9.9. Date/Time Functions and Operators

Table 9.33 shows the available functions for date/time value processing, with details appearing in the following subsections. Table 9.32 illustrates the behaviors of the basic arithmetic operators (`+`, `*`, etc.). For formatting functions, refer to Section 9.8. You should be familiar with the background information on date/time data types from Section 8.5.

In addition, the usual comparison operators shown in Table 9.1 are available for the date/time types. Dates and timestamps (with or without time zone) are all comparable, while times (with or without time zone) and intervals can only be compared to other values of the same data type. When comparing a timestamp without time zone to a timestamp with time zone, the former value is assumed to be given in the time zone specified by the TimeZone configuration parameter, and is rotated to UTC for comparison to the latter value (which is already in UTC internally). Similarly, a date value is assumed to represent midnight in the `TimeZone` zone when comparing it to a timestamp.

All the functions and operators described below that take `time` or `timestamp` inputs actually come in two variants: one that takes `time with time zone` or `timestamp with time zone`, and one that takes `time without time zone` or `timestamp without time zone`. For brevity, these variants are not shown separately. Also, the `+` and `*` operators come in commutative pairs (for example both `date` + `integer` and `integer` + `date`); we show only one of each such pair.

Table 9.32. Date/Time Operators

Operator	Description	Example(s)
`date + integer → date`	Add a number of days to a date	`date '2001-09-28' + 7 → 2001-10-05`
`date + interval → timestamp`	Add an interval to a date	`date '2001-09-28' + interval '1 hour' → 2001-09-28 01:00:00`
`date + time → timestamp`	Add a time-of-day to a date	`date '2001-09-28' + time '03:00' → 2001-09-28 03:00:00`
`interval + interval → interval`	Add intervals	`interval '1 day' + interval '1 hour' → 1 day 01:00:00`
`timestamp + interval → timestamp`	Add an interval to a timestamp	`timestamp '2001-09-28 01:00' + interval '23 hours' → 2001-09-29 00:00:00`
`time + interval → time`	Add an interval to a time	`time '01:00' + interval '3 hours' → 04:00:00`
`- interval → interval`	Negate an interval	`- interval '23 hours' → -23:00:00`
`date - date → integer`	Subtract dates, producing the number of days elapsed	`date '2001-10-01' - date '2001-09-28' → 3`
`date - integer → date`	Subtract a number of days from a date	`date '2001-10-01' - 7 → 2001-09-24`
`date - interval → timestamp`	Subtract an interval from a date	`date '2001-09-28' - interval '1 hour' → 2001-09-27 23:00:00`
`time - time → interval`	Subtract times	`time '05:00' - time '03:00' → 02:00:00`
`time - interval → time`	Subtract an interval from a time	`time '05:00' - interval '2 hours' → 03:00:00`
`timestamp - interval → timestamp`	Subtract an interval from a timestamp	`timestamp '2001-09-28 23:00' - interval '23 hours' → 2001-09-28 00:00:00`
`interval - interval → interval`	Subtract intervals	`interval '1 day' - interval '1 hour' → 1 day -01:00:00`
`timestamp - timestamp → interval`	Subtract timestamps (converting 24-hour intervals into days, similarly to `justify_hours()`)	`timestamp '2001-09-29 03:00' - timestamp '2001-07-27 12:00' → 63 days 15:00:00`

`interval * double precision → interval`	Multiply an interval by a scalar	`interval '1 second' * 900 → 00:15:00`
		`interval '1 day' * 21 → 21 days`
		`interval '1 hour' * 3.5 → 03:30:00`

`interval / double precision → interval`	Divide an interval by a scalar	`interval '1 hour' / 1.5 → 00:40:00`

