yugabyteDB

Timestamps, timezones, and interval arithmetic ~ what you need to know, and what you don’t need to know

Bryn Llewellyn

Technical Product Manager at Yugabyte

Who am I?
~
Who do I think you are?

PostgreSQL Person of the Week — Bryn Llewellyn — 2-Aug-2021

Bryn Llewellyn

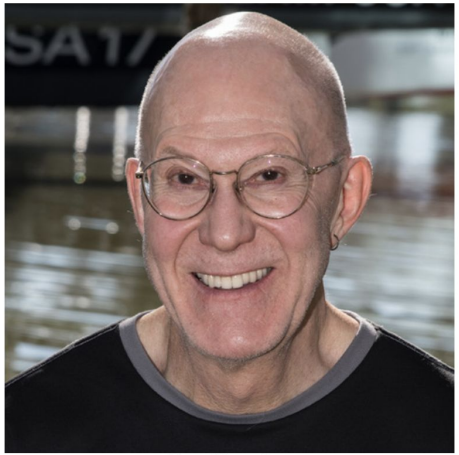
02.08.2021

Tags: postgresql database distributed sql yugabyte
Category: Interviews
Interviewed by: Andreas Scherbaum

PostgreSQL is the World's most advanced Open Source Relational Database. The interview series "PostgreSQL Person of the Week" presents the people who make the project what it is today. Read all interviews here.

LinkedIn, Twitter, and blogs

- LinkedIn: Bryn Llewellyn
- Twitter: @BrynLite
- Blogs: blog.yugabyte.com/author/bryn/



Bryn Llewellyn

Tell us about your present job, and how it relates to PostgreSQL

I came to PostgreSQL only relatively recently, in the spring of 2019, when I left my job in the PL/SQL Team at Oracle HQ to join Yugabyte Inc. This is an exciting Silicon Valley startup, one of whose founders had been a close colleague of mine in the PL/SQL team. My first blog post in my new job, Why I Moved from Oracle to YugaByte, explains how I was easily persuaded to make this change after close to thirty years with Oracle.

yugabyteDB

Google for:

"PostgreSQL Person of the Week" Bryn

- 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 as long ago as the nineteen-sixties
- 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

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

“Date and time data types” in the YugabyteDB documentation:

docs.yugabyte.com/latest/api/ysql/datatypes/type_datetime/

APIs > YSQL > Data types >

Date and time data types

Synopsis

YSQL supports the following data types for values that represent a date, a time of day, a date-and-time-of-day pair, or a duration. These data types will be referred to jointly as the *date-time* data types.

DATA TYPE	PURPOSE	INTERNAL FORMAT	MIN	MAX	RESOLUTION
date	date moment (wall-clock)	4-bytes	4713 BC	5874897 AD	1 day
time [(p)]	time moment (wall-clock)	8-bytes	00:00:00	24:00:00	1 microsecond
timetz [(p)]	*avoid this*				
timestamp [(p)]	date-and-time moment (wall-clock)	12-bytes	4713 BC	294276 AD	1 microsecond
timestamptz [(p)]	date-and-time moment (absolute)	12-bytes	4713 BC	294276 AD	1 microsecond
interval [fields] [(p)]	duration between two moments	16-bytes 3-field struct			1 microsecond

Companion downloadable code:

docs.yugabyte.com/latest/api/ysql/datatypes/type_datetime/download-date-time-utilities/

Each section divider has link(s) to the relevant subsection(s) in the YugabyteDB documentation

The main point

- 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

Timezones

Timezones and UTC offsets

- People have always organized their lives by the the sun, so you can't get away from timezones — despite the PRC's one-timezone policy

- *pg_timezone_names* has 593 rows (in YSQL on PG 11.2)

- See the doc section "The *extended_timezone_names* view"

- Leads to the *canonical_real_country_no_dst* and *canonical_real_country_with_dst* views

  39 distinct UTC offsets across these two views

```
-- 01.sql
with
  c1 as (
    select utc_offset as "Offset"
    from canonical_real_country_no_dst
    union
    select std_offset as "Offset"
    from canonical_real_country_with_dst
    union
    select dst_offset as "Offset"
    from canonical_real_country_with_dst),
  c2 as (
    select interval_mm_dd_ss("Offset") as x
    from c1)
select to_char((c2.x).ss/3600.0, '09.99') as "Offset"
from c2
order by (c2.x);
```

```
-11.00
-10.00
-09.50
-09.00
-08.00
-07.00
-06.00
-05.00
-04.00
-03.50
 ...
 09.00
 09.50
 10.00
 10.50
 11.00
 12.00
 12.75
 13.00
 13.75
 14.00
```

