Chapter 4

Dealing with Time Zones

As long as your database is only used at one location, you don't need to worry about time zones. But when you have temporal data tied to different locations around the world, you may get different local times as input, and you may need to provide local times as output. To deal with it, you can rely on PostgreSQL support for time zones.

In this chapter, we are going to complement our compendium of time related types with types which are capable of storing time zone information.

We will also look at the way Postgres stores values of these types, and the tools it provides for converting timestamps between different time zones.

But prior to all of that, I would like to explain what time zones are and how they work, because they can definitely be tricky.

Time Zones

Time zones are a relatively recent invention, and they only got established in their modern form around the world in the 20th century. Before time zones, everybody kept their own time which was linked to the solar day in that particular location. Of course, the Earth rotates, and so the sunrise and sunset actually occur at different times in different places.

It wasn't a big deal when travel took a long time and thus everything was fairly isolated. But with the development of railroads and the telegraph in the 19th century, travel became much faster and instant communication became possible. Now there was a need to have a synchronised time, and to exchange times that would make sense to both parties in different locations.

Synchronised time was initially adopted by railroads, the first one being the Great Western Railway in 1840. In 1852, the Royal Greenwich Observatory started transmitting time signals via telegraph. My country, New Zealand, was one of the first in the world to officially adopt a nationally observed standard time in 1868, which was set 11.5 hours ahead of the Greenwich Mean Time based on New Zealand's longitude.

By the start of the 20th century, the world was divided into time zones, because it allows nearby places to have the same time for the most part, and to have local time which is more or less aligned with the solar time.

To begin with, these time zones weren't based on standardised offsets from a reference time which is now called Universal Coordinated Time, or UTC. The conversion to standard offsets happened gradually, with Nepal being the last to switch in 1986.

How time zones are defined

There are actually two types of time zones: nautical and terrestrial. Let's start with the nautical time zones even though Postgres doesn't have explicit support for them.

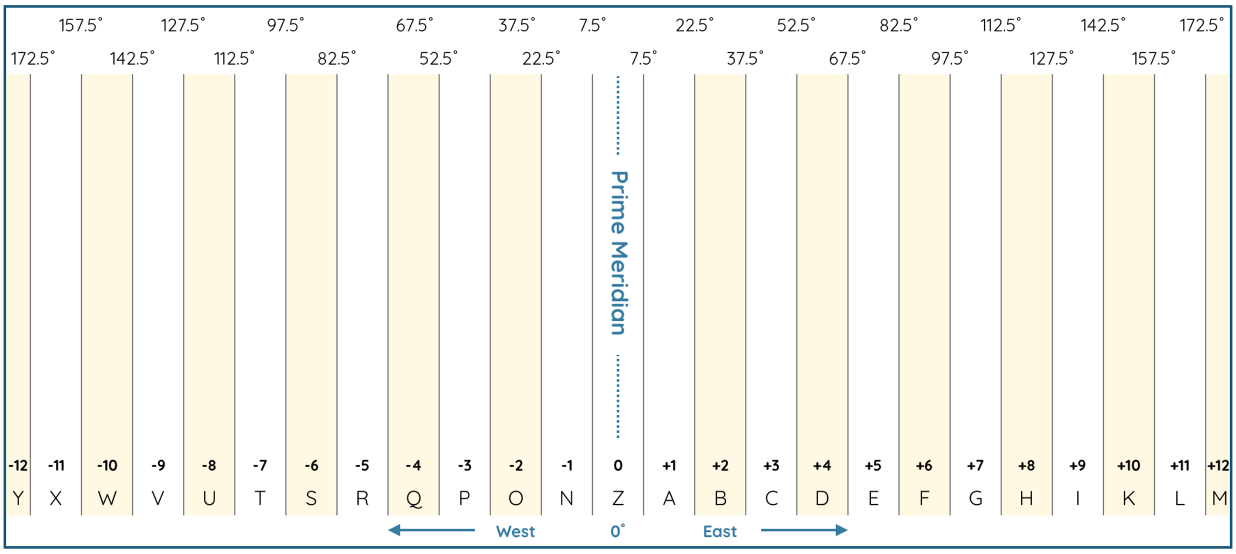

Figure 4.1: Nautical time zones

Nautical time zones work the way you'd logically expect, so they are useful for demonstrating the basic principle. The whole planet is divided into 24 15-degree slices along the meridians. The slice with zero offset has the prime meridian running through the middle, and it is on UTC time.

The prime meridian is an arbitrary meridian at which longitude is defined to be 0 degrees. It can also be referred to as the Reference meridian or Greenwich meridian because when it was originally chosen, it was based at the Royal Greenwich Observatory in London.

