MobilityDB 1.0 User's Manual

COLLABORATORS			
	TITLE : MobilityDB 1.0 User's Manual		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Esteban Zimányi	May 8, 2021	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

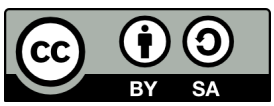
Contents

List of Figures

Abstract

MobilityDB is an extension to the PostgreSQL database system and its spatial extension PostGIS. It allows temporal and spatio-temporal objects to be stored in the database, that is, objects whose attribute values and/or location evolves in time. MobilityDB includes functions for analysis and processing of temporal and spatio-temporal objects and provides support for GiST and SP-GiST indexes. MobilityDB is open source and its code is available on Github. An adapter for the Python programming language is also available on Github.

MobilityDB is developed by the Computer & Decision Engineering Department of the Université Libre de Bruxelles (ULB) under the direction of Prof. Esteban Zimányi. ULB is an OGC Associate Member and member of the OGC Moving Feature Standard Working Group (MF-SWG).



Chapter 1

Introduction

MobilityDB is an extension of PostgreSQL and PostGIS that provides *temporal types*. Such data types represent the evolution on time of values of some element type, called the *base type* of the temporal type. For instance, temporal integers may be used to represent the evolution on time of the number of employees of a department. In this case, the data type is *temporal integer* and the base type is *integer*. Similarly, a temporal float may be used to represent the evolution on time of the temperature of a room. As another example, a temporal point may be used to represent the evolution on time of the location of a car, as reported by GPS devices. Temporal types are useful because representing values that evolve in time is essential in many applications, for example in mobility applications. Furthermore, the operators on the base types (such as arithmetic operators and aggregation for integers and floats, spatial relationships and distance for geometries) can be intuitively generalized when the values evolve in time.

MobilityDB provides the following temporal types: `tbool`, `tint`, `tfloat`, `ttext`, `tgeompoint`, and `tgeogpoint`. These temporal types are based, respectively, on the `bool`, `int`, `float`, and `text` base types provided by PostgreSQL, and on the `geometry` and `geography` base types provided by PostGIS (restricted to 2D or 3D points).[1] Furthermore, MobilityDB uses four time types to represent extents of time: the `timestamptz` type provided by PostgreSQL and three new types which are `period`, `timestampset`, and `periodset`. In addition, two range types are defined in MobilityDB: `intrange` and `floatrange`.

1.1 Project Steering Committee

The MobilityDB Project Steering Committee (PSC) coordinates the general direction, release cycles, documentation, and outreach efforts for the MobilityDB project. In addition, the PSC provides general user support, accepts and approves patches from the general MobilityDB community and votes on miscellaneous issues involving MobilityDB such as developer commit access, new PSC members or significant API changes.

The current members in alphabetical order and their main responsibilities are given next:

- Mohamed Bakli: MobilityDB-docker, cloud and distributed versions, integration with Citus
- Krishna Chaitanya Bommakanti: MobilityDB SQLAlchemy, MEOS (Mobility Engine Open Source), pyMEOS
- Anita Graser: integration with Moving Pandas and the Python ecosystem, integration with QGIS
- Darafei Praliaskouski: integration with PostGIS
- Mahmoud Sakr: co-founder of the MobilityDB project, MobilityDB workshop, integration with pgRouting
- Esteban Zimányi (chair): co-founder of the MobilityDB project, overall project coordination, main contributor of the backend code, BerlinMOD benchmark, MobilityDB-python

[1] Although 4D temporal points can be represented, the M dimension is currently not taken into account.

1.2 Other Code Contributors

- Arthur Lesuisse
- Xinyiang Li
- Maxime Schoemans

1.3 Corporate Sponsors

These are corporate entities (in alphabetical order) that have contributed developer time or direct monetary funding to the MobilityDB project.

- Adonmo, India
- Innoviris, Belgium
- Université libre de Bruxelles, Belgium

1.4 Licensing

The following licenses can be found in MobilityDB:

Resource	Licence
MobilityDB code	PostgreSQL Licence
MobilityDB documentation	Creative Commons Attribution-Share Alike 3.0 License

1.5 Installation

1.5.1 Short Version

Extracting the tar ball

```
tar xvfz MobilityDB-1.0.tar.gz
cd MobilityDB-1.0
```

To compile assuming you have all the dependencies in your search path

```
mkdir build
cd build
cmake  ..
make
sudo make install
```

Once MobilityDB is installed, it needs to be enabled in each individual database you want to use it in.

```
createdb mobility
psql mobility -c 'CREATE EXTENSION PostGIS'
psql mobility -c 'CREATE EXTENSION MobilityDB'
```

1.5.2 Get the Sources

The MobilityDB latest release can be found in https://github.com/MobilityDB/MobilityDB/releases/latest

wget

To download this release:

```
wget -O mobilitydb-1.0.tar.gz https://github.com/MobilityDB/MobilityDB/archive/v1.0.tar.gz
```

Go to Section 1.5.1 to the extract and compile instructions.

git

To download the repository

```
git clone https://github.com/MobilityDB/MobilityDB.git
cd MobilityDB
git checkout v1.0
```

Go to Section 1.5.1 to the compile instructions (there is no tar ball).

1.5.3 Enabling the Database

MobilityDB is an extension that depends on PostGIS. Enabling PostGIS before enabling MobilityDB in the database can be done as follows

```
CREATE EXTENSION postgis;
CREATE EXTENSION mobilitydb;
```

Alternatively, this can be done in a single command by using `CASCADE`, which installs the required PostGIS extension before installing the MobilityDB extension

```
CREATE EXTENSION mobilitydb CASCADE;
```

1.5.4 Dependencies

Compilation Dependencies

To be able to compile MobilityDB, make sure that the following dependencies are met:

- GNU C compiler (`gcc`). Some other ANSI C compilers can be used but may cause problems compiling some dependencies such as PostGIS.
- GNU Make (`gmake` or `make`) version 3.1 or higher. For many systems, GNU make is the default version of make. Check the version by invoking `make -v`.
- PostgreSQL version 10 or higher. PostgreSQL is available from http://www.postgresql.org. Notice that for using SP-GiST indexes for MobilityDB you need at least PostgreSQL version 11.
- PostGIS version 2.5. PostGIS is available from https://postgis.net/. PostGIS version 3.0 or higher is currently **not supported**, this is planned for future releases of MobilityDB.
- GNU Scientific Library (GSL). GSL is available from https://www.gnu.org/software/gsl/. GSL is used for the random number generators.

Please notice that PostGIS has its own dependencies such as Proj4, GEOS, LibXML2, or JSON-C, and these libraries are also used in MobilityDB. For a full PostgreSQL/PostGIS support matrix and PostGIS/GEOS support matrix refer to http://trac.osgeo.org/postgis/wiki/UsersWikiPostgreSQLPostGIS.

Optional Dependencies

For user's documentation

- DocBook (`xsltproc`) is required for building the documentation. Docbook is available from http://www.docbook.org/
- DBLatex (`dblatex`) is required for building the documentation in PDF format. DBLatex is available from http://dblatex.sourceforge.n

Example: Installing dependencies on Linux

Database dependencies

```
sudo apt-get install postgresql-12 postgresql-server-dev-12 postgresql-12-postgis
```

Build dependencies

```
sudo apt-get install cmake gcc libgsl-dev
```

1.5.5 Configuring

MobilityDB uses the `cmake` system to do the configuration. The build directory must be different from the source directory.

To create the build directory

```
mkdir build
```

To see the variables that can be configured

```
cd build
cmake -L ..
```

1.5.6 Build and Install

Please notice that the current version of MobilityDB has only been tested on Linux systems. It may work on other UNIX-like systems, but remain untested. Support for Windows is planned. We are looking for volunteers to help us to test MobilityDB on multiple platforms.

The following instructions start from `path/to/MobilityDB` on a Linux system

```
mkdir build
cd build
cmake ..
make
sudo make install
```

When the configuration changes

```
rm -rf build
```

and start the build process as mentioned above.

1.5.7 Testing

MobilityDB uses `ctest`, the CMake test driver program, for testing. This program will run the tests and report results.

To run all the tests

```
ctest
```

To run a given test file

```
ctest -R '21_tbox'
```

To run a set of given test files you can use wildcards

```
ctest -R '22_*'
```

1.6 Support

MobilityDB community support is available through the MobilityDB github page, documentation, tutorials, mailing lists and others.

1.6.1 Reporting Problems

Bugs are reported and managed in an issue tracker. Please follow these steps:

1. Search the tickets to see if your problem has already been reported. If so, add any extra context you might have found, or at least indicate that you too are having the problem. This will help us prioritize common issues.
2. If your problem is unreported, create a new issue for it.
3. In your report include explicit instructions to replicate your issue. The best tickets include the exact SQL necessary to replicate a problem. Please also, note the operating system and versions of MobilityDB, PostGIS, and PostgreSQL.
4. It is recommended to use the following wrapper on the problem to pin point the step that is causing the problem.

```
SET client_min_messages TO debug;
<your code>
SET client_min_messages TO notice;
```

1.6.2 Mailing Lists

There are two mailing lists for MobilityDB hosted on OSGeo mailing list server:

- User mailing list: http://lists.osgeo.org/mailman/listinfo/mobilitydb-users
- Developer mailing list: http://lists.osgeo.org/mailman/listinfo/mobilitydb-dev

For general questions and topics about how to use MobilityDB, please write to the user mailing list.

Chapter 2

Time Types and Range Types

Temporal types are based on four time types: the `timestamptz` type provided by PostgreSQL and three new types which are `period`, `timestampset`, and `periodset`.

The `period` type is a specialized version of the `tstzrange` (short for timestamp with time zone range) type provided by PostgreSQL. Type `period` has similar functionality as type `tstzrange` but has a more efficient implementation, in particular it is of fixed length while the `tstzrange` type is of variable length. Furthermore, empty periods and infinite bounds are not allowed in `period` values, while they are allowed in `tstzrange` values.

