How to navigate the “interval” minefield

Bryn Llewellyn

Technical Product Manager at Yugabyte

yugabyteDB

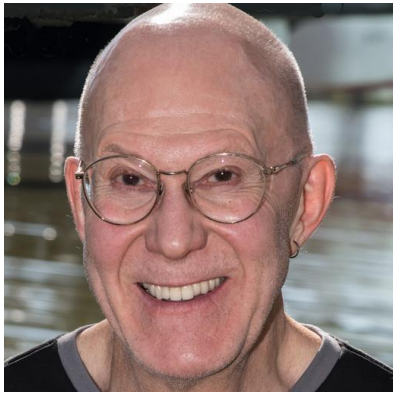
Who am I?

linkedin.com/in/bryn-llewellyn/

twitter.com/BrynLite

blog.yugabyte.com/author/bryn/

Google for: “PostgreSQL Person of the Week” Bryn



Who do I think you are?

- You know PostgreSQL very well
- Not a week goes by without you typing SQL at the *psql* prompt
- You don't need me to tell you about the reasons to use SQL
- You don't mind that Codd and Date laid the foundations a very long time ago
- Maybe you even have some exposure to YugabyteDB
- **You might find the whole business of *date-time* datatypes, and the operations that use these, mysterious and daunting**

References

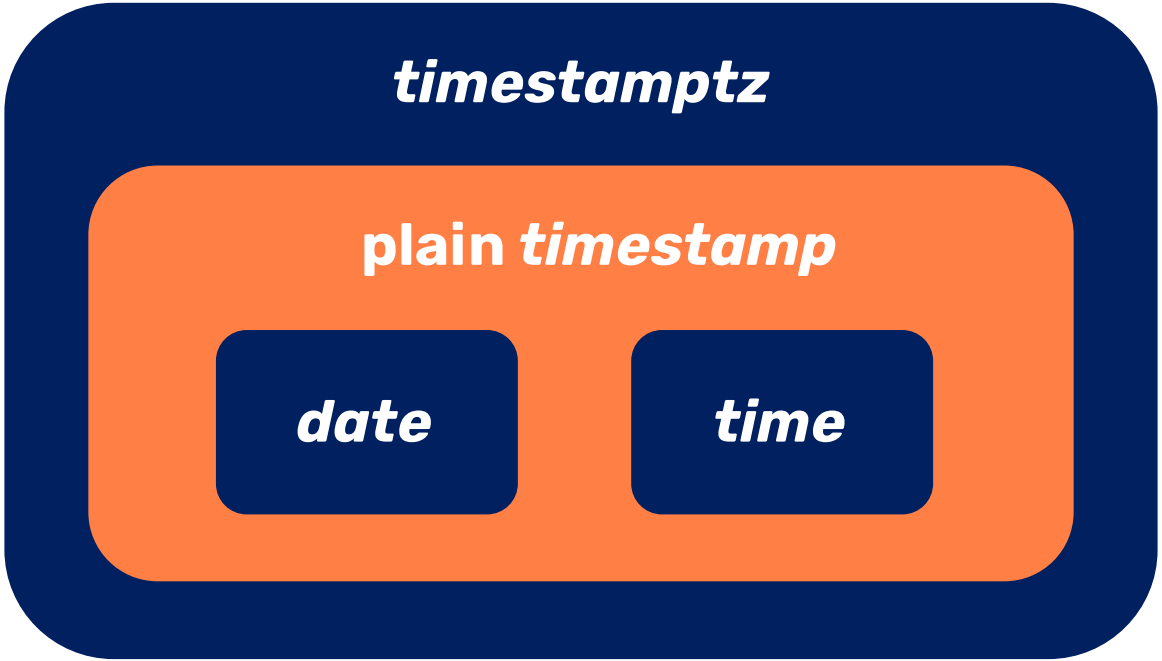
Table of contents for the date-time data types section (YugabyteDB doc)
Download and install the date-time utilities code
Blogpost: PostgreSQL Timestamps and Timezones: How to Navigate the Interval Minefield

Caveat:
moments and durations are tough stuff

- The date-time apparatus is vast and complex
- It's complex because of inescapable facts of astronomy and human convention
- The SQL Standard folks introduced notions in successive iterations of thinking — bringing some inconsistency
- Early PostgreSQL versions implemented questionable decisions
- You can easily go wrong
- For new work, you need only a small "tamed" subset of the whole apparatus. This depends on some user-defined utilities