Each subsequent slice is offset by 1 hour when moving west, or -1 hour when moving to the east. The slice on the opposite side of the planet to the zero slice is additionally chopped into two halves, one of them with an offset of +12 hours, and another with an offset of -12 hours.

Now let's look at the terrestrial time zones. They present a much less elegant picture because geography, politics and culture get thrown into the mix to define them.

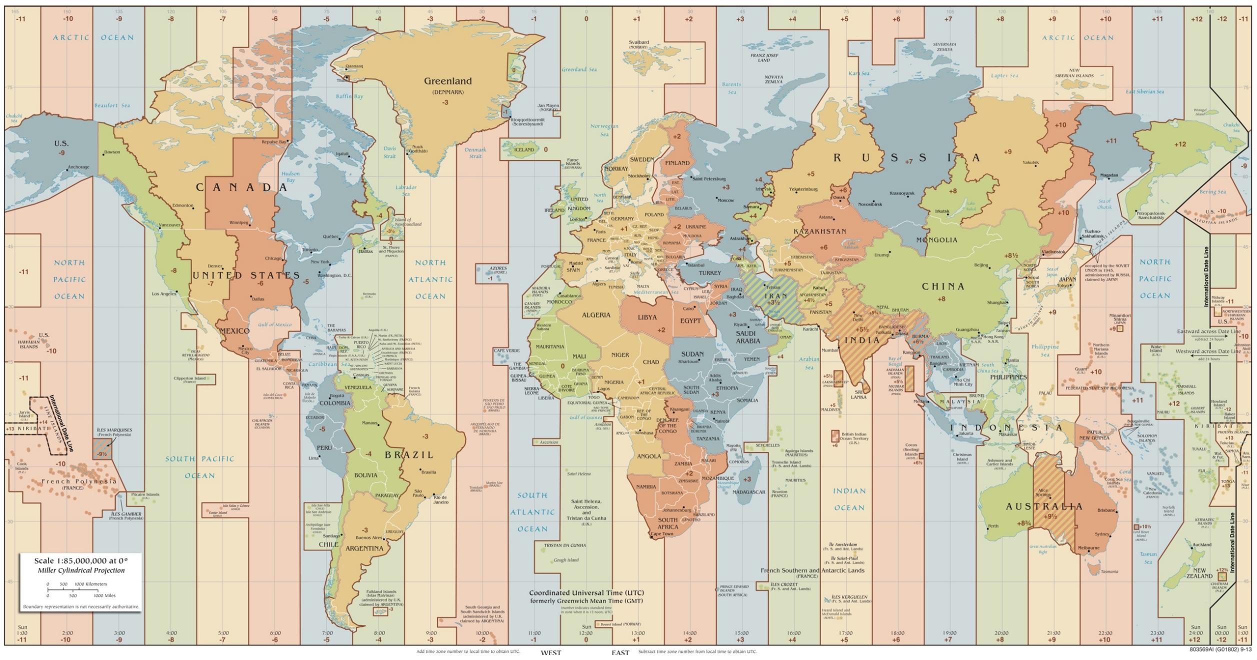

Figure 4.2: Standard world time zones

Some people look at this and think that it would be easier just to have everybody in the world on UTC, but I disagree. At the very least, it's not easier, and it might well be harder because with time zones, the local time is roughly tied to the solar day, but without it, you'd have to work out how the solar day, the work hours and so on relate to the UTC time. If there were no time zones, and you had to call somebody in Europe from the US on a Wednesday morning, then you'd have to work out what their work hours are in the first place (e.g. they start at 5am).

Going back to time zones, the local time is, in fact, defined by the UTC time and the latitude/longitude coordinates on the surface of the planet. Usually, you won't need to check the latitude/longitude coordinates against time zone geofences. Often, users provide their time zone name via settings.

Now, if you look at the nautical time zones, they only allow two dates to exist contemporaneously. Terrestrial zones are fancier, and they actually allow three dates to coexist at the same UTC time. This is because the terrestrial offsets go all the way to +14 in the Line Islands which are part of the Republic of Kiribati. Samoa and Tonga also shift to +14 during their daylight saving time.

So when it's 1am on Wednesday in the Line Islands of Kiribati, it's 10 or 11pm on Tuesday in New Zealand, and it's 11pm on Monday in Niue - another island nation in the Pacific which is located, interestingly, to the west of Kiribas.

Another thing worth mentioning is that some time zones are offset by a fractional number of hours. For example, Indian Standard Time is UTC +5.5 hours, and New Zealand's Chatham Islands are 12 hours 45 minutes ahead of UTC.

Depending on their geography, some countries have just one time zone, and some have multiple. The country with the most time zones is France, which is because of its overseas territories. French Polynesia is 10 hours behind UTC, and Wallis & Futuna is 12 hours ahead of UTC. Of course, all of these interesting places are also islands in the Pacific Ocean.