A value of the `period` type has two bounds, the lower bound and the upper bound, which are `timestamptz` values. The bounds can be inclusive or exclusive. An inclusive bound means that the boundary instant is included in the period, while an exclusive bound means that the boundary instant is not included in the period. In the text form of a `period` value, inclusive and exclusive lower bounds are represented, respectively, by "`[`" and "`(`". Likewise, inclusive and exclusive upper bounds are represented, respectively, by "`]`" and "`)`". In a `period` value, the lower bound must be less than or equal to the upper bound. A `period` value with equal and inclusive bounds is called an *instantaneous period* and corresponds to a `timestamptz` value. Examples of `period` values are as follows:

```
SELECT period '[2012-01-01 08:00:00, 2012-01-03 09:30:00)';
-- Instant period
SELECT period '[2012-01-01 08:00:00, 2012-01-01 08:00:00]';
-- Erroneous period: invalid bounds
SELECT period '[2012-01-01 08:10:00, 2012-01-01 08:00:00]';
-- Erroneous period: empty period
SELECT period '[2012-01-01 08:00:00, 2012-01-01 08:00:00)';
```

The `timestampset` type represents a set of different `timestamptz` values. A `timestampset` value must contain at least one element, in which case it corresponds to a `timestamptz` value. The elements composing a `timestampset` value must be ordered. Examples of `timestampset` values are as follows:

```
SELECT timestampset '{2012-01-01 08:00:00, 2012-01-03 09:30:00}';
-- Singleton timestampset
SELECT timestampset '{2012-01-01 08:00:00}';
-- Erroneous timestampset: unordered elements
SELECT timestampset '{2012-01-01 08:10:00, 2012-01-01 08:00:00}';
-- Erroneous timestampset: duplicate elements
SELECT timestampset '{2012-01-01 08:00:00, 2012-01-01 08:00:00}';
```

Finally, the `periodset` type represents a set of disjoint `period` values. A `periodset` value must contain at least one element, in which case it corresponds to a `period` value. The elements composing a `periodset` value must be ordered. Examples of `periodset` values are as follows:

```
SELECT periodset '{[2012-01-01 08:00:00, 2012-01-01 08:10:00],
[2012-01-01 08:20:00, 2012-01-01 08:40:00]}';
-- Singleton periodset
SELECT periodset '{[2012-01-01 08:00:00, 2012-01-01 08:10:00]}';
-- Erroneous periodset: unordered elements
SELECT periodset '{[2012-01-01 08:20:00, 2012-01-01 08:40:00],
  [2012-01-01 08:00:00, 2012-01-01 08:10:00]}';
-- Erroneous periodset: overlapping elements
SELECT periodset '{[2012-01-01 08:00:00, 2012-01-01 08:10:00],
  [2012-01-01 08:05:00, 2012-01-01 08:15:00]}';
```

Values of the `periodset` type are converted into *normal form* so that equivalent values have identical representations. For this, consecutive adjacent period values are merged when possible. An example of transformation into normal form is as follows:

```
SELECT periodset '{[2012-01-01 08:00:00, 2012-01-01 08:10:00),
[2012-01-01 08:10:00, 2012-01-01 08:10:00], (2012-01-01 08:10:00, 2012-01-01 08:20:00]}';
-- "{[2012-01-01 08:00:00+00,2012-01-01 08:20:00+00]}"
```

Besides the built-in range types provided by PostgreSQL, MobilityDB defines two additional range types: `intrange` (another name for `int4range`) and `floatrange`.

2.1 Functions and Operators for Time Types and Range Types

We present next the functions and operators for time types. These functions and operators are polymorphic, that is, their arguments may be of several types, and the result type may depend on the type of the arguments. To express this in the signature of the operators, we use the following notation:

- A set of types such as `{period, timestampset, periodset}` represents any of the types listed,
- `time` represents any time type, that is, `timestamptz`, `period`, `timestampset`, or `periodset`,
- `number` represents any number type, that is, `int` or `float`,
- `range` represents any number range type, that is, `intrange` or `floatrange`.
- `type[]` represents an array of `type`.

As an example, the signature of the contains operator (`@>`) is as follows:

```
{timestampset, period, periodset} @> time
```

In the following, for conciseness, the time part of the timestamps is omitted in the examples. Recall that in that case PostgreSQL assumes the time `00:00:00`.

2.1.1 Constructor Functions

The `period` type has a constructor function that accepts two or four arguments. The two-argument form constructs a period in *normal form*, that is, with inclusive lower bound and exclusive upper bound. The four-argument form constructs a period with bounds specified by the third and fourth arguments, which are Boolean values stating, respectively, whether the left and right bounds are inclusive or not.

- Constructor for `period`

  `period(timestamptz, timestamptz, left_inc = true, right_inc = false):  period`

```
-- Period defined with two arguments
SELECT period('2012-01-01 08:00:00', '2012-01-03 08:00:00');
--  [2012-01-01 08:00:00+01, 2012-01-03 08:00:00+01)
-- Period defined with four arguments
SELECT period('2012-01-01 08:00:00', '2012-01-03 09:30:00', false, true);
-- (2012-01-01 08:00:00+01, 2012-01-03 09:30:00+01]
```

The `timestampset` type has a constructor function that accepts a single argument which is an array of `timestamptz` values.

- Constructor for `timestampset`

  `timestampset(timestamptz[]):  timestampset`

```
SELECT timestampset(ARRAY[timestamptz '2012-01-01 08:00:00', '2012-01-03 09:30:00']);
-- "{2012-01-01 08:00:00+00, 2012-01-03 09:30:00+00}"
```

The `periodset` type has a constructor function that accepts a single argument which is an array of `period` values.

- Constructor for `periodset`

  `periodset(period[]):  periodset`

```
SELECT periodset(ARRAY[period '[2012-01-01 08:00:00, 2012-01-01 08:10:00]',
-- '[2012-01-01 08:20:00, 2012-01-01 08:40:00]']);
```

2.1.2 Casting

Values of the `timestamptz`, `tstzrange`, or the time types can be converted to one another using the function `CAST` or using the `::` notation.

- Cast a `timestamptz` to another time type

  `timestamptz::timestampset`

  `timestamptz::period`

  `timestamptz::periodset`

```
SELECT CAST(timestamptz '2012-01-01 08:00:00' AS timestampset);
-- "{2012-01-01 08:00:00+01}"
SELECT CAST(timestamptz '2012-01-01 08:00:00' AS period);
-- "[2012-01-01 08:00:00+01, 2012-01-01 08:00:00+01]"
SELECT CAST(timestamptz '2012-01-01 08:00:00' AS periodset);
-- "{[2012-01-01 08:00:00+01, 2012-01-01 08:00:00+01]}"
```

- Cast a `timestampset` to a `periodset`

  `timestampset::periodset`

```
SELECT CAST(timestampset '{2012-01-01 08:00:00, 2012-01-01 08:15:00,
2012-01-01 08:25:00}' AS periodset);
-- "{[2012-01-01 08:00:00+01, 2012-01-01 08:00:00+01],
[2012-01-01 08:15:00+01, 2012-01-01 08:15:00+01],
[2012-01-01 08:25:00+01, 2012-01-01 08:25:00+01]}"
```

- Cast a `period` to another time type

  `period::periodset`

  `period::tstzrange`

```
SELECT period '[2012-01-01 08:00:00, 2012-01-01 08:30:00)'::periodset;
-- "{[2012-01-01 08:00:00+01, 2012-01-01 08:30:00+01)}"
SELECT period '[2012-01-01 08:00:00, 2012-01-01 08:30:00)'::tstzrange;
-- "["2012-01-01 08:00:00+01","2012-01-01 08:30:00+01")"
```

- Cast a `tstzrange` to a `period`

  `tstzrange::period`

```
SELECT tstzrange '[2012-01-01 08:00:00, 2012-01-01 08:30:00)'::period;
-- "[2012-01-01 08:00:00+01, 2012-01-01 08:30:00+01)"
```

2.1.3 Accessor Functions

- Get the memory size in bytes

  `memSize({timestampset, periodset}): integer`

```
SELECT memSize(timestampset '{2012-01-01, 2012-01-02, 2012-01-03}');
-- 104
SELECT memSize(periodset '{[2012-01-01, 2012-01-02], [2012-01-03, 2012-01-04],
[2012-01-05, 2012-01-06]}');
-- 136
```

- Get the lower bound

  `lower(period): timestamptz`

```
SELECT lower(period '[2011-01-01, 2011-01-05)');
-- "2011-01-01"
```

- Get the upper bound

  `upper(period): timestamptz`

```
SELECT upper(period '[2011-01-01, 2011-01-05)');
-- "2011-01-05"
```

- Is the lower bound inclusive?

  `lower_inc(period): boolean`

```
SELECT lower_inc(period '[2011-01-01, 2011-01-05)');
-- true
```

- Is the upper bound inclusive?

  `upper_inc(period): boolean`

```
SELECT upper_inc(period '[2011-01-01, 2011-01-05)');
-- false
```

- Get the duration

  `duration({period, periodset}):  interval`

```
SELECT duration(period '[2012-01-01, 2012-01-03)');
-- "2 days"
SELECT duration(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-05)}');
-- "3 days"
```

- Get the timespan ignoring the potential time gaps

  `timespan({timestampset, periodset}):  interval`

```
SELECT timespan(timestampset '{2012-01-01, 2012-01-03}');
-- "2 days"
SELECT timespan(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-05)}');
-- "4 days"
```

- Get the period on which the timestamp set or period set is defined ignoring the potential time gaps

  `period({timestampset, periodset}):  period`

```
SELECT period(timestampset '{2012-01-01, 2012-01-03, 2012-01-05}');
-- "[2012-01-01, 2012-01-05]"
SELECT period(periodset '{[2012-01-01, 2012-01-02), [2012-01-03, 2012-01-04)}');
-- "[2012-01-01, 2012-01-04)"
```

- Get the number of different timestamps

  `numTimestamps({timestampset, periodset}):  int`

```
SELECT numTimestamps(timestampset '{2012-01-01, 2012-01-03, 2012-01-04}');
-- 3
SELECT numTimestamps(periodset '{[2012-01-01, 2012-01-03), (2012-01-03, 2012-01-05)}');
-- 3
```

- Get the start timestamp

  `startTimestamp({timestampset, periodset}):  timestamptz`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT startTimestamp(periodset '{[2012-01-01, 2012-01-03), (2012-01-03, 2012-01-05)}');
-- "2012-01-01"
```

- Get the end timestamp

  `endTimestamp({timestampset, periodset}):  timestamptz`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT endTimestamp(periodset '{[2012-01-01, 2012-01-03), (2012-01-03, 2012-01-05)}');
-- "2012-01-05"
```

- Get the n-th different timestamp

  `timestampN({timestampset, periodset}, int): timestamptz`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT timestampN(periodset '{[2012-01-01, 2012-01-03), (2012-01-03, 2012-01-05)}', 3);
-- "2012-01-04"
```

- Get the different timestamps

  `timestamps({timestampset, periodset}): timestampset`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT timestamps(periodset '{[2012-01-01, 2012-01-03), (2012-01-03, 2012-01-05)}');
-- "{"2012-01-01", "2012-01-03", "2012-01-05"}"
```

- Get the number of periods

  `numPeriods(periodset): int`

```
SELECT numPeriods(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-04],
[2012-01-05, 2012-01-06)}');
-- 3
```

- Get the start period

  `startPeriod(periodset): period`

```
SELECT startPeriod(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-04],
[2012-01-05, 2012-01-06)}');
-- "[2012-01-01,2012-01-03)"
```

- Get the end period

  `endPeriod(periodset): period`

```
SELECT endPeriod(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-04],
[2012-01-05, 2012-01-06)}');
-- "[2012-01-05,2012-01-06)"
```

- Get the n-th period

  `periodN(periodset, int): period`

```
SELECT periodN(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-04],
[2012-01-05, 2012-01-06)}', 2);
-- "[2012-01-04,2012-01-04]"
```

- Get the periods

  `periods(periodset): period[]`

```
SELECT periods(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-04],
[2012-01-05, 2012-01-06)}');
-- "{"[2012-01-01,2012-01-03)", "[2012-01-04,2012-01-04]", "[2012-01-05,2012-01-06)"}"
```

- Shift the time value by an interval

  `shift({timestampset,period,periodset}): {timestampset,period,periodset}`

```
SELECT shift(timestampset '{2001-01-01, 2001-01-03, 2001-01-05}', '1 day'::interval);
-- "{2001-01-02, 2001-01-04, 2001-01-06}"
SELECT shift(period '[2001-01-01, 2001-01-03]', '1 day'::interval);
-- "[2001-01-02, 2001-01-04]"
SELECT shift(periodset '{[2001-01-01, 2001-01-03], [2001-01-04, 2001-01-05]}',
'1 day'::interval);
-- "{[2001-01-02, 2001-01-04], [2001-01-05, 2001-01-06]}"
```

2.1.4 Comparison Operators

The comparison operators (=, <, and so on) require that the left and right arguments be of the same type. Excepted equality and inequality, the other comparison operators are not useful in the real world but allow B-tree indexes to be constructed on time types. For period values, the operators compare first the lower bound, then the upper bound. For timestamp set and period set values, the operators compare first the bounding periods, and if those are equal, they compare the first N instants or periods, where N is the minimum of the number of composing instants or periods of both values.