Choosing the right “moment” data type from *date*, *time*, plain *timestamp*, *timestamptz*

The plain timestamp and timestamptz data types



- plain *timestamp* combines *date* and *time*
- *timestamptz* adds timezone awareness to plain *timestamp*

- **Always prefer *timestamptz* for data that you persist**

- **If appropriate, record the reigning timezone offset and name when the moment was recorded in partner columns**

- **You can always derive a plain *timestamp* value, a *date* value, or a *time* value from a *timestamptz* value**

- **Document your reasons for going against this recommended practice if you decide that you have to. This will clarify your thinking for yourself and others — and creating the write-up might make you change your mind**

timestamptz example

```
drop function if exists reigning_timezone_offset() cascade;
create function reigning_timezone_offset()
  returns interval
  language plpgsql
as $body$
declare
  t  constant timestamptz not null := transaction_timestamp();
  h  constant int         not null := date_part('timezone_hour', t)::int;
  m  constant int         not null := date_part('timezone_minute', t)::int;
  i  constant interval    not null := make_interval(hours=>h, mins=>m);
begin
  return i;
end;
$body$;

drop table if exists events cascade;
create table events(
  k               serial       primary key,
  created_ts      timestamptz  not null default transaction_timestamp(),
  created_tz      interval     not null default reigning_timezone_offset(),
  created_tzname  text         not null default current_setting('TimeZone'),
  what            text         not null);
```

timestamptz example — *cont.*

```
call set_timezone('America/Los_Angeles');
insert into events(What) values('Arrived Los Angeles');

call set_timezone('Europe/London');
insert into events(What) values('Arrived London');

call set_timezone('Asia/Kathmandu');
insert into events(What) values('Arrived Kathmandu');

call set_timezone('UTC');
select created_ts::text, created_tz::text, created_tzname, what
from events
order by k;
```

Result:

```
          created_ts            | created_tz |   created_tzname    |        what
--------------------------------+------------+---------------------+--------------------
 2021-10-27 01:34:01.597628+00 | -07:00:00  | America/Los_Angeles | Arrived Los_Angeles
 2021-10-27 01:34:01.658141+00 |  01:00:00  | Europe/London       | Arrived London
 2021-10-27 01:34:01.708605+00 |  05:45:00  | Asia/Kathmandu      | Arrived Kathmandu
```

timestamptz example — *cont.*

```
with c as (
  select k, at_timezone(created_tz, created_ts) as ts, what
  from events)
select
  to_char(ts::date, 'Day dd-Mon-yyyy'         )  as "Local Date",
  to_char(ts::time,                 'hh24:mi')  as "Local Time",
  what
from c
order by k;
```

Result:

```
      Local Date        | Local Time |        what
------------------------+------------+--------------------
 Tuesday   26-Oct-2021 | 18:34      | Arrived Los_Angeles
 Wednesday 27-Oct-2021 | 02:34      | Arrived London
 Wednesday 27-Oct-2021 | 07:19      | Arrived Kathmandu
```

interval arithmetic

Interval arithmetic
Sensitivity of timestamptz-interval arithmetic to the current timezone

Clock-time and calendar time durations

- This is entirely a matter of human convention—which, in turn, sets the requirements for an implementaion.
- Clock-time durations are measured in *seconds*, *minutes*, and *hours*. They're good for journeys that cross timezones and especially for ones that last more than 24 hours.
- Calendar-time durations are sometimes measured in *days* and sometimes in *months* and *years* (where one year is just shorthand for twelve months).
- The semantics are different for these three kinds of duration.

Clock-time use case: cycling from Los Angeles to San Francisco

- A strong cyclist sets out at eight on Saturday evening, 12-Mar-2022 on a non-stop road trip from LA to San Francisco. It's about 740 km by the shortest bikeable route. It should be able to manage a bit more than 30 km per hour on an overnight endurance challenge—resulting in an exactly twenty-four hour ride.

- (This is nothing compared to Tour de France speeds!)