Table 9.33. Date/Time Functions

Function	Description	Example(s)
`age ( timestamp, timestamp ) → interval`	Subtract arguments, producing a “symbolic” result that uses years and months, rather than just days	`age(timestamp '2001-04-10', timestamp '1957-06-13') → 43 years 9 mons 27 days`
`age ( timestamp ) → interval`	Subtract argument from `current_date` (at midnight)	`age(timestamp '1957-06-13') → 62 years 6 mons 10 days`
`clock_timestamp ( ) → timestamp with time zone`	Current date and time (changes during statement execution); see Section 9.9.5	`clock_timestamp() → 2019-12-23 14:39:53.662522-05`
`current_date → date`	Current date; see Section 9.9.5	`current_date → 2019-12-23`
`current_time → time with time zone`	Current time of day; see Section 9.9.5	`current_time → 14:39:53.662522-05`
`current_time ( integer ) → time with time zone`	Current time of day, with limited precision; see Section 9.9.5	

Function	Description	Example(s)
		`current_time(2)` → `14:39:53.66-05`
`current_timestamp` → `timestamp with time zone`	Current date and time (start of current transaction); see Section 9.9.5	`current_timestamp` → `2019-12-23 14:39:53.662522-05`
`current_timestamp` (`integer`) → `timestamp with time zone`	Current date and time (start of current transaction), with limited precision; see Section 9.9.5	`current_timestamp(0)` → `2019-12-23 14:39:53-05`
`date_bin` (`interval`, `timestamp`, `timestamp`) → `timestamp`	Bin input into specified interval aligned with specified origin; see Section 9.9.3	`date_bin('15 minutes', timestamp '2001-02-16 20:38:40', timestamp '2001-02-16 20:05:00')` → `2001-02-16 20:35:00`
`date_part` (`text`, `timestamp`) → `double precision`	Get timestamp subfield (equivalent to `extract`); see Section 9.9.1	`date_part('hour', timestamp '2001-02-16 20:38:40')` → `20`
`date_part` (`text`, `interval`) → `double precision`	Get interval subfield (equivalent to `extract`); see Section 9.9.1	`date_part('month', interval '2 years 3 months')` → `3`
`date_trunc` (`text`, `timestamp`) → `timestamp`	Truncate to specified precision; see Section 9.9.2	`date_trunc('hour', timestamp '2001-02-16 20:38:40')` → `2001-02-16 20:00:00`
`date_trunc` (`text`, `timestamp with time zone`, `text`) → `timestamp with time zone`	Truncate to specified precision in the specified time zone; see Section 9.9.2	`date_trunc('day', timestamptz '2001-02-16 20:38:40+00', 'Australia/Sydney')` → `2001-02-16 13:00:00+00`
`date_trunc` (`text`, `interval`) → `interval`	Truncate to specified precision; see Section 9.9.2	`date_trunc('hour', interval '2 days 3 hours 40 minutes')` → `2 days 03:00:00`
`extract` (*`field`* `from` `timestamp`) → `numeric`	Get timestamp subfield; see Section 9.9.1	`extract(hour from timestamp '2001-02-16 20:38:40')` → `20`
`extract` (*`field`* `from` `interval`) → `numeric`	Get interval subfield; see Section 9.9.1	`extract(month from interval '2 years 3 months')` → `3`
`isfinite` (`date`) → `boolean`	Test for finite date (not +/-infinity)	`isfinite(date '2001-02-16')` → `true`
`isfinite` (`timestamp`) → `boolean`	Test for finite timestamp (not +/-infinity)	`isfinite(timestamp 'infinity')` → `false`
`isfinite` (`interval`) → `boolean`	Test for finite interval (currently always true)	`isfinite(interval '4 hours')` → `true`
`justify_days` (`interval`) → `interval`	Adjust interval so 30-day time periods are represented as months	`justify_days(interval '35 days')` → `1 mon 5 days`
`justify_hours` (`interval`) → `interval`	Adjust interval so 24-hour time periods are represented as days	`justify_hours(interval '27 hours')` → `1 day 03:00:00`
`justify_interval` (`interval`) → `interval`	Adjust interval using `justify_days` and `justify_hours`, with additional sign adjustments	`justify_interval(interval '1 mon -1 hour')` → `29 days 23:00:00`
`localtime` → `time`	Current time of day; see Section 9.9.5	`localtime` → `14:39:53.662522`
`localtime` (`integer`) → `time`	Current time of day, with limited precision; see Section 9.9.5	`localtime(0)` → `14:39:53`
`localtimestamp` → `timestamp`	Current date and time (start of current transaction); see Section 9.9.5	`localtimestamp` → `2019-12-23 14:39:53.662522`
`localtimestamp` (`integer`) → `timestamp`	Current date and time (start of current transaction), with limited precision; see Section 9.9.5	`localtimestamp(2)` → `2019-12-23 14:39:53.66`
`make_date` (*`year`* `int`, *`month`* `int`, *`day`* `int`) → `date`	Create date from year, month and day fields (negative years signify BC)	`make_date(2013, 7, 15)` → `2013-07-15`
`make_interval` ([*`years`* `int` [, *`months`* `int` [, *`weeks`* `int` [, *`days`* `int` [, *`hours`* `int` [, *`mins`* `int` [, *`secs`* `double precision`]]]]]]]) → `interval`	Create interval from years, months, weeks, days, hours, minutes and seconds fields, each of which can default to zero	`make_interval(days => 10)` → `10 days`
`make_time` (*`hour`* `int`, *`min`* `int`, *`sec`* `double precision`) → `time`	Create time from hour, minute and seconds fields	`make_time(8, 15, 23.5)` → `08:15:23.5`
`make_timestamp` (*`year`* `int`, *`month`* `int`, *`day`* `int`, *`hour`* `int`, *`min`* `int`, *`sec`* `double precision`) → `timestamp`	Create timestamp from year, month, day, hour, minute and seconds fields (negative years signify BC)	`make_timestamp(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5`
`make_timestamptz` (*`year`* `int`, *`month`* `int`, *`day`* `int`, *`hour`* `int`, *`min`* `int`, *`sec`* `double precision` [, *`timezone`* `text`]) → `timestamp with time zone`	Create timestamp with time zone from year, month, day, hour, minute and seconds fields (negative years signify BC). If *`timezone`* is not specified, the current time zone is used; the examples assume the session time zone is `Europe/London`	`make_timestamptz(2013, 7, 15, 8, 15, 23.5)` → `2013-07-15 08:15:23.5+01`