- **39 distinct UTC offsets**
- **Recommendation:**
  use only timezones listed in *canonical_real_country_no_dst* and *canonical_real_country_with_dst*
- **Else:**
  Katmandu* vs *Kathmandu

Setting the timezone

Three syntax contexts that use the specification of a UTC offset
Four ways to specify the UTC offset
Recommended practice for specifying the UTC offset

Three syntax contexts that use the specification of a UTC offset

- The three contexts:
  - Using the session environment parameter TimeZone
  - Using the *at time zone* operator — or its alternative function form *timezone()*
  - Within the text of a *timestamptz* literal or for *make_timestamptz()*'s *timezone* parameter
- You can specify the UTC offset *either* indirectly by name or directly as an *interval* value

The problem – 1

```
-- 02.sql
select name, utc_offset
from pg_timezone_names
where name in ('Etc/GMT-8', 'Etc/GMT-99');
```

Result:

```
   name     | utc_offset
------------+------------
 Etc/GMT-8  | 08:00:00
```

But this causes no error!

```
set timezone = 'Etc/GMT-99';
select ('2021-06-15 12:00:00 UTC'::timestamptz)::text;
```

Result:

```
2021-06-19 15:00:00+99
```

What's going on?

```
-- 02.sql (cont.)
select name, std_offset, dst_offset
from extended_timezone_names
where name in ('Etc/GMT-8', 'America/Los_Angeles')
order by name;
```

Result:

```
       name          | std_offset | dst_offset
---------------------+------------+------------
America/Los_Angeles | -08:00:00  | -07:00:00
Etc/GMT-8           |  08:00:00  |  08:00:00
```

There are two different conventions at work: "conventional" and POSIX
Result = confusion

POSIX brings no ultimate benefit in current PostgreSQL — so avoid it

The problem – 2

```
-- 03.sql
set timezone = 'UTC';

with c as (
  select '2021-01-01 01:00:00'::timestamp as t)
select
  timezone('America/Los_Angeles',     t)::text as t1,
  timezone(make_interval(hours=>-8), t)::text as t2,
  timezone('Etc/GMT-8',               t)::text as t3,
  timezone('Foo-8',                   t)::text as t4
from c;
```

Result:

```
t1 | 2021-01-01 09:00:00+00
t2 | 2021-01-01 09:00:00+00
t3 | 2020-12-31 17:00:00+00
t4 | 2020-12-31 17:00:00+00
```

The problem – 3

```
-- 03.sql (cont.)
set timezone = 'UTC';
with c as (
  select '2021-01-01 01:00:00'::text as t)
select
  (t||' America/Los_Angeles')::timestamptz as t1,
  (t||' Foo-8'               )::timestamptz as t2
from c;
```

Result:

```
t1 | 2021-01-01 09:00:00+00
t2 | 2020-12-31 17:00:00+00
```

Recommended practice for specifying the UTC offset

- See the doc section with this title (search the ToC). Create two user-defined function overload sets, thus:
  - *set_timezone()*, with *(text)* and *(interval)* overloads
  - *at_timezone()* with *(text, timestamp)*, *(interval, timestamp)*, *(text, timestamptz)*, *(interval, timestamptz)* overloads
- Use these wrappers instead of the “raw” native functionality
- The practice ensures that only approved arguments are used

Example

```
-- 04.sql
deallocate all;
prepare stmt as
with c as (
  select '2021-01-01 01:00:00'::timestamp as t)
select
  at_timezone('America/Los_Angeles',    t)::text as t1,
  at_timezone(make_interval(hours=>-8), t)::text as t2
from c;

call set_timezone('America/New_York');
execute stmt;
```

Result:

```
t1 | 2021-01-01 04:00:00-05
t2 | 2021-01-01 04:00:00-05
```

Next:

```
call set_timezone(make_interval(hours=>-5));
execute stmt;
```

Result is the same

Counter-examples

```
-- 04.sql (cont.)
call set_timezone('Etc/GMT-99');
```

Causes error:

```
ERROR:  22023: Invalid value for parameter TimeZone "Etc/GMT-99"
HINT:  Use a name that's found exactly once in "approved_timezone_names"
```