The comparison operators available for the time types are given next.

- Are the time values equal?

  `time = time`

```
SELECT period '[2012-01-01, 2012-01-04)' = period '[2012-01-01, 2012-01-04)';
-- true
```

- Are the time values different?

  `time <> time`

```
SELECT period '[2012-01-01, 2012-01-04)' <> period '[2012-01-03, 2012-01-05)';
-- true
```

- Is the first time value less than the second one?

  `time < time`

```
SELECT timestampset '{2012-01-01, 2012-01-04}' < timestampset '{2012-01-01, 2012-01-05}';
-- true
```

- Is the first time value greater than the second one?

  `time > time`

```
SELECT period '[2012-01-03, 2012-01-04)' > period '[2012-01-02, 2012-01-05)';
-- true
```

- Is the first time value less than or equal to the second one?

  `time <= time`

```
SELECT periodset '{[2012-01-01, 2012-01-04)}' <=
periodset '{[2012-01-01, 2012-01-05), [2012-01-06, 2012-01-07)}';
-- true
```

- Is the first time value greater than or equal to the second one?

  `time >= time`

```
SELECT period '[2012-01-03, 2012-01-05)' >= period '[2012-01-03, 2012-01-04)';
-- true
```

2.1.5 Set Operators

The set operators available for the time types are given next.

- Union of the time values

  `time + time`

```
SELECT timestampset '{2011-01-01, 2011-01-03, 2011-01-05}' +
timestampset '{2011-01-03, 2011-01-06}';
-- "{2011-01-01, 2011-01-03, 2011-01-05, 2011-01-06}"
SELECT period '[2011-01-01, 2011-01-05)' + period '[2011-01-03, 2011-01-07)';
-- "[2011-01-01, 2011-01-07)"
SELECT periodset '{[2011-01-01, 2011-01-03), [2011-01-04, 2011-01-05)}' +
period '[2011-01-03, 2011-01-04)';
-- "{[2011-01-01, 2011-01-05)}"
```

- Intersection of the time values

  `time * time`

```
SELECT timestampset '{2011-01-01, 2011-01-03}' * timestampset '{2011-01-03, 2011-01-05}';
-- "{2011-01-03}"
SELECT period '[2011-01-01, 2011-01-05)' * period '[2011-01-03, 2011-01-07)';
-- "[2011-01-03, 2011-01-05)"
```

- Difference of the time values

  `time - time`

```
SELECT period '[2011-01-01, 2011-01-05)' - period '[2011-01-03, 2011-01-07)';
-- "[2011-01-01, 2011-01-03)"
SELECT period '[2011-01-01, 2011-01-05]' - period '[2011-01-03, 2011-01-04]'
-- "{[2011-01-01,2011-01-03), (2011-01-04,2011-01-05]}"
SELECT periodset '{[2011-01-01, 2011-01-06], [2011-01-07, 2011-01-10]}' -
periodset '{[2011-01-02, 2011-01-03], [2011-01-04, 2011-01-05],
[2011-01-08, 2011-01-09]}';
-- "{[2011-01-01,2011-01-02), (2011-01-03,2011-01-04), (2011-01-05,2011-01-06],
[2011-01-07,2011-01-08), (2011-01-09,2011-01-10]}"
```

2.1.6 Topological Operators

The topological operators available for the time types are given next.

- Do the time values overlap (have instants in common)?

  `{timestampset, period, periodset} && {timestampset, period, periodset}`

```
SELECT period '[2011-01-01, 2011-01-05)' && period '[2011-01-02, 2011-01-07)';
-- true
```

- Does the first time value contain the second one?

  `{timestampset, period, periodset} @> time`

```
SELECT period '[2011-01-01, 2011-05-01)' @> period '[2011-02-01, 2011-03-01)';
-- true
SELECT period '[2011-01-01, 2011-05-01)' @> timestamptz '2011-02-01';
-- true
```

- Is the first time value contained by the second one?

  `time <@ {timestampset, period, periodset}`

```
SELECT period '[2011-02-01, 2011-03-01)' <@ period '[2011-01-01, 2011-05-01)';
-- true
SELECT timestamptz '2011-01-10' <@ period '[2011-01-01, 2011-05-01)';
-- true
```

- Is the first time value adjacent to the second one?

  `time -|- time`

```
SELECT period '[2011-01-01, 2011-01-05)' -|- timestampset '{2011-01-05, 2011-01-07}';
-- true
SELECT periodset '{[2012-01-01, 2012-01-02]}' -|- period '[2012-01-02, 2012-01-03)';
-- false
```

2.1.7 Relative Position Operators

In PostgreSQL, the range operators `<<`, `&<`, `>>`, `&>`, and `-|-` only accept ranges as left or right argument. We extended these operators for numeric ranges so that one argument may be an integer or a float.

The relative position operators available for the time types and range types are given next.

- Is the first number or range value strictly left of the second one?

  `{number, range} << {number, range}`

```
SELECT intrange '[15, 20)' << 20;
-- true
```

- Is the first number or range value strictly right of the second one?

  `{number, range} >> {number, range}`

```
SELECT intrange '[15, 20)' >> 10;
-- true
```

- Is the first number or range value not to the right of the second one?

  `{number, range} &< {number, range}`

```
SELECT intrange '[15, 20)' &< 18;
-- false
```

- Is the first number or range value not to the left of the second one?

  `{number, range} &> {number, range}`

```
SELECT period '[2011-01-01, 2011-01-03)' &> period '[2011-01-01, 2011-01-05)';
-- true
SELECT intrange '[15, 20)' &> 30;
-- true
```

- Is the first number or range value adjacent to the second one?

  `{number, range} -|- {number, range}`

```
SELECT floatrange '[15, 20)' -|- 20;
-- true
```

- Is the first time value scritly before the second one?

  `time <<# time`

```
SELECT period '[2011-01-01, 2011-01-03)' <<# timestampset '{2011-01-03, 2011-01-05}';
-- true
```

- Is the first time value strictly after the second one?

  `time #>> time`

```
SELECT period '[2011-01-04, 2011-01-05)' #>>
periodset '{[2011-01-01, 2011-01-04), [2011-01-05, 2011-01-06)}';
-- true
```

- Is the first time value not after the second one?

  `time &<# time`

```
SELECT timestampset '{2011-01-02, 2011-01-05}' &<# period '[2011-01-01, 2011-01-05)';
-- false
```

- Is the first time value not before the second one?

  `time #&> time`

```
SELECT timestamp '2011-01-01' #&> period '[2011-01-01, 2011-01-05)';
-- true
```

2.1.8 Aggregate Functions

The temporal aggregate functions generalize the traditional aggregate functions. Their semantics is that they compute the value of the function at every instant in the *union* of the temporal extents of the values to aggregate. In contrast, recall that all other functions manipulating time types compute the value of the function at every instant in the *intersection* of the temporal extents of the arguments.

The temporal aggregate functions are the following ones:

- Function `tcount` generalizes the traditional function `count`. The temporal count can be used to compute at each point in time the number of available objects (for example, number of periods). Function `tcount` returns a temporal integer (see Chapter 3).
- Function `extent` returns a bounding period that encloses a set of time values.

Union is a very useful operation for time types. As we have seen in Section 2.1.5, we can compute the union of two time values using the + operator. However, it is also very useful to have an aggregate version of the union operator for combining an arbitrary number of values. Function `tunion` can be used for this purpose.

- Temporal count

  `tcount({timestampset, period, periodset}):  {tinti, tints}`

```
WITH times(ts) AS (
  SELECT timestampset '{2000-01-01, 2000-01-03, 2000-01-05}' UNION
  SELECT timestampset '{2000-01-02, 2000-01-04, 2000-01-06}' UNION
  SELECT timestampset '{2000-01-01, 2000-01-02}'
)
SELECT tcount(ts) FROM times;
-- "{2@2000-01-01, 2@2000-01-02, 1@2000-01-03, 1@2000-01-04, 1@2000-01-05, 1@2000-01-06}"

WITH periods(ps) AS (
  SELECT periodset '{[2000-01-01, 2000-01-02], [2000-01-03, 2000-01-04]}' UNION
  SELECT periodset '{[2000-01-01, 2000-01-04], [2000-01-05, 2000-01-06]}' UNION
  SELECT periodset '{[2000-01-02, 2000-01-06]}'
)
SELECT tcount(ps) FROM periods;
-- {[2@2000-01-01, 3@2000-01-02], (2@2000-01-02, 3@2000-01-03, 3@2000-01-04], (1@2000 ↩
    -01-04, 2@2000-01-05, 2@2000-01-06]}
```

- Bounding period

  `extent({timestampset, period, periodset}):  period`

```
WITH times(ts) AS (
  SELECT timestampset '{2000-01-01, 2000-01-03, 2000-01-05}' UNION
  SELECT timestampset '{2000-01-02, 2000-01-04, 2000-01-06}' UNION
  SELECT timestampset '{2000-01-01, 2000-01-02}'
)
SELECT extent(ts) FROM times;
-- "[2000-01-01, 2000-01-06]"

WITH periods(ps) AS (
  SELECT periodset '{[2000-01-01, 2000-01-02], [2000-01-03, 2000-01-04]}' UNION
  SELECT periodset '{[2000-01-01, 2000-01-04], [2000-01-05, 2000-01-06]}' UNION
  SELECT periodset '{[2000-01-02, 2000-01-06]}'
)
SELECT extent(ps) FROM periods;
-- "[2000-01-01, 2000-01-06]"
```

- Temporal union

  `tunion({timestampset, period, periodset}):  {timestampset, periodset}`

```
WITH times(ts) AS (
  SELECT timestampset '{2000-01-01, 2000-01-03, 2000-01-05}' UNION
  SELECT timestampset '{2000-01-02, 2000-01-04, 2000-01-06}' UNION
  SELECT timestampset '{2000-01-01, 2000-01-02}'
)
SELECT tunion(ts) FROM times;
-- "{2000-01-01, 2000-01-02, 2000-01-03, 2000-01-04, 2000-01-05, 2000-01-06}"
WITH periods(ps) AS (
  SELECT periodset '{[2000-01-01, 2000-01-02], [2000-01-03, 2000-01-04]}' UNION
  SELECT periodset '{[2000-01-02, 2000-01-03], [2000-01-05, 2000-01-06]}' UNION
  SELECT periodset '{[2000-01-07, 2000-01-08]}'
)
SELECT tunion(ps) FROM periods;
-- "{[2000-01-01, 2000-01-04], [2000-01-05, 2000-01-06], [2000-01-07, 2000-01-08]}"
```

2.2 Indexing of Time Types

GiST and SP-GiST indexes can be created for table columns of the `timestampset`, `period`, and `periodset` types. An example of creation of a GiST index in a column `During` of type `period` in a table `Reservation` is as follows:

```
CREATE TABLE Reservation (ReservationID integer PRIMARY KEY, RoomID integer,
  During period);
CREATE INDEX Reservation_During_Idx ON Reservation USING GIST(During);
```

A GiST or SP-GiST index can accelerate queries involving the following operators: `=`, `&&`, `<@`, `@>`, `-|-`, `<<`, `>>`, `&<`, and `&>`.