Some countries such as India and China use a single time zone even though geographically it doesn't make sense and for some parts of the country the solar day ends up being quite unaligned with the local time.

Just as if this isn't complicated enough, there is also daylight saving time (DST), which is the practice of moving clocks forward by some amount, usually one hour, for the summer months. In the countries applying DST, the time zones have extra rules attached to them to define when this happens.

Finally, it's important to remember that time zones are not static. Their offsets and daylight savings rules can and do change, sometimes with little notice. For example, until 2011 Russia had DST, but in 2011 it decided to stay on DST all year long.

This caused a lot of complaints because winter mornings became way too dark, so in 2014 Russia shifted back to standard time, this time abandoning DST altogether. Israel's DST rules used to be so irregular that Microsoft Windows didn't support them properly until Israel standardised them in 2013. Haiti cancelled its DST for 2016 with just one day's notice. As you can see, things can be pretty chaotic.

Time zone database

So how does PostgreSQL know how to perform conversions between time zones? Like most other software, it relies on the time zone database maintained by IANA. This database is also known as the Olson database, after its longtime previous maintainer Arthur David Olson.

This database records historical and current rules for all time zones, and Postgres looks them up when doing calculations with timestamps. You'll learn more about this database in the last chapter where I discuss various time related configuration in PostgreSQL.

Time zone summary

Before we move on to the details of PostgreSQL support for time zones, let's take stock of what we've learned about time zones so far:

- They are irregularly shaped geofences on the surface of the Earth.
- Each time zone is defined in terms of an offset from universal coordinated time, generally with negative offsets to the east of the prime meridian and positive offsets to the west.
- Time zones are not static: they have changed in the past, and they can always change in the future.
- In addition to their regular offsets from UTC, they can have an extra daylight saving offset for part of the year, which is also subject to change.
- Time zones and DST lead to interesting effects such as three contemporaneous dates at different points on the globe, as well as 23 or 25 hour days.

Time Zone Aware Types

In order to support time zones, Postgres provides several additional types I didn't include in the previous chapter. There is a timestamp type, a time type, and a timestamp range. The date type doesn't have a time zone aware equivalent.

`timestamptz`

The SQL standard specifies two types: `timestamp without time zone` which can be abbreviated to just `timestamp` as we've been doing in the previous chapter, and `timestamp with time zone` which in Postgres can also be referred to as `timestamptz`.

This type behaves very similarly to `timestamp without time zone`, with the key difference that it will use time zone information:

```sql
select timestamptz '2015-10-21 01:00:00+5:30';
```

```
      timestamptz
------------------------
 2015-10-20 19:30:00+00
```

We can see that a different timestamp is displayed, because on input, Postgres converts everything into UTC. In fact, after the conversion it discards the time zone information and only stores the UTC time. So the name "timestamp with time zone" doesn't actually mean that it *stores* the time zone. Instead, it means that it is capable of taking into account the time zone on input and output of such values. On output, each `timestamptz` is converted according to the time zone setting for the session.

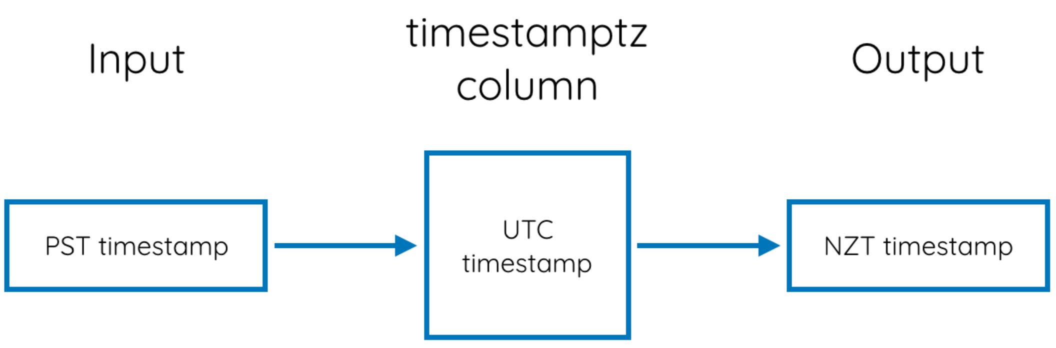

Figure 4.3: Example conversions on `timestamptz` input and output

Postgres has a bunch of rules to look up the time zone, and for every session it sets the time zone accordingly. I'll explain the full rules for this in the last chapter. The important thing is that there is always *some* time zone set for the session.

You could reasonably ask: what is the point of `timestamptz` if it doesn't store the time zone? Why not just use `timestamp` columns, supply UTC values on input and convert to the desired time zone on output?