Function Description Example(s)
`make_timestamptz(2013, 7, 15, 8, 15, 23.5, 'America/New_York')` → `2013-07-15 13:15:23.5+01`

`now ( )` → `timestamp with time zone`
Current date and time (start of current transaction); see Section 9.9.5
`now()` → `2019-12-23 14:39:53.662522-05`

`statement_timestamp ( )` → `timestamp with time zone`
Current date and time (start of current statement); see Section 9.9.5
`statement_timestamp()` → `2019-12-23 14:39:53.662522-05`

`timeofday ( )` → `text`
Current date and time (like `clock_timestamp`, but as a `text` string); see Section 9.9.5
`timeofday()` → `Mon Dec 23 14:39:53.662522 2019 EST`

`transaction_timestamp ( )` → `timestamp with time zone`
Current date and time (start of current transaction); see Section 9.9.5
`transaction_timestamp()` → `2019-12-23 14:39:53.662522-05`

`to_timestamp ( double precision )` → `timestamp with time zone`
Convert Unix epoch (seconds since 1970-01-01 00:00:00+00) to timestamp with time zone
`to_timestamp(1284352323)` → `2010-09-13 04:32:03+00`

In addition to these functions, the SQL `OVERLAPS` operator is supported:

```
(start1, end1) OVERLAPS (start2, end2)
(start1, length1) OVERLAPS (start2, length2)
```

This expression yields true when two time periods (defined by their endpoints) overlap, false when they do not overlap. The endpoints can be specified as pairs of dates, times, or time stamps; or as a date, time, or time stamp followed by an interval. When a pair of values is provided, either the start or the end can be written first; `OVERLAPS` automatically takes the earlier value of the pair as the start. Each time period is considered to represent the half-open interval ***`start`*** `<=` ***`time`*** `<` ***`end`***, unless ***`start`*** and ***`end`*** are equal in which case it represents that single time instant. This means for instance that two time periods with only an endpoint in common do not overlap.

```
SELECT (DATE '2001-02-16', DATE '2001-12-21') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: true
SELECT (DATE '2001-02-16', INTERVAL '100 days') OVERLAPS
       (DATE '2001-10-30', DATE '2002-10-30');
Result: false
SELECT (DATE '2001-10-29', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: false
SELECT (DATE '2001-10-30', DATE '2001-10-30') OVERLAPS
       (DATE '2001-10-30', DATE '2001-10-31');
Result: true
```