In addition, B-tree indexes can be created for table columns of a time type. For these index types, basically the only useful operation is equality. There is a B-tree sort ordering defined for values of time types with corresponding < and > operators, but the ordering is rather arbitrary and not usually useful in the real world. The B-tree support is primarily meant to allow sorting internally in queries, rather than creation of actual indexes.

Chapter 3

Temporal Types

There are six built-in temporal types, namely `tbool`, `tint`, `tfloat`, `ttext`, `tgeompoint`, and `tgeogpoint`, which are, respectively, based on the base types `bool`, `int`, `float`, `text`, `geometry`, and `geography` (the last two types restricted to 2D or 3D points with Z dimension).

The *interpolation* of a temporal value states how the value evolves between successive instants. The interpolation is *stepwise* when the value remains constant between two successive instants. For example, the number of employees of a department may be represented with a temporal integer, which indicates that its value is constant between two time instants. On the other hand, the interpolation is *linear* when the value evolves linearly between two successive instants. For example, the temperature of a room may be represented with a temporal float, which indicates that the values are known at the two time instants but continuously evolve between them. Similarly, the location of a vehicule may be represented by a temporal point where the location between two consecutive GPS readings is obtained by linear interpolation. Temporal types based on discrete base types, that is the `tbool`, `tint`, or `ttext` evolve necesssarily in a stepwise manner. On the other hand, temporal types based on continuous base types, that is `tfloat`, `tgeompoint`, or `tgeogpoint` may evolve in a stepwise or linear manner.

The *subtype* of a temporal value states the temporal extent at which the evolution of values is recorded. Temporal values come in four subtypes, namely, instant, instant set, sequence, and sequence set.

A temporal value of *instant* subtype (briefly, *an instant value*) represents the value at a time instant, for example

```
SELECT tfloat '17@2018-01-01 08:00:00';
```

A temporal value of *instant set* subtype (briefly, an *instant set value*) represents the evolution of the value at a set of time instants, where the values between these instants are unknown. An example is as follows:

```
SELECT tfloat '{17@2018-01-01 08:00:00, 17.5@2018-01-01 08:05:00, 18@2018-01-01 08:10:00}';
```

A temporal value of *sequence* subtype (briefly, a *sequence value*) represents the evolution of the value during a sequence of time instants, where the values between these instants are interpolated using either a stepwise or a linear function (see below). An example is as follows:

```
SELECT tint '(10@2018-01-01 08:00:00, 20@2018-01-01 08:05:00, 15@2018-01-01 08:10:00]';
```

As can be seen, a sequence value has a lower and an upper bound that can be inclusive (represented by '`[`' and '`]`') or exclusive (represented by '`(`' and '`)`'). A sequence value with a single instant such as

```
SELECT tint '[10@2018-01-01 08:00:00]';
```

is called an *instantaneous sequence*. In that case, both bounds must be inclusive.

The value of a temporal sequence is interpreted by assuming that the period of time defined by every pair of consecutive values `v1@t1` and `v2@t2` is lower inclusive and upper exclusive, unless they are the first or the last instants of the sequence and in that case the bounds of the whole sequence apply. Furthermore, the value taken by the temporal sequence between two consecutive instants depends on whether the interpolation is stepwise or linear. For example, the temporal sequence above represents that the value is `10` during `(2018-01-01 08:00:00, 2018-01-01 08:05:00)`, `20` during `[2018-01-01 08:05:00, 2018-01-01 08:10:00)`, and `15` at the end instant `2018-01-01 08:10:00`. On the other hand, the following temporal sequence

```
SELECT tfloat '(10@2018-01-01 08:00:00, 20@2018-01-01 08:05:00, 15@2018-01-01 08:10:00]';
```

represents that the value evolves linearly from `10` to `20` during `(2018-01-01 08:00:00, 2018-01-01 08:05:00)` and evolves from `20` to `15` during `[2018-01-01 08:05:00, 2018-01-01 08:10:00]`.

Finally, a temporal value of *sequence set* subtype (briefly, a *sequence set value*) represents the evolution of the value at a set of sequences, where the values between these sequences are unknown. An example is as follows:

```
SELECT tfloat '{[17@2018-01-01 08:00:00, 17.5@2018-01-01 08:05:00],
  [18@2018-01-01 08:10:00, 18@2018-01-01 08:15:00]}';
```

Temporal values with instant or sequence subtype are called *temporal unit values*, while temporal values with instant set or sequence set subtype are called *temporal set values*. Temporal set values can be thought of as an array of the corresponding unit values. Temporal set values must be *uniform*, that is, they must be constructed from unit values of the same base type and the same subtype.

Temporal sequence values are converted into *normal form* so that equivalent values have identical representations. For this, consecutive instant values are merged when possible. For stepwise interpolation, three consecutive instant values can be merged into two if they have the same value. For linear interpolation, three consecutive instant values can be merged into two if the linear functions defining the evolution of values are the same. Examples of transformation into normal form are as follows.

```
SELECT tint '[1@2001-01-01, 2@2001-01-03, 2@2001-01-04, 2@2001-01-05)';
-- "[1@2001-01-01 00:00:00+00, 2@2001-01-03 00:00:00+00, 2@2001-01-05 00:00:00+00)"
SELECT tgeompoint '[Point(1 1)@2001-01-01 08:00:00, Point(1 1)@2001-01-01 08:05:00,
Point(1 1)@2001-01-01 08:10:00)';
-- "[Point(1 1)@2001-01-01 08:00:00, Point(1 1)@2001-01-01 08:10:00)"
SELECT tfloats(ARRAY[tfloat '[1@2001-01-01, 2@2001-01-03, 3@2001-01-05]']);
-- "{[1@2001-01-01 00:00:00+00, 3@2001-01-05 00:00:00+00]}"
SELECT tgeompoint '[Point(1 1)@2001-01-01 08:00:00, Point(2 2)@2001-01-01 08:05:00,
Point(3 3)@2001-01-01 08:10:00]';
-- "[Point(1 1)@2001-01-01 08:00:00, Point(3 3)@2001-01-01 08:10:00]"
```

Similary, temporal sequence set values are converted into normal form. For this, consecutive sequence values are merged when possible. Examples of transformation into a normal form are as follows.

```
SELECT tints(ARRAY[tint '[1@2001-01-01, 1@2001-01-03)', '[2@2001-01-03, 2@2001-01-05)']);
-- '{[1@2001-01-01 00:00:00+00, 2@2001-01-03 00:00:00+00, 2@2001-01-05 00:00:00+00)}'
SELECT tfloats(ARRAY[tfloat '[1@2001-01-01, 2@2001-01-03)',
'[2@2001-01-03, 3@2001-01-05]']);
-- '{[1@2001-01-01 00:00:00+00, 3@2001-01-05 00:00:00+00]}'
SELECT tfloats(ARRAY[tfloat '[1@2001-01-01, 3@2001-01-05)', '[3@2001-01-05]']);
-- '{[1@2001-01-01 00:00:00+00, 3@2001-01-05 00:00:00+00]}'
SELECT tgeompoint '{[Point(0 0)@2001-01-01 08:00:00,
Point(1 1)@2001-01-01 08:05:00, Point(1 1)@2001-01-01 08:10:00),
[Point(1 1)@2001-01-01 08:10:00, Point(1 1)@2001-01-01 08:15:00)}';
-- "{[[Point(0 0)@2001-01-01 08:00:00, Point(1 1)@2001-01-01 08:05:00,
```

```
Point(1 1)@2001-01-01 08:15:00)}"
SELECT tgeompoint '{[Point(1 1)@2001-01-01 08:00:00, Point(2 2)@2001-01-01 08:05:00),
[Point(2 2)@2001-01-01 08:05:00, Point(3 3)@2001-01-01 08:10:00]}';
-- "{[Point(1 1)@2001-01-01 08:00:00, Point(3 3)@2001-01-01 08:10:00]}"
SELECT tgeompoint '{[Point(1 1)@2001-01-01 08:00:00, Point(3 3)@2001-01-01 08:10:00),
[Point(3 3)@2001-01-01 08:10:00]}';
-- "{[Point(1 1)@2001-01-01 08:00:00, Point(3 3)@2001-01-01 08:10:00]}"
```

Temporal types support *type modifiers* (or `typmod` in PostgreSQL terminology), which specify additional information for a column definition. For example, in the following table definition:

```
CREATE TABLE Department(DeptNo integer, DeptName varchar(25), NoEmps tint(Sequence));
```

the type modifier for the type `varchar` is the value 25, which indicates the maximum length of the values of the column, while the type modifier for the type `tint` is the string `Sequence`, which restricts the subtype of the values of the column to be sequences. In the case of temporal alphanumeric types (that is, `tbool`, `tint`, `tfloat`, and `ttext`), the possible values for the type modifier are `Instant`, `InstantSet`, `Sequence`, and `SequenceSet`. If no type modifier is specified for a column, values of any subtype are allowed.

On the other hand, in the case of temporal point types (that is, `tgeompoint` or `tgeogpoint`) the type modifier may be used to specify specify the subtype, the dimensionality, and/or the spatial reference identifier (SRID). For example, in the following table definition:

```
CREATE TABLE Flight(FlightNo integer, Route tgeogpoint(Sequence, PointZ, 4326));
```

the type modifier for the type `tgeogpoint` is composed of three values, the first one indicating the subtype as above, the second one the spatial type of the geographies composing the temporal point, and the last one the SRID of the composing geographies. For temporal points, the possible values for the first argument of the type modifier are as above, those for the second argument are either `Point` or `PointZ`, and those for the third argument are valid SRIDs. All the three arguments are optional and if any of them is not specified for a column, values of any subtype, dimensionality, and/or SRID are allowed.

Each temporal type is associated to another type, referred to as its *bounding box*, which represent its extent in the value and/or the time dimension. The bounding box of the various temporal types are as follows:

- The `period` type for the `tbool` and `ttext` types, where only the temporal extent is considered.
- A `tbox` (temporal box) type for the `tint` and `tfloat` types, where the value extent is defined in the X dimension and the temporal extent in the T dimension.
- A `stbox` (spatiotemporal box) type for the `tgeompoint` and `tgeogpoint` types, where the spatial extent is defined in the X, Y, and Z dimensions, and the temporal extent in the T dimension.

A rich set of functions and operators is available to perform various operations on temporal types. They are explained in Chapter 5. Some of these operations, in particular those related to indexes, manipulate bounding boxes for efficiency reasons.

