts h
CROSS JOIN overall o
ORDER BY h.trip_hour;
```

CTE 1 aggregates by hour. CTE 2 computes the average across all hours. The main query joins them with `CROSS JOIN` (since `overall` is a single row) and classifies each hour.

This pattern – aggregate, compute a benchmark, compare – shows up constantly in real analysis.

10 Part 10: Putting It All Together (Exercises)

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

10.1 Combined Exercises

10.1.1 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.)

```
-- Your query here
```

. . .

```sql
SELECT
    CASE date_part('dow', tpep_pickup_datetime)
        WHEN 0 THEN 'Sunday'
        WHEN 1 THEN 'Monday'
        WHEN 2 THEN 'Tuesday'
        WHEN 3 THEN 'Wednesday'
        WHEN 4 THEN 'Thursday'
        WHEN 5 THEN 'Friday'
        WHEN 6 THEN 'Saturday'
    END AS day_name,
    count(*) AS num_trips
FROM nyc_yellow_taxi_trips
GROUP BY day_name
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.

10.1.2 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?

```sql
-- Your query here
```

. . .

```sql
SELECT
    tpep_pickup_datetime AS nyc_time,
    tpep_pickup_datetime AT TIME ZONE 'Asia/Tokyo' AS tokyo_time
FROM nyc_yellow_taxi_trips
ORDER BY tpep_pickup_datetime
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.

10.1.3 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()`.

```
-- Your query here

...
```

```
SELECT
    percentile_cont(0.25)
        WITHIN GROUP (ORDER BY tpep_dropoff_datetime - tpep_pickup_datetime)
        AS p25,
    percentile_cont(0.50)
        WITHIN GROUP (ORDER BY tpep_dropoff_datetime - tpep_pickup_datetime)
        AS median,
    percentile_cont(0.75)
        WITHIN GROUP (ORDER BY tpep_dropoff_datetime - tpep_pickup_datetime)
        AS p75
FROM nyc_yellow_taxi_trips;
```

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

11 Part 11: What We Learned

11.1 Summary

11.1.1 Key Concepts

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

Concept	What It Does
`justify_interval()`	Clean up messy interval output
`WITH` (CTE)	Define named temporary result sets for readability and reuse
`COALESCE`	Return the first non-NULL value (essential for outer joins)
`HAVING`	Filter groups after aggregation (unlike `WHERE`, which filters rows)

11.1.2 The Big Ideas

1. **Always use `timestamptz`.** 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. **CTEs make complex queries readable.** Break multi-step logic into named stages that flow top-to-bottom.
5. **Real-world data is messy.** Always sanity-check your results. A 24-hour taxi ride with zero distance is not a real ride.
6. **`date_part()` is your Swiss Army knife.** Need to group by hour? By month? By quarter? It handles all of them.

11.1.3 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