Clock-time use case — *cont:* cycling from Los Angeles to San Francisco

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  start   constant timestamptz not null := '2022-03-12 20:00 America/Los_Angeles';
  i       constant interval    not null := '24 hours';
  finish  constant timestamptz not null := start + i;
begin
  call set_timezone('America/Los_Angeles');
  z := 'start:  '||start  ::text;          return next;
  z := 'finish: '||finish ::text;          return next;
end;
$body$;

select z from f();
```

Clock-time use case — *cont:* cycling from Los Angeles to San Francisco

```
start:  2022-03-12 20:00:00-08
finish: 2022-03-13 21:00:00-07
```

“Day” calendar-time use case: postponing a meeting by one day

- You live in LA and you’d arranged to ring a friend in San Francisco at eight on Saturday evening, 12-Mar-2022. Something came up and you had to message your friend to say “I have to push out our call by a day. Same time tomorrow. OK?” There’s actually only twenty-three clock duration hours between eight on Saturday evening, 12-Mar-2022 and eight on Sunday evening, 13-Mar-2022 in this timezone—yet, by convention, the duration is nevertheless understood to be one calendar time day so that your friend understands that you mean that the new time for the call is eight on Sunday evening.

“Day” calendar-time use case — *cont:* postponing a meeting by one day

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  orig     constant timestamptz not null := '2022-03-12 20:00 America/Los_Angeles';
  i        constant interval    not null := '1 day';
  updated  constant timestamptz not null := orig + i;
begin
  call set_timezone('America/Los_Angeles');
  z := 'orig:    '||orig    ::text;          return next;
  z := 'updated: '||updated ::text;          return next;
end;
$body$;

select z from f();
```

“Day” calendar-time use case — *cont:* postponing a meeting by one day

```
orig:    2022-03-12 20:00:00-08
updated: 2022-03-13 20:00:00-07
```

"Month" calendar-time use case: adding one month to a date

- Consider the dates from 27-Jan through 31-Jan in some non-leap year.
- Add one month to each of them.
- What do you want to see?
- You can only want 27-Feb for the first and 28-Feb for all the others — else, you'd add, say, *30 days* and not *1 month*

“Month” calendar-time use case — *cont*: adding one month to a date

```
drop function if exists fmt(text, timestamptz) cascade;
create function fmt(tz in text,t in timestamptz)
  returns text
  language sql
as $body$
  select to_char(at_timezone(tz, t), 'dd-Mon-yyyy');
$body$;
```

“Month” calendar-time use case— *cont*: adding one month to a date

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  NY  constant text        not null := 'America/New_York';
  t1  constant timestamptz not null := '2019-01-27 12:00'||NY;
  t2  constant timestamptz not null := '2019-01-28 12:00'||NY;
  t3  constant timestamptz not null := '2019-01-29 12:00'||NY;
  t4  constant timestamptz not null := '2019-01-30 12:00'||NY;
  t5  constant timestamptz not null := '2019-01-31 12:00'||NY;

  i       constant interval    not null := '1 month';
begin
  call set_timezone('America/New_York');
  z := 't1, t1 + i: '||fmt(NY, t1)||', '||fmt(NY, (t1  + i));  return next;
  z := 't2, t2 + i: '||fmt(NY, t2)||', '||fmt(NY, (t2  + i));  return next;
  z := 't3, t3 + i: '||fmt(NY, t3)||', '||fmt(NY, (t3  + i));  return next;
  z := 't4, t4 + i: '||fmt(NY, t4)||', '||fmt(NY, (t4  + i));  return next;
  z := 't5, t5 + i: '||fmt(NY, t5)||', '||fmt(NY, (t5  + i));  return next;
end;
$body$;

select z from f();
```

“Month” calendar-time use case— *cont*: adding one month to a date

```
t1, t1 + i: 27-Jan-2019, 27-Feb-2019
t2, t2 + i: 28-Jan-2019, 28-Feb-2019
t3, t3 + i: 29-Jan-2019, 28-Feb-2019
t4, t4 + i: 30-Jan-2019, 28-Feb-2019
t5, t5 + i: 31-Jan-2019, 28-Feb-2019
```

Adding an *interval* to a *timestamptz* — semantics

- The semantic rules are different for the three kinds of "pure" *interval* values: pure months; pure days; and pure seconds

- In particular, the rules for pure seconds *interval* values are timezone-sensitive when the value spans the Daylight Savings Time moment

- Who knows what the rules are for hybrid *interval* values !

  (The PG doc is silent on the topic.)

The internal representation of an *interval* arithmetic

How does YSQL represent an interval value?

Create functions to provide some re-usable *interval* and *timestamptz* values