3.1 Examples of Temporal Types

Examples of usage of temporal alphanumeric types are given next.

```
CREATE TABLE Department(DeptNo integer, DeptName varchar(25), NoEmps tint);
INSERT INTO Department VALUES
(10, 'Research', tint '[10@2012-01-01, 12@2012-04-01, 12@2012-08-01)'),
(20, 'Human Resources', tint '[4@2012-02-01, 6@2012-06-01, 6@2012-10-01)');
CREATE TABLE Temperature(RoomNo integer, Temp tfloat);
```

```
INSERT INTO Temperature VALUES
(1001, tfloat '{18.5@2012-01-01 08:00:00, 20.0@2012-01-01 08:10:00}'),
(2001, tfloat '{19.0@2012-01-01 08:00:00, 22.5@2012-01-01 08:10:00}');
-- Value at a timestamp
SELECT RoomNo, valueAtTimestamp(Temp, '2012-01-01 08:10:00')
FROM temperature;
-- 1001;
-- 2001;22.5
-- Restriction to a value
SELECT DeptNo, atValue(NoEmps, 10)
FROM Department;
-- 10;"[10@2012-01-01 00:00:00+00, 10@2012-04-01 00:00:00+00)"
-- 20; NULL
-- Restriction to a period
SELECT DeptNo, atPeriod(NoEmps, '[2012-01-01, 2012-04-01]')
FROM Department;
-- 10;"[10@2012-01-01 00:00:00+00, 12@2012-04-01 00:00:00+00]"
-- 20;"[4@2012-02-01 00:00:00+00, 4@2012-04-01 00:00:00+00]"
-- Temporal comparison
SELECT DeptNo, NoEmps #<= 10
FROM Department;
-- 10;"[t@2012-01-01 00:00:00+00, f@2012-04-01 00:00:00+00, f@2012-08-01 00:00:00+00)"
-- 20;"[t@2012-04-02 00:00:00+00, t@2012-10-01 00:00:00+00)"
-- Temporal aggregation
SELECT tsum(NoEmps)
FROM Department;
-- "{[10@2012-01-01 00:00:00+00, 14@2012-02-01 00:00:00+00, 16@2012-04-01 00:00:00+00,
18@2012-06-01 00:00:00+00, 6@2012-08-01 00:00:00+00, 6@2012-10-01 00:00:00+00)}"
```

Examples of usage of temporal point types are given next.

```
CREATE TABLE Trips(CarId integer, TripId integer, Trip tgeompoint);
INSERT INTO Trips VALUES
(10, 1, tgeompoint '{[Point(0 0)@2012-01-01 08:00:00, Point(2 0)@2012-01-01 08:10:00,
Point(2 1)@2012-01-01 08:15:00)}'),
(20, 1, tgeompoint '{[Point(0 0)@2012-01-01 08:05:00, Point(1 1)@2012-01-01 08:10:00,
Point(3 3)@2012-01-01 08:20:00)}');
-- Value at a given timestamp
SELECT CarId, ST_AsText(valueAtTimestamp(Trip, timestamptz '2012-01-01 08:10:00'))
FROM Trips;
-- 10;"POINT(2 0)"
-- 20;"POINT(1 1)"
-- Restriction to a given value
SELECT CarId, asText(atValue(Trip, 'Point(2 0)'))
FROM Trips;
-- 10;"{"[POINT(2 0)@2012-01-01 08:10:00+00]"}"
-- 20; NULL
-- Restriction to a period
SELECT CarId, asText(atPeriod(Trip, '[2012-01-01 08:05:00,2012-01-01 08:10:00]'))
FROM Trips;
-- 10;"{[POINT(1 0)@2012-01-01 08:05:00+00, POINT(2 0)@2012-01-01 08:10:00+00]}"
-- 20;"{[POINT(0 0)@2012-01-01 08:05:00+00, POINT(1 1)@2012-01-01 08:10:00+00]}"
-- Temporal distance
SELECT T1.CarId, T2.CarId, T1.Trip <-> T2.Trip
FROM Trips T1, Trips T2
WHERE T1.CarId < T2.CarId;
-- 10;20;"{[1@2012-01-01 08:05:00+00, 1.4142135623731@2012-01-01 08:10:00+00,
1@2012-01-01 08:15:00+00)}"
```

3.2 Validity of Temporal Types

Values of temporal types must satisfy several constraints so that they are well defined. These constraints are given next.

- The constraints on the base type and the `timestamptz` type must be satisfied.
- A sequence value must be composed of at least one instant value.
- An instantaneous sequence value must have inclusive lower and upper bounds.
- In a sequence value, the timestamps of the composing instants must be different and ordered.
- In a sequence value with stepwise interpolation, the last two values must be equal if upper bound is exclusive.
- A set value must be composed of at least one unit value.
- In an instant set value, the composing instants must be different and ordered. This implies that the temporal extent of an instant set value is an ordered set of `timestamptz` values without duplicates.
- In a sequence set value, the composing sequence values must be non overlapping and ordered. This implies that the temporal extent of a sequence set value is an ordered set of disjoint periods.

An error is raised whenever one of these constraints are not satisfied. Examples of incorrect temporal values are as follows.

```
-- Incorrect value for base type
SELECT tbool '1.5@2001-01-01 08:00:00';
-- Base type value is not a point
SELECT tgeompoint 'Linestring(0 0,1 1)@2001-01-01 08:05:00';
-- Incorrect timestamp
SELECT tint '2@2001-02-31 08:00:00';
-- Empty sequence
SELECT tint '';
-- Incorrect bounds for instantaneous sequence
SELECT tint '[1@2001-01-01 09:00:00)';
-- Duplicate timestamps
SELECT tint '[1@2001-01-01 08:00:00, 2@2001-01-01 08:00:00]';
-- Unordered timestamps
SELECT tint '[1@2001-01-01 08:10:00, 2@2001-01-01 08:00:00]';
-- Incorrect end value
SELECT tint '[1@2001-01-01 08:00:00, 2@2001-01-01 08:10:00)';
-- Empty sequence set
SELECT tints(ARRAY[]);
-- Duplicate timestamps
SELECT tinti(ARRAY[tint '1@2001-01-01 08:00:00', '2@2001-01-01 08:00:00']);
-- Overlapping periods
SELECT tints(ARRAY[tint '[1@2001-01-01 08:00:00, 1@2001-01-01 10:00:00)',
  '[2@2001-01-01 09:00:00, 2@2001-01-01 11:00:00)']);
```

Chapter 4

Manipulating Bounding Box Types

4.1 Input/Output of Bounding Box Types

A `tbox` is composed of a numeric value and/or time dimensions. For each dimension, minimum and maximum values are given. Examples of input of `tbox` values are as follows:

```
-- Both value and time dimensions
SELECT tbox 'TBOX((1.0, 2000-01-01), (2.0, 2000-01-02))';
-- Only value dimension
SELECT tbox 'TBOX((1.0,), (2.0,))';
-- Only time dimension
SELECT tbox 'TBOX((, 2000-01-01), (, 2000-01-02))';
```

An `stbox` is composed of a spatial value and/or time dimensions, where the coordinates of the spatial value dimension may be 2D or 3D. For each dimension, minimum and maximum values are given. The coordinates may be Cartesian (planar) or geodetic (spherical). The SRID of the coordinates may be specified; if it is not the case, a value of 0 (unknown) and 4326 (corresponding to WGS84) is assumed, respectively, for planar and geodetic boxes. Examples of input of `stbox` values are as follows:

```
-- Only value dimension with X and Y coordinates
SELECT stbox 'STBOX((1.0, 2.0), (1.0, 2.0))';
-- Only value dimension with X, Y, and Z coordinates
SELECT stbox 'STBOX Z((1.0, 2.0, 3.0), (1.0, 2.0, 3.0))';
-- Both value (with X and Y coordinates) and time dimensions
SELECT stbox 'STBOX T((1.0, 2.0, 2001-01-03), (1.0, 2.0, 2001-01-03))';
-- Both value (with X, Y, and Z coordinates) and time dimensions
SELECT stbox 'STBOX ZT((1.0, 2.0, 3.0, 2001-01-04), (1.0, 2.0, 3.0, 2001-01-04))';
-- Only time dimension
SELECT stbox 'STBOX T(( , , 2001-01-03), ( , , 2001-01-03))';
-- Only value dimension with X, Y, and Z geodetic coordinates
SELECT stbox 'GEODSTBOX((1.0, 2.0, 3.0), (1.0, 2.0, 3.0))';
-- Both value (with X, Y and Z geodetic coordinates) and time dimension
SELECT stbox 'GEODSTBOX T((1.0, 2.0, 3.0, 2001-01-04), (1.0, 2.0, 3.0, 2001-01-04))';
-- Only time dimension for geodetic box
SELECT stbox 'GEODSTBOX T(( , , 2001-01-03), ( , , 2001-01-03))';
-- SRID is given
SELECT stbox 'SRID=5676;STBOX T((1.0, 2.0, 2001-01-04), (1.0, 2.0, 2001-01-04))';
SELECT stbox 'SRID=4326;GEODSTBOX((1.0, 2.0, 3.0), (1.0, 2.0, 3.0))';
```

4.2 Constructor Functions

Type `tbox` has several constructor functions depending on whether the value and/or the time dimensions are given. These functions have two arguments for the minimum and maximum `float` values and/or two arguments for the minimum and maximum `timestamptz` values.

- Constructor for `tbox`

```
tbox(float, float):  tbox
tboxt(timestamptz, timestamptz):  tbox
tbox(float, timestamptz, float, timestamptz):  tbox
```

```
-- Both value and time dimensions
SELECT tbox(1.0, '2001-01-01', 2.0, '2001-01-02');
-- Only value dimension
SELECT tbox(1.0, 2.0);
-- Only time dimension
SELECT tboxt('2001-01-01', '2001-01-02');
```

Type `stbox` has several constructor functions depending on whether the coordinates and/or the time dimensions are given. Furthermore, the coordinates can be 2D or 3D and can be either Cartesian or geodetic. These functions have several arguments for the minimum and maximum coordinate values and/or two arguments for the minimum and maximum `timestamptz` values. The SRID can be specified in an optional last argument. If not given, a value 0 (respectively 4326) is assumed by default for planar (respectively geodetic) boxes.

- Constructor for `stbox`

```
stbox(float, float, float, float, integer):  stbox
stbox(float, float, float, float, float, float, integer):  stbox
stbox(float, float, float, timestamptz, float, float, float, timestamptz, integer):  stbo
stboxt(timestamptz, timestamptz, integer):  stbox
stbox(float, float, timestamptz, float, float, timestamptz, integer):  stbox
stbox(geo, {timestamp, period}):  stbox
```

```
-- Only value dimension with X and Y coordinates
SELECT stbox(1.0, 2.0, 1.0, 2.0);
-- Only value dimension with X, Y, and Z coordinates
SELECT stbox(1.0, 2.0, 3.0, 1.0, 2.0, 3.0);
-- Only value dimension with X, Y, and Z coordinates and SRID
SELECT stbox(1.0, 2.0, 3.0, 1.0, 2.0, 3.0);
-- Both value (with X and Y coordinates) and time dimensions
SELECT stboxt(1.0, 2.0, '2001-01-03', 1.0, 2.0, '2001-01-03');
-- Both value (with X, Y, and Z coordinates) and time dimensions
SELECT stbox(1.0, 2.0, 3.0, '2001-01-04', 1.0, 2.0, 3.0, '2001-01-04');
-- Only time dimension
SELECT stboxt('2001-01-03', '2001-01-03');
-- Only value dimension with X, Y, and Z geodetic coordinates
SELECT geodstbox(1.0, 2.0, 3.0, 1.0, 2.0, 3.0);
--  Both value (with X, Y, and Z geodetic coordinates) and time dimensions
SELECT geodstbox(1.0, 2.0, 3.0, '2001-01-04', 1.0, 2.0, 3.0, '2001-01-03');
-- Only time dimension for geodetic box
SELECT geodstboxt('2001-01-03', '2001-01-03');
SELECT stbox(geometry 'Linestring(1 1 1,2 2 2)', period '[2012-01-03, 2012-01-05]');
-- "STBOX ZT((1,1,1,2012-01-03),(2,2,2,2012-01-05))"
```

```
SELECT stbox(geography 'Linestring(1 1 1,2 2 2)', period '[2012-01-03, 2012-01-05]');
-- "GEODSTBOX T((0.99878198,0.017449748,0.017452406,2012-01-03),
(0.99969542,0.034878239,0.034899499,2012-01-05))"
```