When adding an `interval` value to (or subtracting an `interval` value from) a `timestamp with time zone` value, the days component advances or decrements the date of the `timestamp with time zone` by the indicated number of days, keeping the time of day the same. Across daylight saving time changes (when the session time zone is set to a time zone that recognizes DST), this means `interval '1 day'` does not necessarily equal `interval '24 hours'`. For example, with the session time zone set to `America/Denver`:

```
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '1 day';
Result: 2005-04-03 12:00:00-06
SELECT timestamp with time zone '2005-04-02 12:00:00-07' + interval '24 hours';
Result: 2005-04-03 13:00:00-06
```

This happens because an hour was skipped due to a change in daylight saving time at `2005-04-03 02:00:00` in time zone `America/Denver`.

Note there can be ambiguity in the `months` field returned by `age` because different months have different numbers of days. PostgreSQL's approach uses the month from the earlier of the two dates when calculating partial months. For example, `age('2004-06-01', '2004-04-30')` uses April to yield `1 mon 1 day`, while using May would yield `1 mon 2 days` because May has 31 days, while April has only 30.

Subtraction of dates and timestamps can also be complex. One conceptually simple way to perform subtraction is to convert each value to a number of seconds using `EXTRACT(EPOCH FROM ...)`, then subtract the results; this produces the number of *seconds* between the two values. This will adjust for the number of days in each month, timezone changes, and daylight saving time adjustments. Subtraction of date or timestamp values with the “`-`” operator returns the number of days (24-hours) and hours/minutes/seconds between the values, making the same adjustments. The `age` function returns years, months, days, and hours/minutes/seconds, performing field-by-field subtraction and then adjusting for negative field values. The following queries illustrate the differences in these approaches. The sample results were produced with `timezone = 'US/Eastern'`; there is a daylight saving time change between the two dates used:

```
SELECT EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
       EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00');
Result: 10537200
SELECT (EXTRACT(EPOCH FROM timestamptz '2013-07-01 12:00:00') -
        EXTRACT(EPOCH FROM timestamptz '2013-03-01 12:00:00'))
        / 60 / 60 / 24;
Result: 121.9583333333333
SELECT timestamptz '2013-07-01 12:00:00' - timestamptz '2013-03-01 12:00:00';
Result: 121 days 23:00:00
SELECT age(timestamptz '2013-07-01 12:00:00', timestamptz '2013-03-01 12:00:00');
Result: 4 mons
```

9.9.1. EXTRACT, date_part

```
EXTRACT(field FROM source)
```

The `extract` function retrieves subfields such as year or hour from date/time values. ***source*** must be a value expression of type `timestamp`, `time`, or `interval`. (Expressions of type `date` are cast to `timestamp` and can therefore be used as well.) ***`field`*** is an identifier or string that selects what field to extract from the source value. The `extract` function returns values of type `numeric`. The following are valid field names:

`century`

The century

```
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2000-12-16 12:21:13');
Result: 20
SELECT EXTRACT(CENTURY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 21
```

The first century starts at 0001-01-01 00:00:00 AD, although they did not know it at the time. This definition applies to all Gregorian calendar countries. There is no century number 0, you go from -1 century to 1 century. If you disagree with this, please write your complaint to: Pope, Cathedral Saint-Peter of Roma, Vatican.