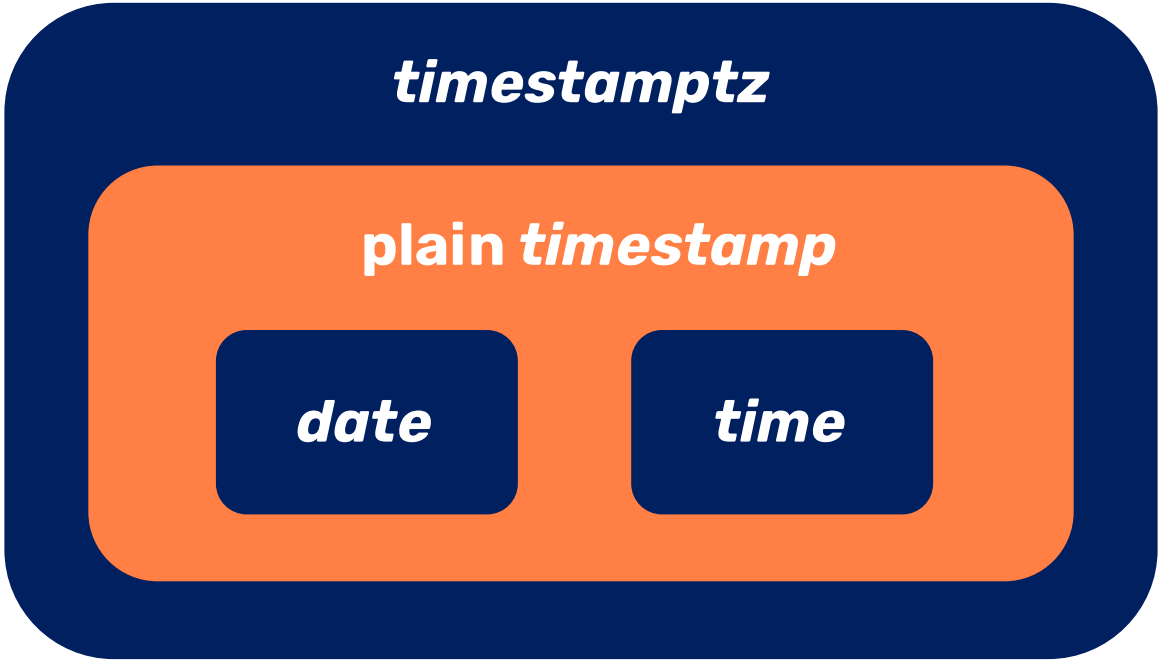
Next:

```
with c as (
  select '2021-01-01 01:00:00'::timestamp as t)
select
  at_timezone('Etc/GMT-99', t)::text as t1
from c;
```

Causes exactly the same error

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**

Example

```
-- 05.sql
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);
```

Example – cont.

```
-- 05.sql (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
```

Example – cont.

```
-- 05.sql (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
```

The timezone-sensitivity of the conversion of a *timestamptz* value to a *text* value

Sensitivity of converting between timestamptz and plain timestamp to the UTC offset

A contrived app for creating and viewing meetings

```
-- 06.sql
-- This is done when the app is installed.
drop table if exists meetings cascade;
create table meetings(k int primary key, t timestamptz);

deallocate all;

prepare cr_mtg(int, text, int, text) as
insert into meetings (k, t) values
  ($1, $2::timestamptz),
  ($3, $4::timestamptz);

prepare qry_mtg as
select
  k                                                                as "Mtg",
  to_char(t, 'Dy hh24-mi on dd-Mon-yyyy TZ ["with offset" TZH:TZM]') as "When"
from meetings
order by k;
```

Rickie lives in LA, adds two meetings, and views the result

```
-- 06.sql (cont.)

-- She has the timezone 'Europe/Amsterdam' set in her calendar preferences.
-- Notice that Daylight Savings Time starts, in LA,
-- on Sunday 14-Mar-2021
set timezone = 'America/Los_Angeles';
execute cr_mtg(
  1, '2021-03-09 08:00',
  2, '2021-03-16 08:00');

execute qry_mtg;
```

Result:

```
 Mtg |                          When
-----+-----------------------------------------------------
   1 | Tue 08-00 on 09-Mar-2021 PST [with offset -08:00]
   2 | Tue 08-00 on 16-Mar-2021 PDT [with offset -07:00]
```