4.3 Casting

- Cast a `tbox` to another type

  `tbox::{floatrange,period}`

```
SELECT tbox 'TBOX((1,2000-01-01),(2,2000-01-02))'::floatrange;
-- "[1,2]"
SELECT tbox 'TBOX((1,2000-01-01),(2,2000-01-02))'::period;
-- "[2000-01-01, 2000-01-02]"
```

- Cast another type to a `tbox`

  `{int, float, numeric, intrange, floatrange}::tbox,`

  `{timestamptz, timestampset, period, periodset, tint, tfloat}::tbox`

```
SELECT floatrange '(1.0, 2.0)'::tbox;
-- "TBOX((1,),(2,))"
SELECT periodset '{(2001-01-01,2001-01-02), (2001-01-03,2001-01-04)}'::tbox;
-- "TBOX((,2001-01-01),(,2001-01-04))"
```

- Cast an `stbox` to a another type

  `stbox::{period, box2d, box3d}`

```
SELECT stbox 'STBOX T((1.0, 2.0, 2001-01-01), (3.0, 4.0, 2001-01-03))'::period;
-- "[2000-01-01, 2000-01-03]"
SELECT stbox 'STBOX Z((1 1 1),(3 3 3))'::box2d;
-- "BOX(1 1,3 3)"
SELECT stbox 'STBOX Z((1 1 1),(3 3 3))'::box3d;
-- "BOX3D(1 1 1,3 3 3)"
```

- Cast another type to an `stbox`

  `{geometry, geography, box2d, box3d}::stbox`

  `{timestamptz, timestampset, period, periodset, tgeompoint, tgeogpoint}::stbox`

```
select geometry 'Linestring(1 1,2 2)'::stbox;
-- "STBOX((1,1),(2,2))"
SELECT periodset '{(2001-01-01,2001-01-02), (2001-01-03,2001-01-04)}'::stbox;
-- "STBOX T((,,2001-01-01),(,,2001-01-04))"
```

4.4 Accessor Functions

- Has X dimension?

  `hasX({tbox, stbox}):  bool`

```
SELECT hasX(tbox 'TBOX((, 2000-01-01), (, 2000-01-03))');
-- false
SELECT hasX(stbox 'STBOX((1.0, 2.0), (3.0, 4.0))');
-- true
```

- Has Z dimension?

  `hasZ(stbox):  bool`

```
SELECT hasZ(stbox 'STBOX((1.0, 2.0), (3.0, 4.0))');
-- false
```

- Has T dimension?

  `hasT({tbox, stbox}):  bool`

```
SELECT hasT(tbox 'TBOX((1.0, 2000-01-01), (3.0, 2000-01-03))');
-- true
SELECT hasT(stbox 'STBOX((1.0, 2.0), (3.0, 4.0))');
-- false
```

- Get the minimum X value

  `Xmin({tbox, stbox}):  float`

```
SELECT Xmin(tbox 'TBOX((1.0, 2000-01-01), (3.0, 2000-01-03))');
-- 1
SELECT Xmin(stbox 'STBOX((1.0, 2.0), (3.0, 4.0))');
-- 1
```

- Get the maximum X value

  `Xmax({tbox, stbox}):  float`

```
SELECT Xmax(tbox 'TBOX((1.0, 2000-01-01), (3.0, 2000-01-03))');
-- 3
SELECT Xmax(stbox 'STBOX((1.0, 2.0), (3.0, 4.0))');
-- 3
```

- Get the minimum Y value

  `Ymin(stbox):  float`

```
SELECT Ymin(stbox 'STBOX((1.0, 2.0), (3.0, 4.0))');
-- 2
```

- Get the maximum Y value

  `Ymax(stbox):  float`

```
SELECT Ymax(stbox 'STBOX((1.0, 2.0), (3.0, 4.0))');
-- 4
```

- Get the minimum Z value

  `Zmin(stbox): float`

```
SELECT Zmin(stbox 'STBOX Z((1.0, 2.0, 3.0), (4.0, 5.0, 6.0))');
-- 3
```

- Get the maximum Z value

  `Zmax(stbox): float`

```
SELECT Zmax(stbox 'STBOX Z((1.0, 2.0, 3.0), (4.0, 5.0, 6.0))');
-- 6
```

- Get the minimum T value

  `Tmin({tbox, stbox}): timestamptz`

```
SELECT Tmin(stbox 'GEODSTBOX T(( , , 2001-01-01), ( , , 2001-01-03))');
-- "2001-01-01"
```

- Get the maximum T value

  `Tmax({tbox, stbox}): timestamptz`

```
SELECT Tmax(stbox 'GEODSTBOX T(( , , 2001-01-01), ( , , 2001-01-03))');
-- "2001-01-03"
```

4.5 Modification Functions

The functions given next expand the bounding boxes on the value and the time dimension or set the precision of the value dimension. These functions raise an error if the corresponding dimension is not present.

- Expand the numeric value dimension of the bounding box by a float value

  `expandValue(tbox, float): tbox`

```
SELECT expandValue(tbox 'TBOX((1,2012-01-01),(2,2012-01-03))', 1);
-- "TBOX((0,2012-01-01),(3,2012-01-03))"
SELECT expandValue(tbox 'TBOX((,2012-01-01),(,2012-01-03))', 1);
-- The box must have value dimension
```

- Expand the spatial value dimension of the bounding box by a float value

  `expandSpatial(stbox, float): stbox`

```
SELECT expandSpatial(stbox 'STBOX ZT((1,1,1,2012-01-01),(2,2,2,2012-01-03))', 1);
-- "STBOX ZT((0,0,0,2012-01-01),(3,3,3,2012-01-03))"
SELECT expandSpatial(stbox 'STBOX T((,2012-01-01),(,2012-01-03))', 1);
-- The box must have XY dimension
```

- Expand the temporal dimension of the bounding box by a time interval

  `expandTemporal({tbox, stbox}, interval): {tbox, stbox}`

```
SELECT expandTemporal(tbox 'TBOX((1,2012-01-01),(2,2012-01-03))', interval '1 day');
-- "TBOX((1,2011-12-31),(2,2012-01-04))"
SELECT expandTemporal(stbox 'STBOX ZT((1,1,1,2012-01-01),(2,2,2,2012-01-03))',
  interval '1 day');
-- "STBOX ZT((1,1,1,2011-12-31),(2,2,2,2012-01-04))"
```

- Round the value or the coordinates of the bounding box to a number of decimal places

  `setPrecision({tbox, stbox}, int): {tbox, stbox}`

```
SELECT setPrecision(tbox 'TBOX((1.12345, 2000-01-01), (2.12345, 2000-01-02))', 2);
-- "TBOX((1.12,2000-01-01),(2.12,2000-01-02))"
SELECT setPrecision(stbox 'STBOX T((1.12345, 1.12345, 2000-01-01),
  (2.12345, 2.12345, 2000-01-02))', 2);
-- "STBOX T((1.12,1.12,2000-01-01),(2.12,2.12,2000-01-02))"
```

4.6 Spatial Reference System Functions

- Get the spatial reference identifier

  `SRID(stbox): int`

```
SELECT SRID(stbox 'STBOX ZT((1.0, 2.0, 3.0, 2000-01-01), (4.0, 5.0, 6.0, 2000-01-02))');
-- 0
SELECT SRID(stbox 'SRID=5676;STBOX T((1.0, 2.0, 2000-01-01), (4.0, 5.0, 2000-01-02))');
-- 5676
SELECT SRID(geodstbox 'GEODSTBOX T((, , 2000-01-01), (, , 2000-01-02))');
-- 4326
```

- Set the spatial reference identifier

  `setSRID(stbox): stbox`

```
SELECT setSRID(stbox 'STBOX ZT((1.0, 2.0, 3.0, 2000-01-01),
(4.0, 5.0, 6.0, 2000-01-02))', 5676);
-- "SRID=5676;STBOX ZT((1,2,3,2000-01-01),(4,5,6,2000-01-02))"
```

- Transform to a different spatial reference

  `transform(stbox, integer): stbox`

```
SELECT transform(stbox 'SRID=4326;STBOX T((2.340088, 49.400250, 2000-01-01),
(6.575317, 51.553167, 2000-01-02))', 3812);
-- "SRID=3812;STBOX T((502773.429980817,511805.120401577,2000-01-01),
(803028.908264815,751590.742628986,2000-01-02))"
```

4.7 Comparison Operators

The traditional comparison operators (=, <, and so on) can be applied to box types. Excepted equality and inequality, the other comparison operators are not useful in the real world but allow B-tree indexes to be constructed on box types. These operators compare first the timestamps and if those are equal, compare the values.

- Are the bounding boxes equal?

```
{tbox, stbox} = {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1, 2012-01-01), (1, 2012-01-04))' =
tbox 'TBOX((2, 2012-01-03), (2, 2012-01-05))';
-- false
```

- Are the bounding boxes different?

```
{tbox, stbox} <> {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1, 2012-01-01), (1, 2012-01-04))' <>
tbox 'TBOX((2, 2012-01-03), (2, 2012-01-05))'
-- true
```

- Is the first bounding box less than the second one?

```
{tbox, stbox} < {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1, 2012-01-01), (1, 2012-01-04))' <
tbox 'TBOX((1, 2012-01-03), (2, 2012-01-05))'
-- true
```

- Is the first bounding box greater than the second one?

```
{tbox, stbox} > {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1, 2012-01-03), (1, 2012-01-04))' >
tbox 'TBOX((1, 2012-01-01), (2, 2012-01-05))'
-- true
```

- Is the first bounding box less than or equal to the second one?

```
{tbox, stbox} <= {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1, 2012-01-01), (1, 2012-01-04))' <=
tbox 'TBOX((2, 2012-01-03), (2, 2012-01-05))'
-- true
```

- Is the first bounding box greater than or equal to the second one?

```
{tbox, stbox} >= {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1, 2012-01-01), (1, 2012-01-04))' >=
tbox 'TBOX((2, 2012-01-03), (2, 2012-01-05))'
-- false
```

4.8 Set Operators

The set operators for box types are union (+) and intersection (*). In the case of union, the operands must have exactly the same dimensions, otherwise an error is raided. Furthermore, if the operands do not overlap on all the dimensions and error is raised, since in this would result in a box with disjoint values, which cannot be represented. The operator computes the union on all dimensions that are present in both arguments. In the case of intersection, the operands must have at least one common dimension, otherwise an error is raised. The operator computes the intersection on all dimensions that are present in both arguments.