`day`

For `timestamp` values, the day (of the month) field (1–31) ; for `interval` values, the number of days

```
SELECT EXTRACT(DAY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT EXTRACT(DAY FROM INTERVAL '40 days 1 minute');
Result: 40
```

`decade`

The year field divided by 10

```
SELECT EXTRACT(DECADE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 200
```

`dow`

The day of the week as Sunday (`0`) to Saturday (`6`)

```
SELECT EXTRACT(DOW FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 5
```

Note that `extract`'s day of the week numbering differs from that of the `to_char(..., 'D')` function.

`doy`

The day of the year (1–365/366)

```
SELECT EXTRACT(DOY FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 47
```

`epoch`

For `timestamp with time zone` values, the number of seconds since 1970-01-01 00:00:00 UTC (negative for timestamps before that); for `date` and `timestamp` values, the nominal number of seconds since 1970-01-01 00:00:00, without regard to timezone or daylight-savings rules; for `interval` values, the total number of seconds in the interval

```
SELECT EXTRACT(EPOCH FROM TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40.12-08'
Result: 982384720.12

SELECT EXTRACT(EPOCH FROM TIMESTAMP '2001-02-16 20:38:40.12');
Result: 982355920.12

SELECT EXTRACT(EPOCH FROM INTERVAL '5 days 3 hours');
Result: 442800
```

You can convert an epoch value back to a `timestamp with time zone` with `to_timestamp`:

```
SELECT to_timestamp(982384720.12);
Result: 2001-02-17 04:38:40.12+00
```

Beware that applying `to_timestamp` to an epoch extracted from a `date` or `timestamp` value could produce a misleading result: the result will effectively assume that the original value had been given in UTC, which might not be the case.

`hour`

The hour field (0–23)

```
SELECT EXTRACT(HOUR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 20
```

`isodow`

The day of the week as Monday (1) to Sunday (7)

```
SELECT EXTRACT(ISODOW FROM TIMESTAMP '2001-02-18 20:38:40');
Result: 7
```

This is identical to `dow` except for Sunday. This matches the ISO 8601 day of the week numbering.

`isoyear`

The ISO 8601 week-numbering year that the date falls in (not applicable to intervals)

```
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-01');
Result: 2005
SELECT EXTRACT(ISOYEAR FROM DATE '2006-01-02');
Result: 2006
```

Each ISO 8601 week-numbering year begins with the Monday of the week containing the 4th of January, so in early January or late December the ISO year may be different from the Gregorian year. See the `week` field for more information.

This field is not available in PostgreSQL releases prior to 8.3.

`julian`

The *Julian Date* corresponding to the date or timestamp (not applicable to intervals). Timestamps that are not local midnight result in a fractional value. See Section B.7 for more information.

```
SELECT EXTRACT(JULIAN FROM DATE '2006-01-01');
Result: 2453737
SELECT EXTRACT(JULIAN FROM TIMESTAMP '2006-01-01 12:00');
Result: 2453737.50000000000000000000
```

`microseconds`

The seconds field, including fractional parts, multiplied by 1 000 000; note that this includes full seconds

```
SELECT EXTRACT(MICROSECONDS FROM TIME '17:12:28.5');
Result: 28500000
```

`millennium`

The millennium

```
SELECT EXTRACT(MILLENNIUM FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 3
```

Years in the 1900s are in the second millennium. The third millennium started January 1, 2001.

`milliseconds`

The seconds field, including fractional parts, multiplied by 1000. Note that this includes full seconds.

```
SELECT EXTRACT(MILLISECONDS FROM TIME '17:12:28.5');
Result: 28500
```

`minute`

The minutes field (0–59)

```
SELECT EXTRACT(MINUTE FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 38
```

`month`

For `timestamp` values, the number of the month within the year (1–12) ; for `interval` values, the number of months, modulo 12 (0–11)

```
SELECT EXTRACT(MONTH FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 3 months');
Result: 3

SELECT EXTRACT(MONTH FROM INTERVAL '2 years 13 months');
Result: 1
```

`quarter`

The quarter of the year (1–4) that the date is in

```
SELECT EXTRACT(QUARTER FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 1
```

`second`

The seconds field, including any fractional seconds

```
SELECT EXTRACT(SECOND FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 40

SELECT EXTRACT(SECOND FROM TIME '17:12:28.5');
Result: 28.5
```

`timezone`

The time zone offset from UTC, measured in seconds. Positive values correspond to time zones east of UTC, negative values to zones west of UTC. (Technically, PostgreSQL does not use UTC because leap seconds are not handled.)