Vincent lives in Amsterdam. He views his meetings

```
-- 06.sql (cont.)

-- He has the timezone 'Europe/Amsterdam' set in his calendar preferences.
-- Notice that Daylight Savings Time starts, in Amsterdam,
-- on Sunday 28-Mar-2021

set timezone = 'Europe/Amsterdam';
execute qry_mtg;
```

Result:

```
 Mtg |                          When
-----+-----------------------------------------------------
   1 | Tue 17-00 on 09-Mar-2021 CET [with offset +01:00]
   2 | Tue 16-00 on 16-Mar-2021 CET [with offset +01:00]
```

The result of *date_value('epoch', timestamptz_value)* is not affected by the session timezone

```
-- 07.sql
deallocate all;
prepare stmt as
select date_part('epoch', '2021-06-15 12:00:00 UTC'::timestamptz) as epoch;

call set_timezone('America/Los_Angeles');
execute stmt;

call set_timezone('Asia/Kathmandu');
execute stmt;
```

The result, both in 'America/Los_Angeles' and in 'Asia/Kathmandu' is the same:

```
1623758400
```

interval arithmetic

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

Example One

```
-- 08.sql
-- Helper function to show the internal representation of an interval value
drop function if exists i_repn(interval) cascade;

create function i_repn(i in interval)
  returns text
  language plpgsql
as $body$
declare
  repn constant interval_mm_dd_ss_t not null := interval_mm_dd_ss(i);
  mm constant text not null := (repn.mm)::text;
  dd constant text not null := (repn.dd)::text;
  ss constant text not null := (repn.ss)::text;
begin
  return '['||mm||', '||dd||', '||ss||']';
end;
$body$;
```

How does YSQL represent an interval value?

Example One — cont

```
-- 08.sql (cont.)
drop function if exists f(text) cascade;

create function f(timezone in text)
  returns table(z text)
  language plpgsql
as $body$
declare
  t constant timestamptz not null := at_timezone(timezone, '2021-03-10 12:00'::timestamp);
  i1 constant interval not null := make_interval(months=>1);
  i2 constant interval not null := make_interval(days=>30);
  i3 constant interval not null := make_interval(secs=>2592000);
begin
  z := 'i1:                    '||i1::text;                        return next;
  z := 'i2:                    '||i2::text;                        return next;
  z := 'i3:                    '||i3::text;                        return next;
  z := '';                                                         return next;

  z := 'i1_repn:               '||i_repn(i1);                      return next;
  z := 'i2_repn:               '||i_repn(i2);                      return next;
  z := 'i3_repn:               '||i_repn(i3);                      return next;
  z := '';                                                         return next;

  z := 'i2 = i1:               '||(i2 = i1)::text;                 return next;
  z := 'i3 = i1:               '||(i3 = i1)::text;                 return next;
  z := '';                                                         return next;

  z := 'i2 == i1:              '||(i2 == i1)::text;                return next;
  z := 'i3 == i1:              '||(i3 == i1)::text;                return next;
  z := '';                                                         return next;

  z := 't:                     '||t::text;                         return next;
  z := 't + i1:                '||(t + i1)::text;                  return next;
  z := 't + i2:                '||(t + i2)::text;                  return next;
  z := 't + i3:                '||(t + i3)::text;                  return next;
  z := '';                                                         return next;

  z := '(t + i1) = (t + i2):   '||((t + i1) = (t + i2))::text;     return next;
  z := '(t + i1) = (t + i3):   '||((t + i1) = (t + i3))::text;     return next;
  z := '(t + i2) = (t + i3):   '||((t + i2) = (t + i3))::text;     return next;
end;
$body$;
```

Example One — cont

```
-- 08.sql (cont.)
call set_timezone('America/Los_Angeles');
select z from f('America/Los_Angeles');

call set_timezone('Asia/Kathmandu');
select z from f('Asia/Kathmandu');
```

- The rules of *interval* arithmetic 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 !