- Union of the bounding boxes

```
{tbox, stbox} + {tbox, stbox}:  {tbox, stbox}
```

```
SELECT tbox 'TBOX((1,2001-01-01),(3,2001-01-03))' +
tbox 'TBOX((2,2001-01-02),(4,2001-01-04))';
-- "TBOX((1,2001-01-01),(4,2001-01-04))"
SELECT stbox 'STBOX ZT((1,1,1,2001-01-01),(2,2,2,2001-01-02))' +
stbox 'STBOX T((2,2,2001-01-01),(3,3,2001-01-03))';
-- ERROR:  Boxes must be of the same dimensionality
SELECT tbox 'TBOX((1,2001-01-01),(3,2001-01-02))' +
tbox 'TBOX((2,2001-01-03),(4,2001-01-04))';
-- ERROR:  Result of box union would not be contiguous
```

- Intersection of the bounding boxes

```
{tbox, stbox} * {tbox, stbox}:  {tbox, stbox}
```

```
SELECT tbox 'TBOX((1,2001-01-01),(3,2001-01-03))' *
tbox 'TBOX((,2001-01-02),(,2001-01-04))';
-- "TBOX((,2001-01-02),(,2001-01-03))"
SELECT stbox 'STBOX ZT((1,1,1,2001-01-01),(3,3,3,2001-01-02))' *
stbox 'STBOX((2,2),(4,4))';
-- "STBOX((2,2),(3,3))"
```

4.9 Topological Operators

There are five topological operators: overlaps (&&), contains (@>), contained (<@), same (~=), and adjacent (-|-). The operators verify the topological relationship between the bounding boxes taking into account the value and/or the time dimension for as many dimensions that are present on both arguments.

The topological operators for bounding boxes are given next.

- Do the bounding boxes overlap?

```
{tbox, stbox} && {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1,2001-01-01),(3,2001-01-03))' &&
tbox 'TBOX((2,2001-01-02),(4,2001-01-04))';
-- true
SELECT stbox 'STBOX T((1,1,2001-01-01),(2,2,2001-01-02))'&&
stbox 'STBOX T((,2001-01-02),(,2001-01-02))';
-- true
```

- Does the first bounding box contain the second one?

```
{tbox, stbox} @> {tbox, stbox}:  boolean
```

```
SELECT tbox 'TBOX((1,2001-01-01),(4,2001-01-04))' @>
tbox 'TBOX((2,2001-01-01),(3,2001-01-02))';
-- true
SELECT stbox 'STBOX Z((1,1,1),(3,3,3))' @>
stbox 'STBOX T((1,1,2001-01-01),(2,2,2001-01-02))';
-- true
```

- Is the first bounding box contained in the second one?

  `{tbox, stbox} <@ {tbox, stbox}: boolean`

```
SELECT tbox 'TBOX((1,2001-01-01),(2,2001-01-02))' <@
tbox 'TBOX((1,2001-01-01),(2,2001-01-02))';
-- true
SELECT stbox 'STBOX T((1,1,2001-01-01),(2,2,2001-01-02))' <@
stbox 'STBOX ZT((1,1,1,2001-01-01),(2,2,2,2001-01-02))';
-- true
```

- Are the bounding boxes equal in their common dimensions?

  `{tbox, stbox} ~= {tbox, stbox}: boolean`

```
SELECT tbox 'TBOX((1,2001-01-01),(2,2001-01-02))' ~=
tbox 'TBOX((,2001-01-01),(,2001-01-02))';
-- true
SELECT stbox 'STBOX T((1,1,2001-01-01),(3,3,2001-01-03))' ~=
stbox 'STBOX Z((1,1,1),(3,3,3))';
-- true
```

- Are the bounding boxes adjacent?

  `{tbox, stbox} -|- {tbox, stbox}: boolean`

  Two boxes are adjacent if they share n dimensions and their intersection is at most of n-1 dimensions.

```
SELECT tbox 'TBOX((1,2001-01-01),(2,2001-01-02))' -|-
tbox 'TBOX((,2001-01-02),(,2001-01-03))';
-- true
SELECT stbox 'STBOX T((1,1,2001-01-01),(3,3,2001-01-03))' -|-
stbox 'STBOX T((2,2,2001-01-03),(4,4,2001-01-04))';
-- true
```

4.10 Relative Position Operators

These operators consider the relative position of the bounding boxes. The operators `<<`, `>>`, `&<`, and `&>` consider the X value for the `tbox` type and the X coordinates for the `stbox` type, the operators `<<|`, `|>>`, `&<|`, and `|&>` consider the Y coordinates for the `stbox` type, the operators `<</`, `/>>`, `&</`, and `/&>` consider the Z coordinates for the `stbox` type, and the operators `<<#`, `#>>`, `#&<`, and `#&>` consider the time dimension for the `tbox` and `stbox` types. The operators raise an error if both boxes do not have the required dimension.

The operators for the numeric value dimension of the `tbox` type are given next.

- Are the X values of the first bounding box strictly less than those of the second one?

  `tbox << tbox: boolean`

```
SELECT tbox 'TBOX((1,2012-01-01),(2,2012-01-02))' <<
tbox 'TBOX((3,2012-01-03),(4,2012-01-04))';
-- true
SELECT tbox 'TBOX((1,2012-01-01),(2,2012-01-02))' <<
tbox 'TBOX((,2012-01-03),(,2012-01-04))';
-- ERROR:  Boxes must have X dimension
```

- Are the X values of the first bounding box strictly greater than those of the second one?

  `tbox >> tbox:  boolean`

```
SELECT tbox 'TBOX((3,2012-01-03),(4,2012-01-04))' >>
tbox 'TBOX((1,2012-01-01),(2,2012-01-02))';
-- true
```

- Are the X values of the first bounding box not greater than those of the second one?

  `tbox &< tbox:  boolean`

```
SELECT tbox 'TBOX((1,2012-01-01),(4,2012-01-04))' &<
tbox 'TBOX((3,2012-01-03),(4,2012-01-04))';
-- true
```

- Are the X values of the first bounding box not less than those of the second one?

  `tbox &> tbox:  boolean`

```
SELECT tbox 'TBOX((1,2012-01-01),(2,2012-01-02))' &>
tbox 'TBOX((1,2012-01-01),(4,2012-01-04))';
-- true
```

The operators for the spatial value dimension of the `stbox` type are given next.

- Are the X values of the first bounding box strictly to the left of those of the second one?

  `stbox << stbox:  boolean`

```
SELECT stbox 'STBOX Z((1,1,1),(2,2,2))' << stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
```

- Are the X values of the first bounding box strictly to the right of those of the second one?

  `stbox >> stbox:  boolean`

```
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' >> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
```

- Are the X values of the first bounding box not to the right of those of the second one?

  `stbox &< stbox:  boolean`

```
SELECT stbox 'STBOX Z((1,1,1),(4,4,4))' &< stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
```

- Are the X values of the first bounding box not to the left of those of the second one?

  `stbox &> stbox: boolean`

```
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' &> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
```

- Are the Y values of the first bounding box strictly below of those of the second one?

  `stbox <<| stbox: boolean`

```
SELECT stbox 'STBOX Z((1,1,1),(2,2,2))' <<| stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
```

- Are the Y values of the first bounding box strictly above of those of the second one?

  `stbox |>> stbox: boolean`

```
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' |>> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
```

- Are the Y values of the first bounding box not above of those of the second one?

  `stbox &<| stbox: boolean`

```
SELECT stbox 'STBOX Z((1,1,1),(4,4,4))' &<| stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
```

- Are the Y values of the first bounding box not below of those of the second one?

  `stbox |&> stbox: boolean`

```
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' |&> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- false
```

- Are the Z values of the first bounding box strictly in front of those of the second one?

  `stbox <</ stbox: boolean`

```
SELECT stbox 'STBOX Z((1,1,1),(2,2,2))' <</ stbox 'STBOX Z((3,3,3),(4,4,4))';
```

- Are the Z values of the first bounding box strictly back of those of the second one?

  `stbox />> stbox: boolean`

```
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' />> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
```

- Are the Z values of the first bounding box not back of those of the second one?

  `stbox &</ stbox: boolean`

```
SELECT stbox 'STBOX Z((1,1,1),(4,4,4))' &</ stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
```

- Are the Z values of the first bounding box not in front of those of the second one?

  `stbox /&> stbox: boolean`

```
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' /&> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
```

The operators for the time dimension of the `tbox` and `stbox` types are as follows.

- Are the T values of the first bounding box strictly before those of the second one?

  `{tbox, stbox} <<# {tbox, stbox}: boolean`

```
SELECT tbox 'TBOX((1,2000-01-01),(2,2000-01-02))' <<#
tbox 'TBOX((3,2000-01-03),(4,2000-01-04)))';
-- true
```

- Are the T values of the first bounding box strictly after those of the second one?

  `{tbox, stbox} #>> {tbox, stbox}: boolean`

```
SELECT stbox 'STBOX T((3,3,2000-01-03),(4,4,2000-01-04))'  #>>
stbox 'STBOX T((1,1,2000-01-01),(2,2,2000-01-02))';
-- true
```

- Are the T values of the first bounding box not after those of the second one?

  `{tbox, stbox} &<# {tbox, stbox}: boolean`

```
SELECT tbox 'TBOX((1,2000-01-01),(4,2000-01-04))' &<#
tbox 'TBOX((3,2000-01-03),(4,2000-01-04)))';
-- true
```

- Are the T values of the first bounding box not before those of the second one?

  `{tbox, stbox} #&> {tbox, stbox}: boolean`

```
SELECT stbox 'STBOX T((1,1,2000-01-01),(3,3,2000-01-03))' #&>
stbox 'STBOX T((3,3,2000-01-03),(4,4,2000-01-04))';
-- true
```

4.11 Indexing of Box Types

GiST and SP-GiST indexes can be created for table columns of the `tbox` and `stbox` types. An example of creation of a GiST index in a column `Box` of type `stbox` in a table `Trips` is as follows:

```
CREATE TABLE Trips(TripID integer PRIMARY KEY, Trip tgeompoint, Box stbox);
CREATE INDEX Trips_Box_Idx ON Trips USING GIST(bbox);
```

A GiST or SP-GiST index can accelerate queries involving the following operators: `&&`, `<@`, `@>`, `~=`, `-|-`, `<<`, `>>`, `&<`, `&>`, `<<|`, `|>>`, `&<|`, `|&>`, `<</`, `/>>`, `&</`, `/&>`, `<<#`, `#>>`, `&<#`, and `#&>`.

In addition, B-tree indexes can be created for table columns of a bounding box type. For these index types, basically the only useful operation is equality. There is a B-tree sort ordering defined for values of bounding box types, with corresponding < and > operators, but the ordering is rather arbitrary and not usually useful in the real world. The B-tree support is primarily meant to allow sorting internally in queries, rather than creation of actual indexes.

Chapter 5

Manipulating Temporal Types

We present next the functions and operators for temporal types. These functions and operators are polymorphic, that is, their arguments may be of several types, and the result type may depend on the type of the arguments. To express this, we use the following notation:

- `ttype` represents any temporal type,
- `time` represents any time type, that is, `timestamptz`, `period`, `timestampset`, or `periodset`,
- `tnumber` represents any temporal number type, that is, `tint` or `tfloat`,
- `torder` represents any temporal type whose base type has a total order defined, that is, `tint`, `tfloat`, or `ttext`,
- `tpoint` represents a temporal point type, that is, `tgeompoint` or `tgeogpoint`,
- `ttypeinst` represents any temporal type with instant subtype,
- `ttypei` represents any temporal type with instant set subtype,
- `ttypeseq` represents any temporal type with sequence subtype,
- `tdiscseq` represents any temporal type with sequence subtype and a discrete base type,
- `tcontseq` represents any temporal type with sequence subtype and a continuous base type,
- `ttypes` represents any temporal type with sequence set subtype,
- `base` represents any base type of a temporal type, that is, `bool`, `int`, `float`, `text`, `geometry`, or `geography`,
- `number` represents any number base type, that is, `int` or `float`,
- `numrange` represents any number range type, that is, either `intrange` or `floatrange`,
- `geo` represents the types `geometry` or `geography`,
- `point` represents the types `geometry` or `geography` restricted to a point.
- `type[]` represents an array of `type`.