`timezone_hour`

The hour component of the time zone offset

`timezone_minute`

The minute component of the time zone offset

`week`

The number of the ISO 8601 week-numbering week of the year. By definition, ISO weeks start on Mondays and the first week of a year contains January 4 of that year. In other words, the first Thursday of a year is in week 1 of that year.

In the ISO week-numbering system, it is possible for early-January dates to be part of the 52nd or 53rd week of the previous year, and for late-December dates to be part of the first week of the next year. For example, `2005-01-01` is part of the 53rd week of year 2004, and `2006-01-01` is part of the 52nd week of year 2005, while `2012-12-31` is part of the first week of 2013. It's recommended to use the `isoyear` field together with `week` to get consistent results.

```
SELECT EXTRACT(WEEK FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 7
```

`year`

The year field. Keep in mind there is no `0 AD`, so subtracting `BC` years from `AD` years should be done with care.

```
SELECT EXTRACT(YEAR FROM TIMESTAMP '2001-02-16 20:38:40');
Result: 2001
```

> **Note**
>
> When the input value is +/-Infinity, `extract` returns +/-Infinity for monotonically-increasing fields (`epoch`, `julian`, `year`, `isoyear`, `decade`, `century`, and `millennium`). For other fields, NULL is returned. PostgreSQL versions before 9.6 returned zero for all cases of infinite input.

The `extract` function is primarily intended for computational processing. For formatting date/time values for display, see Section 9.8.

The `date_part` function is modeled on the traditional Ingres equivalent to the SQL-standard function `extract`:

```
date_part('field', source)
```

Note that here the ***`field`*** parameter needs to be a string value, not a name. The valid field names for `date_part` are the same as for `extract`. For historical reasons, the `date_part` function returns values of type `double precision`. This can result in a loss of precision in certain uses. Using `extract` is recommended instead.

```
SELECT date_part('day', TIMESTAMP '2001-02-16 20:38:40');
Result: 16

SELECT date_part('hour', INTERVAL '4 hours 3 minutes');
Result: 4
```

9.9.2. date_trunc

The function `date_trunc` is conceptually similar to the `trunc` function for numbers.

```
date_trunc(field, source [, time_zone ])
```

source is a value expression of type `timestamp`, `timestamp with time zone`, or `interval`. (Values of type `date` and `time` are cast automatically to `timestamp` or `interval`, respectively.) ***field*** selects to which precision to truncate the input value. The return value is likewise of type `timestamp`, `timestamp with time zone`, or `interval`, and it has all fields that are less significant than the selected one set to zero (or one, for day and month).

Valid values for ***field*** are:

`microseconds`
`milliseconds`
`second`
`minute`
`hour`
`day`
`week`
`month`
`quarter`
`year`
`decade`
`century`
`millennium`

When the input value is of type `timestamp with time zone`, the truncation is performed with respect to a particular time zone; for example, truncation to `day` produces a value that is midnight in that zone. By default, truncation is done with respect to the current TimeZone setting, but the optional ***time_zone*** argument can be provided to specify a different time zone. The time zone name can be specified in any of the ways described in Section 8.5.3.

A time zone cannot be specified when processing `timestamp without time zone` or `interval` inputs. These are always taken at face value.

Examples (assuming the local time zone is `America/New_York`):

```
SELECT date_trunc('hour', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-02-16 20:00:00

SELECT date_trunc('year', TIMESTAMP '2001-02-16 20:38:40');
Result: 2001-01-01 00:00:00

SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00');
Result: 2001-02-16 00:00:00-05

SELECT date_trunc('day', TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40+00', 'Austr
Result: 2001-02-16 08:00:00-05

SELECT date_trunc('hour', INTERVAL '3 days 02:47:33');
Result: 3 days 02:00:00
```

9.9.3. `date_bin`

The function `date_bin` "bins" the input timestamp into the specified interval (the *stride*) aligned with a specified origin.

```
date_bin(stride, source, origin)
```

`source` is a value expression of type `timestamp` or `timestamp with time zone`. (Values of type `date` are cast automatically to `timestamp`.) ***`stride`*** is a value expression of type `interval`. The return value is likewise of type `timestamp` or `timestamp with time zone`, and it marks the beginning of the bin into which the ***`source`*** is placed.