```
drop function if exists i1() cascade;
create function i1()
  returns interval
  language sql
as $body$
  select '1 month'::interval;
$body$;

drop function if exists i2() cascade;
create function i2()
  returns interval
  language sql
as $body$
  select '30 days'::interval;
$body$;

drop function if exists i3() cascade;
create function i3()
  returns interval
  language sql
as $body$
  select ((24*30)::text||' hours')::interval;
$body$;

drop function if exists t1() cascade;
create function t1()
  returns timestamptz
  language sql
as $body$
  -- The "spring forward" moment is '2022-03-13 02:00:00 America/Los Angeles'.
  select '2022-03-10 12:00:00 America/Los_Angeles'::timestamptz;
$body$;

drop function if exists t2() cascade;
create function t2()
  returns timestamptz
  language sql
as $body$
  select '2022-04-15 12:00:00 America/Los_Angeles'::timestamptz;
$body$;
```

interval value internal representation example

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
begin
  z := 'i1:                        '||i1()::text;  return next;
  z := 'i2:                        '||i2()::text;  return next;
  z := 'i3:                        '||i3()::text;  return next;
  z := '';                                                          return next;
  z := 'interval_mm_dd_ss(i1): '||interval_mm_dd_ss(i1())::text;  return next;
  z := 'interval_mm_dd_ss(i2): '||interval_mm_dd_ss(i2())::text;  return next;
  z := 'interval_mm_dd_ss(i3): '||interval_mm_dd_ss(i3())::text;  return next;
end;
$body$;

select z from f();
```

function interval_mm_dd_ss(interval) returns interval_mm_dd_ss_t

interval value internal representation example — *cont*

```
i1:                          1 mon
i2:                          30 days
i3:                          720:00:00

interval_mm_dd_ss(i1): (1,0,0)
interval_mm_dd_ss(i2): (0,30,0)
interval_mm_dd_ss(i3): (0,0,2592000)
```

function interval_mm_dd_ss(interval) returns interval_mm_dd_ss_t

interval value equality example

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
begin
  z := 'i1:                        '||i1()::text;                        return next;
  z := 'i2:                        '||i2()::text;                        return next;
  z := 'i3:                        '||i3()::text;                        return next;
  z := '';                                                               return next;
  z := 'interval_mm_dd_ss(i1): '||interval_mm_dd_ss(i1())::text;  return next;
  z := 'interval_mm_dd_ss(i2): '||interval_mm_dd_ss(i2())::text;  return next;
  z := 'interval_mm_dd_ss(i3): '||interval_mm_dd_ss(i3())::text;  return next;
  z := '';                                                               return next;
  z := 'i1 = i2:                '||(i1() = i2())::text;                 return next;
  z := 'i1 = i3:                '||(i1() = i3())::text;                 return next;
  z := 'i2 = i3:                '||(i2() = i3())::text;                 return next;
  z := '';                                                               return next;
  z := 'i1 == i2:               '||(i1() == i2())::text;                return next;
  z := 'i1 == i3:               '||(i1() == i3())::text;                return next;
  z := 'i2 == i3:               '||(i2() == i3())::text;                return next;
end;
$body$;

select z from f();
```

The user-defined "strict equals" interval-interval "==" operator

interval value equality example — *cont*

```
i1:                          1 mon
i2:                          30 days
i3:                          720:00:00

interval_mm_dd_ss(i1): (1,0,0)
interval_mm_dd_ss(i2): (0,30,0)
interval_mm_dd_ss(i3): (0,0,2592000)

i1 = i2:                     true
i1 = i3:                     true
i2 = i3:                     true