It is indeed an option, but the automatic `timestamptz` conversions are quite convenient, and an interesting thing is that you can use the `set time zone <name>` command to change the time zone setting for the duration of your session:

```sql
set time zone 'NZ';
set time zone 'US/Pacific';
```

This gives us two possibilities:

- If there are people looking at the DB from different time zones, each of them can configure their session to show timestamps converted to their time zone. This is useful, for example, if you're investigating something that happened 2 hours ago, that is, if you are dealing with relative times.
- If there is data coming from a particular time zone, for example a stream of positions from a particular customer who is in Australia, then you can set your session time zone to Australia, and all of the timestamps will be displayed in customer's local time. This is useful, for example, if you're investigating something that happened on Monday morning in the customer's time zone.

If you need to do conversions to other time zones in queries and the like, it's also more convenient to use `timestamp with time zone`.

There is one more thing to be aware of. If you accidentally create a `timestamp without time zone` column instead of `with time zone`, and then insert values with time zone information into it, Postgres will silently discard the time zone information and use the timestamp anyway. This is a common mistake, so be careful when defining columns, otherwise it may be a while before the problem is discovered.

Specifying the time zone

Note that you aren't limited to offsets which correspond to actual time zones. You can specify any value down to seconds:

```
select timestamptz '2015-10-21 01:00:00+1:23:45';
```

```
      timestamptz
------------------------
 2015-10-20 23:36:15+00
```

A time zone name or an abbreviation known to Postgres can be used instead of an offset:

```
select timestamptz '2015-10-21 01:00:00 America/Los_Angeles';
```

```
      timestamptz
------------------------
 2015-10-21 08:00:00+00
```

You may be wondering where to find these names. Postgres provides a list of supported time zones in the `pg_timezone_names` view:

```
select * from pg_timezone_names;
```

```
              name                | abbrev | utc_offset | is_dst
----------------------------------+--------+------------+--------
 Africa/Abidjan                   | GMT    | 00:00:00   | f
 Africa/Accra                     | GMT    | 00:00:00   | f
 Africa/Addis_Ababa               | EAT    | 03:00:00   | f
 Africa/Algiers                   | CET    | 01:00:00   | f
 Africa/Asmara                    | EAT    | 03:00:00   | f
 Africa/Asmera                    | EAT    | 03:00:00   | f
 Africa/Bamako                    | GMT    | 00:00:00   | f
 ...
```

This view includes the zone name, the corresponding abbreviation, the zone's offset, and a flag to indicate whether daylight saving time is observed in that time zone as of the current timestamp.

There is also a list of abbreviations provided in the `pg_timezone_abbrevs` view:

```
select * from pg_timezone_abbrevs;
```

```
 abbrev | utc_offset | is_dst
--------+------------+--------
 ACDT   | 10:30:00   | t
 ACSST  | 10:30:00   | t
 ACST   | 09:30:00   | f
 ACT    | -05:00:00  | f
 ACWST  | 08:45:00   | f
 ADT    | -03:00:00  | t
 AEDT   | 11:00:00   | t
 ...
```

There are some subtleties in the handling of abbreviations, and I will discuss them later in the chapter when I highlight some of the things that could trip you up.

Calculations

When it comes to manipulating `timestamptz` values, you can use all of the same operators and functions that we discussed in connection with the timestamp type.

One difference is that instead of `make_timestamp`, you need to use `make_timestamptz` which takes an additional time zone name parameter. If you don't specify a time zone, then the database session time zone is used:

```
select make_timestamptz(2015, 10, 21, 1, 2, 3.4);
```

```
     make_timestamptz
--------------------------
 2015-10-21 01:02:03.4+00
```

If you specify a time zone, then the timestamp is considered to be in that time zone and is converted to UTC as usual:

```sql
select make_timestamptz(2015, 10, 21, 1, 2, 3.4, 'NZ');
```

```
     make_timestamptz
--------------------------
 2015-10-20 12:02:03.4+00
```

Another difference is that when working with `timestamptz`, you can use 3 extra parameters with `date_part` and `extract`:

- `timezone`
- `timezone_hour`
- `timezone_minute`

`timezone` parameter produces the offset from UTC in seconds, whereas `timezone_hour` and `timezone_minute` produce the hour and minute portions of the offset.

Note that they *always* produce the offset using the time zone setting for the current session, because that's the only time zone Postgres is aware of.

`timetz` / `time with time zone`

This type is not recommended for use. It's only included for legacy applications and for compliance with the SQL standard.

The SQL standard has done something strange in this case. As you've learned, calculating local time requires knowing *both* the UTC time and the time zone. But `timetz` does not include the date, thus making it impossible to apply the time zone rules and calculate the correct offset, so it doesn't make sense to use it in practice.

`tstzrange`

`tstzrange` consists of two timestamps with time zone. Other than that, it behaves just like `tsrange`, so I'll keep this section brief.