Examples:

```
SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP '2001-01-
Result: 2020-02-11 15:30:00

SELECT date_bin('15 minutes', TIMESTAMP '2020-02-11 15:44:17', TIMESTAMP '2001-01-
Result: 2020-02-11 15:32:30
```

In the case of full units (1 minute, 1 hour, etc.), it gives the same result as the analogous `date_trunc` call, but the difference is that `date_bin` can truncate to an arbitrary interval.

The *`stride`* interval must be greater than zero and cannot contain units of month or larger.

9.9.4. `AT TIME ZONE`

The `AT TIME ZONE` operator converts time stamp *without* time zone to/from time stamp *with* time zone, and `time with time zone` values to different time zones. Table 9.34 shows its variants.

Table 9.34. `AT TIME ZONE` Variants

Operator Description Example(s)

`timestamp without time zone` AT TIME ZONE ***zone*** → `timestamp with time zone`
Converts given time stamp *without* time zone to time stamp *with* time zone, assuming the given value is in the named time zone.
`timestamp '2001-02-16 20:38:40' at time zone 'America/Denver'` → `2001-02-17 03:38:40+00`

`timestamp with time zone` AT TIME ZONE ***zone*** → `timestamp without time zone`
Converts given time stamp *with* time zone to time stamp *without* time zone, as the time would appear in that zone.
`timestamp with time zone '2001-02-16 20:38:40-05' at time zone 'America/Denver'` → `2001-02-16 18:38:40`

`time with time zone` AT TIME ZONE ***zone*** → `time with time zone`
Converts given time *with* time zone to a new time zone. Since no date is supplied, this uses the currently active UTC offset for the named destination zone.
`time with time zone '05:34:17-05' at time zone 'UTC'` → `10:34:17+00`

In these expressions, the desired time zone ***`zone`*** can be specified either as a text value (e.g., `'America/Los_Angeles'`) or as an interval (e.g., `INTERVAL '-08:00'`). In the text case, a time zone name can be specified in any of the ways described in Section 8.5.3. The interval case is only useful for zones that have fixed offsets from UTC, so it is not very common in practice.

Examples (assuming the current TimeZone setting is `America/Los_Angeles`):

```
SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'America/Denver';
Result: 2001-02-16 19:38:40-08

SELECT TIMESTAMP WITH TIME ZONE '2001-02-16 20:38:40-05' AT TIME ZONE 'America/Der
Result: 2001-02-16 18:38:40

SELECT TIMESTAMP '2001-02-16 20:38:40' AT TIME ZONE 'Asia/Tokyo' AT TIME ZONE 'Ame
Result: 2001-02-16 05:38:40
```

The first example adds a time zone to a value that lacks it, and displays the value using the current `TimeZone` setting. The second example shifts the time stamp with time zone value to the specified time zone, and returns the value without a time zone. This allows storage and display of values different from the current `TimeZone` setting. The third example converts Tokyo time to Chicago time.

The function `timezone(`***`zone`***`, `***`timestamp`***`)` is equivalent to the SQL-conforming construct ***`timestamp`***` AT TIME ZONE `***`zone`***.

9.9.5. Current Date/Time

PostgreSQL provides a number of functions that return values related to the current date and time. These SQL-standard functions all return values based on the start time of the current transaction:

```
CURRENT_DATE
CURRENT_TIME
CURRENT_TIMESTAMP
CURRENT_TIME(precision)
CURRENT_TIMESTAMP(precision)
LOCALTIME
LOCALTIMESTAMP
LOCALTIME(precision)
LOCALTIMESTAMP(precision)
```

`CURRENT_TIME` and `CURRENT_TIMESTAMP` deliver values with time zone; `LOCALTIME` and `LOCALTIMESTAMP` deliver values without time zone.