i1 == i2:                    false
i1 == i3:                    false
i2 == i3:                    false
```

timestamptz value plus *interval* value example

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
begin
  call set_timezone('America/Los_Angeles');
  z := 'i1:                       '||i1()::text;                       return next;
  z := 'i2:                       '||i2()::text;                       return next;
  z := 'i3:                       '||i3()::text;                       return next;
  z := '';                                                             return next;
  z := 'interval_mm_dd_ss(i1): '||interval_mm_dd_ss(i1())::text;  return next;
  z := 'interval_mm_dd_ss(i2): '||interval_mm_dd_ss(i2())::text;  return next;
  z := 'interval_mm_dd_ss(i3): '||interval_mm_dd_ss(i3())::text;  return next;
  z := '';                                                             return next;
  z := 't1:                       '||t1()::text;                       return next;
  z := 't1 + i1:                  '||(t1() + i1())::text;              return next;
  z := 't1 + i2:                  '||(t1() + i2())::text;              return next;
  z := 't1 + i3:                  '||(t1() + i3())::text;              return next;
  z := '';                                                             return next;
  z := 't2:                       '||t2()::text;                       return next;
  z := 't2 + i1:                  '||(t2() + i1())::text;              return next;
  z := 't2 + i2:                  '||(t2() + i2())::text;              return next;
  z := 't2 + i3:                  '||(t2() + i3())::text;              return next;
end;
$body$;

select z from f();
```

The moment-interval overloads of the "+" and "-" operators

timestamptz value plus *interval* value example — *cont*

```
i1:                          1 mon
i2:                          30 days
i3:                          720:00:00

interval_mm_dd_ss(i1): (1,0,0)
interval_mm_dd_ss(i2): (0,30,0)
interval_mm_dd_ss(i3): (0,0,2592000)

t1:                          2022-03-10 12:00:00-08
t1 + i1:                     2022-04-10 12:00:00-07
t1 + i2:                     2022-04-09 12:00:00-07
t1 + i3:                     2022-04-09 13:00:00-07

t2:                          2022-04-15 12:00:00-07
t2 + i1:                     2022-05-15 12:00:00-07
t2 + i2:                     2022-05-15 12:00:00-07
t2 + i3:                     2022-05-15 12:00:00-07
```

timestamptz value minus *timestamptz* value example

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  i1  constant interval    not null := make_interval(hours=>(57*24 + 7));
  t1  constant timestamptz not null := '2022-03-01 12:00:00 America/Los_Angeles';
  t2  constant timestamptz not null := t1 + i1;
  i2  constant interval    not null := t2 - t1;
  t3  constant timestamptz not null := t1 + i2;
begin
  call set_timezone('America/Los_Angeles');
  z := 'i1: '||i1::text||' '||interval_mm_dd_ss(i1)::text;          return next;
  z := 't1: '||t1::text;                                           return next;
  z := 't2: '||t2::text;                                           return next;
  z := 'i2: '||i2::text||' '||interval_mm_dd_ss(i2)::text;          return next;
  z := 't3: '||t3::text;                                           return next;
end;
$body$;

select z from f();
```

The moment-moment overloads of the "-" operator

timestamptz value minus *timestamptz* value example – cont

```
i1: 1375:00:00 (0,0,4950000)

t1: 2022-03-01 12:00:00-08

t2: 2022-04-27 20:00:00-07

i2: 57 days 07:00:00 (0,57,25200)

t3: 2022-04-27 19:00:00-07
```

Minefield summary

- When you subtract one *timestamptz* value from another, you get an *interval* value whose *months* component is always zero and whose *days* and *seconds* components are, in general, both non-zero.

- In other words, you get (in general) a *hybrid* value. And you cannot influence this outcome. (The decorated *interval* declarations don't help here and seem to be a solution looking for a problem.)

- But the rules are for hybrid *interval* values are undefined.

  (The PG doc is silent on the topic.)

decorated *interval* declararions

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  i  constant interval                 not null := '1 year 10 months 29 days 23:59:59.123456';
  ss constant interval           second not null := i;
  mi constant interval           minute not null := i;
  hh constant interval           hour   not null := i;
  dd constant interval           day    not null := i;
  mm constant interval           month  not null := i;
  yy constant interval           year   not null := i;

  ds constant interval day    to second not null := i;
  hs constant interval hour   to second not null := i;
  ms constant interval minute to second not null := i;
begin
  z := 'bare declaration: '||interval_mm_dd_ss(i)  ::text;                    return next;
  z := 'interval second   '||interval_mm_dd_ss(ss) ::text;                    return next;
  z := 'interval minute   '||interval_mm_dd_ss(mi) ::text;                    return next;
  z := 'interval hour:    '||interval_mm_dd_ss(hh) ::text;                    return next;
  z := 'interval day:     '||interval_mm_dd_ss(dd) ::text;                    return next;
  z := 'interval month:   '||interval_mm_dd_ss(mm) ::text;                    return next;
  z := 'interval year:    '||interval_mm_dd_ss(yy) ::text;                    return next;
  z := '';                                                                    return next;
  z := 'interval day    to second == interval second: '||(ds == ss)::text;    return next;
  z := 'interval hour   to second == interval second: '||(hs == ss)::text;    return next;
  z := 'interval minute to second == interval second: '||(ms == ss)::text;    return next;
end;
$body$;

select z from f();
```