Example Two

```
-- 09.sql
drop function if exists f() cascade;

create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  tz    constant text        not null := 'America/Los_Angeles';
  t1    constant timestamptz not null := at_timezone(tz, '2021-03-10 12:00'::timestamp);
  t2    constant timestamptz not null := at_timezone(tz, '2021-04-09 13:30'::timestamp);
  diff  constant interval    not null := t2 - t1;
  t3    constant timestamptz not null := t1 + diff;
  b     constant boolean     not null := t3 = t2;
begin
  z := 't1:                      '||t1::text;              return next;
  z := 't2:                      '||t2::text;              return next;
  z := 't2 - t1:                 '||i_repn(diff);          return next;
  z := 't1 + (t2 - t1):          '||t3::text;              return next;
  z := '';                                                 return next;
  z := '(t1 + (t2 - t1)) = t2:   '||b::text;               return next;
end;
$body$;
```

Example Two — cont.

```
-- 09.sql (cont.)
call set_timezone('America/Los_Angeles');
select z from f();
```

Result:

```
t1:                          2021-03-10 12:00:00-08
t2:                          2021-04-09 13:30:00-07
t2 - t1:                     [0, 30, 1800]
t1 + (t2 - t1):              2021-04-09 12:30:00-07

(t1 + (t2 - t1)) = t2:   false
```

Example Three

```
-- 10.sql
drop function if exists f() cascade;

create function f()
  returns table(z text)
  language plpgsql
as $body$
declare
  tz  constant text        not null := 'America/Los_Angeles';
  t1  constant timestamptz not null := at_timezone(tz, '2021-03-10 12:00'::timestamp);
  i1  constant interval    not null := '1 month'  ::interval;
  i2  constant interval    not null := '29 days'  ::interval;
  i3  constant interval    not null := '23:59:59' ::interval;

  t2  constant timestamptz not null := ((t1 + i1) + i2) + i3;
  t3  constant timestamptz not null := ((t1 + i3) + i2) + i1;

  b   constant boolean     not null := t2 = t3;
begin
  z := 't1:                                                  '||t1::text;      return next;
  z := '((t1 + i1) + i2) + i3:                               '||t2::text;      return next;
  z := '((t1 + i3) + i2) + i1:                               '||t3::text;      return next;
  z:= '';                                                                      return next;
  z := '(((t1 + i1) + i2) + i3) = (((t1 + i3) + i2) + i1):   '||b::text;       return next;
end;
$body$;
```

Example Three — cont.

```
-- 10.sql (cont.)
call set_timezone('America/Los_Angeles');
select z from f();
```

Result:

```
t1:                                                  2021-03-10 12:00:00-08
((t1 + i1) + i2) + i3:                               2021-05-10 12:00:28-07
((t1 + i3) + i2) + i1:                               2021-05-09 12:00:28-07

(((t1 + i1) + i2) + i3) = (((t1 + i3) + i2) + i1):   false
```

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

- See the doc section *"Custom domain types for specializing the native interval functionality"* (search the ToC).
- It ensures 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
- It provides 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

Example One — subtracting one *timestamptz* value from another

```
-- 11.sql
-- Uses the helper function i_repn(interval) from 08.sql
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 := i_repn(t2 - t1);
  i_mm       constant text        not null := i_repn(interval_months (t2, t1));
  i_dd       constant text        not null := i_repn(interval_days   (t2, t1));
  i_ss       constant text        not null := i_repn(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;
$body$;
```

Example One — cont

```
select z from f();
```

Result:

```
i_hybrid: [0, 333, 42955]
i_mm:     [10, 0, 0]
i_dd:     [0, 333, 0]
i_ss:     [0, 0, 28814155]
```

Example Two — multiplying a *timestamptz* value by a real number

```
select (interval_months(months=>99)*1.5432)::text;
select (interval_months(months=>99)*1.5432)::interval_months_t;
```

The second *select* causes the *23514* (*check_violation*) error:

```
value for domain interval_months_t violates check constraint "interval_months_t_check"
```

Next:

```
with c as (
  select interval_months(months=>99) as i_mm999)
select
  i_repn(i_mm999)                                    as "pure input",
  i_repn(i_mm999*1.5432)                             as "crazy hybrid result",
  i_repn(interval_months(i=>i_mm999, f=>1.5432))     as "sensible pure result"
from c;
```

Result:

```
 pure input | crazy hybrid result | sensible pure result
------------+---------------------+----------------------
 [99, 0, 0] | [152, 23, 26265.6]  | [153, 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**

  - **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

Questions?

Download
download.yugabyte.com

Join Slack Discussions
yugabyte.com/slack

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