`CURRENT_TIME`, `CURRENT_TIMESTAMP`, `LOCALTIME`, and `LOCALTIMESTAMP` can optionally take a precision parameter, which causes the result to be rounded to that many fractional digits in the seconds field. Without a precision parameter, the result is given to the full available precision.

Some examples:

```
SELECT CURRENT_TIME;
Result: 14:39:53.662522-05

SELECT CURRENT_DATE;
Result: 2019-12-23

SELECT CURRENT_TIMESTAMP;
Result: 2019-12-23 14:39:53.662522-05

SELECT CURRENT_TIMESTAMP(2);
Result: 2019-12-23 14:39:53.66-05

SELECT LOCALTIMESTAMP;
Result: 2019-12-23 14:39:53.662522
```

Since these functions return the start time of the current transaction, their values do not change during the transaction. This is considered a feature: the intent is to allow a single transaction to have a consistent notion of the "current" time, so that multiple modifications within the same transaction bear the same time stamp.

> **Note**
>
> Other database systems might advance these values more frequently.

PostgreSQL also provides functions that return the start time of the current statement, as well as the actual current time at the instant the function is called. The complete list of non-SQL-standard time functions is:

```
transaction_timestamp()
statement_timestamp()
clock_timestamp()
timeofday()
now()
```

`transaction_timestamp()` is equivalent to `CURRENT_TIMESTAMP`, but is named to clearly reflect what it returns. `statement_timestamp()` returns the start time of the current statement (more specifically, the time of receipt of the latest command message from the client). `statement_timestamp()` and `transaction_timestamp()` return the same value during the first command of a transaction, but might differ during subsequent commands. `clock_timestamp()` returns the actual current time, and therefore its value changes even within a single SQL command. `timeofday()` is a historical PostgreSQL function. Like `clock_timestamp()`, it returns the actual current time, but as a formatted `text` string rather than a `timestamp with time zone` value. `now()` is a traditional PostgreSQL equivalent to `transaction_timestamp()`.

All the date/time data types also accept the special literal value `now` to specify the current date and time (again, interpreted as the transaction start time). Thus, the following three all return the same result:

```
SELECT CURRENT_TIMESTAMP;
SELECT now();
SELECT TIMESTAMP 'now';  -- but see tip below
```

Tip

Do not use the third form when specifying a value to be evaluated later, for example in a `DEFAULT` clause for a table column. The system will convert `now` to a `timestamp` as soon as the constant is parsed, so that when the default value is needed, the time of the table creation would be used! The first two forms will not be evaluated until the default value is used, because they are function calls. Thus they will give the desired behavior of defaulting to the time of row insertion. (See also Section 8.5.1.4.)

9.9.6. Delaying Execution

The following functions are available to delay execution of the server process:

```
pg_sleep ( double precision )
pg_sleep_for ( interval )
pg_sleep_until ( timestamp with time zone )
```

`pg_sleep` makes the current session's process sleep until the given number of seconds have elapsed. Fractional-second delays can be specified. `pg_sleep_for` is a convenience function to allow the sleep time to be specified as an `interval`. `pg_sleep_until` is a convenience function for when a specific wake-up time is desired. For example:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

Note

The effective resolution of the sleep interval is platform-specific; 0.01 seconds is a common value. The sleep delay will be at least as long as specified. It might be longer depending on factors such as server load. In particular, `pg_sleep_until` is not guaranteed to wake up exactly at the specified time, but it will not wake up any earlier.

Warning

Make sure that your session does not hold more locks than necessary when calling `pg_sleep` or its variants. Otherwise other sessions might have to wait for your sleeping process, slowing down the entire system.

Prev	Up	Next
9.8. Data Type Formatting Functions	Home	9.10. Enum Support Functions

Submit correction

If you see anything in the documentation that is not correct, does not match your experience with the particular feature or requires further clarification, please use this form to report a documentation issue.