Declaring intervals

decorated *interval* declararions — *cont*

```
bare declaration: (22,29,86399.123456)
interval second   (22,29,86399.123456)
interval minute   (22,29,86340)
interval hour:    (22,29,82800)
interval day:     (22,29,0)
interval month:   (22,0,0)
interval year:    (12,0,0)

interval day    to second == interval second: true
interval hour   to second == interval second: true
interval minute to second == interval second: true
```

Custom domain types for specializing the native interval functionality

Custom domain types for specializing the native interval functionality

Recommended practice for *interval* values and *interval* arithmetic

- These domains ensure that you work only with pure *interval* values
- You get an error if you try, say, to add a pure seconds *interval* value to a pure days *interval* value
- They provide functions for moment subtraction and to multiply, say, a pure seconds *interval* value by a real number to return a pure seconds *interval* value result

Subtracting one *timestamptz* value from another

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  t1         constant timestamptz not null := '2021-01-01 12:00:00 UTC';
  t2         constant timestamptz not null := '2021-11-30 23:55:55 UTC';
  i_hybrid   constant text        not null := interval_mm_dd_ss(t2 - t1);
  i_mm       constant text        not null := interval_mm_dd_ss(interval_months (t2, t1));
  i_dd       constant text        not null := interval_mm_dd_ss(interval_days   (t2, t1));
  i_ss       constant text        not null := interval_mm_dd_ss(interval_seconds(t2, t1));
begin
  z := 'i_hybrid: '||i_hybrid;                  return next;
  z := 'i_mm:     '||i_mm;                      return next;
  z := 'i_dd:     '||i_dd;                      return next;
  z := 'i_ss:     '||i_ss;                      return next;
end;
$body$;

select z from f();
```

Subtracting one *timestamptz* value from another — *cont*

```
i_hybrid:  (0,333,42955)
i_mm:      (10,0,0)
i_dd:      (0,333,0)
i_ss:      (0,0,28814155)
```

Subtracting one *timestamptz* value from another using the *interval_seconds_t* user-defined domain

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  t1  constant timestamptz        not null := '2011-03-01 12:00:00 America/Los_Angeles';
  i1  constant interval_seconds_t not null := interval_seconds(hours=>(11*365*24), mins=>33,
secs=>47);
  t2  constant timestamptz        not null := t1 + i1;
  i2  constant interval_seconds_t not null := interval_seconds(t2, t1);
begin
  call set_timezone(America/Los_Angeles);
  z := 'i1: '||interval_mm_dd_ss(i1)::text;          return next;
  z := 't1: '||t1::text;                             return next;
  z := 't2: '||t2::text;                             return next;

  assert i2 == i1, 'assert failed';
end;
$body$;

select z from f();
```

Subtracting one *timestamptz* value from another using the *interval_seconds_t* user-defined domain – *cont*

```
i1: (0,0,346898027)
t1: 2011-03-01 12:00:00-08
t2: 2022-02-26 12:33:47-08
```

Finishes without error – `so "assert i2 == i1"` succeeded

Subtracting one *timestamptz* value from another using the *interval_days_t* user-defined domain

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  t1  constant timestamptz     not null := '2011-03-01 12:00:00 America/Los_Angeles';
  i1  constant interval_days_t not null := interval_days(days=>(11*365 + 93));
  t2  constant timestamptz     not null := t1 + i1;
  i2  constant interval_days_t not null := interval_days(t2, t1);
begin
  call set_timezone('America/Los_Angeles');
  z := 'i1: '||interval_mm_dd_ss(i1)::text;          return next;
  z := 't1: '||t1::text;                             return next;
  z := 't2: '||t2::text;                             return next;

  assert i2 == i1, 'assert failed';
end;
$body$;

select z from f();
```

Subtracting one *timestamptz* value from another using the *interval_days_t* user-defined domain

```
i1: (0,4108,0)
t1: 2011-03-01 12:00:00-08
t2: 2022-05-30 12:00:00-07
```

Finishes without error — `so "assert i2 == i1"` succeeded

Subtracting one *timestamptz* value from another using the *interval_months_t* user-defined domain

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  t1  constant timestamptz       not null := '2011-03-01 12:00:00 America/Los_Angeles';
  i1  constant interval_months_t not null := interval_months(months=>(11*12 + 7));
  t2  constant timestamptz       not null := t1 + i1;
  i2  constant interval_months_t not null := interval_months(t2, t1);
begin
  call set_timezone('America/Los_Angeles');
  z := 'i1: '||interval_mm_dd_ss(i1)::text;          return next;
  z := 't1: '||t1::text;                             return next;
  z := 't2: '||t2::text;                             return next;

  assert i2 == i1, 'assert failed';
end;
$body$;

select z from f();
```

Subtracting one *timestamptz* value from another using the *interval_months_t* user-defined domain

```
i1: (139,0,0)
t1: 2011-03-01 12:00:00-08
t2: 2022-10-01 12:00:00-07
```

Finishes without error — `so "assert i2 == i1"` succeeded

Multiplying an *interval* value by a real number using the user-defined domain features

```
drop function if exists f() cascade;
create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  f   constant numeric           not null := 2.111;
  i1  constant interval          not null := '2 months';
  i2  constant interval          not null := i1*f;

  i3  constant interval_months_t not null := i1;
  i4  constant interval_months_t not null := interval_months(i3, f);
begin
  z := 'i1: '||interval_mm_dd_ss(i1);          return next;
  z := 'i2: '||interval_mm_dd_ss(i2);          return next;
  z := '';                                     return next;
  z := 'i3: '||interval_mm_dd_ss(i3);          return next;
  z := 'i4: '||interval_mm_dd_ss(i4);          return next;
end;
$body$;

select z from f();
```

Example Three — multiplying an *interval* value by a real number using the user-defined domain features — *cont*

```
i1:  (2,0,0)
i2:  (4,6,57024)

i3:  (2,0,0)
i4:  (4,0,0)
```

Summary

Date and time data types
Download and install the date-time utilities code

- **You've seen many ways that you can produce nonsense results. Avoid the risk as follows:**

- **Use only *timestamptz* to persist date-time values. If appropriate, record the reigning creation/modification timezone name and offset**

- **Beware *interval* arithmetic**

- **The doc sections *"Recommended practice for specifying the UTC offset"* and *"Custom domain types for specializing the native interval functionality"* come to the rescue**

- **To write brand-new application code (if you're happy simply to accept Yugabyte's various recommendations without studying the reasoning that supports these) you'll need to read only a small part of the YSQL doc's *"Date and time data types"* major section**

Here's what you'll need to read for writing brand-new application code

- Conceptual background
- Real timezones that observe Daylight Savings Time
- Real timezones that don't observe Daylight Savings Time
- The plain *timestamp* and *timestamptz* data types
- Sensitivity of converting between *timestamptz* and plain *timestamp* to the UTC offset
- Sensitivity of *timestamptz-interval* arithmetic to the current timezone
- Recommended practice for specifying the UTC offset
- Custom domain types for specializing the native interval functionality

- **But if you have to maintain extant *date-time* code, especially if it's poorly commented, has no external design documentation, and its authors have vanished without trace, then...**

- **You'll have to bite the bullet and study the entire *"Date and time data types"* major YSQL doc section very carefully — and especially try the code examples and make sure that you understand why they get the results that they do**

Enjoy!

Finally...

Distributed PostgreSQL on a Google Spanner Architecture": (1) Storage Layer; and (2) Query Layer

Most Advanced Open Source Distributed SQL

Query Layer

World's Most Advanced
Open Source SQL Engine

Storage Layer

World's Most Advanced
Distributed OLTP Architecture

Reuse

Inspiration

yugabyteDB

Thank you

Join us on Slack:
www.yugabyte.com/slack

Star us on GitHub:
github.com/yugabyte/yugabyte-db

yugabyteDB