A common way to generalize the traditional operations to the temporal types is to apply the operation *at each instant*, which yields a temporal value as result. In that case, the operation is only defined on the intersection of the emporal extents of the operands; if the temporal extents are disjoint, then the result is null. For example, the temporal comparison operators, such as `#<`, test whether the values taken by their operands at each instant satisfy the condition and return a temporal Boolean. Examples of the various generalizations of the operators are given next.

```
-- Temporal comparison
SELECT tint '[2@2001-01-01, 2@2001-01-03)' #< tfloat '[1@2001-01-01, 3@2001-01-03)';
-- "{[f@2001-01-01, f@2001-01-02], (t@2001-01-02, t@2001-01-03)}"
SELECT tfloat '[1@2001-01-01, 3@2001-01-03)' #< tfloat '[3@2001-01-03, 1@2001-01-05)';
-- NULL
-- Temporal addition
SELECT tint '[1@2001-01-01, 1@2001-01-03)' + tint '[2@2001-01-02, 2@2001-01-05)';
-- "[3@2001-01-02, 3@2001-01-03)"
-- Temporal intersects
SELECT tintersects(tgeompoint '[Point(0 1)@2001-01-01, Point(3 1)@2001-01-04)',
geometry 'Polygon((1 0,1 2,2 2,2 0,1 0))');
-- "{[f@2001-01-01, t@2001-01-02, t@2001-01-03], (f@2001-01-03, f@2001-01-04]}"
-- Temporal distance
SELECT tgeompoint '[Point(0 0)@2001-01-01 08:00:00, Point(0 1)@2001-01-03 08:10:00)' <->
tgeompoint '[Point(0 0)@2001-01-02 08:05:00, Point(1 1)@2001-01-05 08:15:00)';
-- "[0.5@2001-01-02 08:05:00+00, 0.7451840337945 57@2001-01-03 08:10:00+00)"
```

Another common requirement is to determine whether the operands *ever* or *always* satisfy a condition with respect to an operation. These can be obtained by applying the ever/always comparison operators. These operators are denoted by prefixing the traditional comparison operators with, respectively, `?` (ever) and `%` (always). Examples of ever and always comparison operators are given next.

```
-- Does the operands ever intersect?
SELECT tintersects(tgeompoint '[Point(0 1)@2001-01-01, Point(3 1)@2001-01-04)',
geometry 'Polygon((1 0,1 2,2 2,2 0,1 0))') ?= true;
-- true
-- Does the operands always intersect?
SELECT tintersects(tgeompoint '[Point(0 1)@2001-01-01, Point(3 1)@2001-01-04)',
geometry 'Polygon((0 0,0 2,4 2,4 0,0 0))') %= true;
-- true
-- Is the left operand ever less than the right one ?
SELECT (tfloat '[1@2001-01-01, 3@2001-01-03)' #<
tfloat '[3@2001-01-01, 1@2001-01-03)') ?= true;
-- true
-- Is the left operand always less than the right one ?
SELECT (tfloat '[1@2001-01-01, 3@2001-01-03)' #<
tfloat '[2@2001-01-01, 4@2001-01-03)') %= true;
-- true
```

For efficiency reasons, some common operations with the ever or the always semantics are natively provided. For example, the `intersects` function determines whether there is an instant at which the two arguments spatially intersect.

We describe next the functions and operators for temporal types. For conciseness, in the examples we mostly use sequences composed of two instants.

5.1 Input/Output of Temporal Types

An instant value is a couple of the form `v@t`, where `v` is a value of the base type and `t` is a `timestamptz` value. A sequence value is a set of values `v1@t1,...,vn@tn` delimited by lower and upper bounds, which can be inclusive (represented by '`[`' and '`]`') or exclusive (represented by '`(`' and '`)`'). Examples of input of temporal unit values are as follows:

```
SELECT tbool 'true@2001-01-01 08:00:00';
SELECT tint '1@2001-01-01 08:00:00';
SELECT tfloat '1.5@2001-01-01 08:00:00';
SELECT ttext 'AAA@2001-01-01 08:00:00';
SELECT tgeompoint 'Point(0 0)@2017-01-01 08:00:05';
```

```
SELECT tgeogpoint 'Point(0 0)@2017-01-01 08:00:05';
SELECT tbool '[true@2001-01-01 08:00:00, true@2001-01-03 08:00:00]';
SELECT tint '[1@2001-01-01 08:00:00, 1@2001-01-03 08:00:00]';
SELECT tfloat '[2.5@2001-01-01 08:00:00, 3@2001-01-03 08:00:00, 1@2001-01-04 08:00:00]';
SELECT tfloat '[1.5@2001-01-01 08:00:00]';  -- Instant sequence
SELECT ttext '[BBB@2001-01-01 08:00:00, BBB@2001-01-03 08:00:00]';
SELECT tgeompoint '[Point(0 0)@2017-01-01 08:00:00, Point(0 0)@2017-01-01 08:05:00)';
SELECT tgeogpoint '[Point(0 0)@2017-01-01 08:00:00, Point(0 1)@2017-01-01 08:05:00,
Point(0 0)@2017-01-01 08:10:00)';
```

The temporal extent of an instant value is a single instant while the temporal extent of a sequence value is a period defined by the first and last instants as well as the upper and lower bounds.

A temporal set value is a set `{v1,...,vn}` where every `vi` is a unit value of the corresponding type. Examples of input of temporal set values are as follows:

```
SELECT tbool '{true@2001-01-01 08:00:00, false@2001-01-03 08:00:00}';
SELECT tint '{1@2001-01-01 08:00:00, 2@2001-01-03 08:00:00}';
SELECT tfloat '{1.0@2001-01-01 08:00:00, 2.0@2001-01-03 08:00:00}';
SELECT ttext '{AAA@2001-01-01 08:00:00, BBB@2001-01-03 08:00:00}';
SELECT tgeompoint '{Point(0 0)@2017-01-01 08:00:00, Point(0 1)@2017-01-02 08:05:00}';
SELECT tgeogpoint '{Point(0 0)@2017-01-01 08:00:00, Point(0 1)@2017-01-02 08:05:00}';
SELECT tbool '{[false@2001-01-01 08:00:00, false@2001-01-03 08:00:00),
[true@2001-01-03 08:00:00], (false@2001-01-04 08:00:00, false@2001-01-06 08:00:00]}';
SELECT tint '{[1@2001-01-01 08:00:00, 1@2001-01-03 08:00:00),
[2@2001-01-04 08:00:00, 3@2001-01-05 08:00:00, 3@2001-01-06 08:00:00]}';
SELECT tfloat '{[1@2001-01-01 08:00:00, 2@2001-01-03 08:00:00, 2@2001-01-04 08:00:00,
3@2001-01-06 08:00:00]}';
SELECT ttext '{[AAA@2001-01-01 08:00:00, BBB@2001-01-03 08:00:00, BBB@2001-01-04 08:00:00),
[CCC@2001-01-05 08:00:00, CCC@2001-01-06 08:00:00]}';
SELECT tgeompoint '{[Point(0 0)@2017-01-01 08:00:00, Point(0 1)@2017-01-01 08:05:00),
[Point(0 1)@2017-01-01 08:10:00, Point(1 1)@2017-01-01 08:15:00)}';
SELECT tgeogpoint '{[Point(0 0)@2017-01-01 08:00:00, Point(0 1)@2017-01-01 08:05:00),
[Point(0 1)@2017-01-01 08:10:00, Point(1 1)@2017-01-01 08:15:00)}';
```

The temporal extent of an instant set value is a set of timestamps while the temporal extent of a sequence set value is a set of periods.

Sequence or sequence set values whose base type is continuous may specify that the interpolation is stepwise. If this is not specified, it is supposed that the interpolation is linear by default.

```
-- Linear interpolation by default
SELECT tfloat '[2.5@2001-01-01, 3@2001-01-03, 1@2001-01-04]';
SELECT tgeompoint '{[Point(2.5 2.5)@2001-01-01, Point(3 3)@2001-01-03],
[Point(1 1)@2001-01-04, Point(1 1)@2001-01-04]}';
-- Stepwise interpolation
SELECT tfloat 'Interp=Stepwise;[2.5@2001-01-01, 3@2001-01-03, 1@2001-01-04]';
SELECT tgeompoint 'Interp=Stepwise;{[Point(2.5 2.5)@2001-01-01, Point(3 3)@2001-01-03],
[Point(1 1)@2001-01-04, Point(1 1)@2001-01-04]}';
```

For sequence set values all component sequences are supposed to be in the same interpolation, either stepwise or linear, as in the examples above.

For temporal points, it is possible to specify the spatial reference identifier (SRID) using the Extended Well-Known text (EWKT) representation as follows:

```
SELECT tgeompoint 'SRID=5435;[Point(0 0)@2000-01-01,Point(0 1)@2000-01-02]'
```

All components geometries will then be of the given SRID. Furthermore, each component geometry can specify its SRID with the EWKT format as in the following example

```
SELECT tgeompoint '[SRID=5435;Point(0 0)@2000-01-01,SRID=5435;Point(0 1)@2000-01-02]'
```

An error is raised if the component geometries are not all in the same SRID or if the SRID of a component geometry is different from the one of the temporal point

```
SELECT tgeompoint '[SRID=5435;Point(0 0)@2000-01-01,SRID=4326;Point(0 1)@2000-01-02]';
ERROR: Geometry SRID (4326) does not match temporal type SRID (5435)
SELECT tgeompoint 'SRID=5435;[SRID=4326;Point(0 0)@2000-01-01,
SRID=4326;Point(0 1)@2000-01-02]'
ERROR: Geometry SRID (4326) does not match temporal type SRID (5435)
```

5.2 Constructor Functions

Each temporal type has constructor functions with the same name as the type and with a suffix for the subtype, where the suffix '`inst`', '`i`', '`seq`', and '`s`' correspond, respectively, to the subtypes instant, instant set, sequence, and sequence set. Examples are `tintseq` or `tgeompoints`. Using the constructor function is frequently more convenient than writing a literal constant.

- A first set of functions have two arguments, a base type and a time type, where the latter is a `timestamptz`, a `timestampset`, a `period`, or a `periodset` value for constructing, respectively, an instant, instant set, sequence, or sequence set value. The functions for sequence or sequence set values with continuous base type have in addition an optional third argument which is a Boolean for stating whether the resulting temporal value has linear interpolation or not. By default this argument is true if it is not specified.
- Another set of functions for instant set values have a single argument, which is an array of values of the corresponding instant values.
- Another set of functions for sequence values have one argument for the array of values of the corresponding instant subtype and two optional Boolean arguments stating, respectively, whether the left and right bounds are inclusive or exclusive. It the these arguments are not specified they are assumed to be true by default. In addition, the functions for sequence values with continuous base type have an additional Boolean argument stating whether the interpolation is linear or not. It this argument is not specified it is assumed to be true by default.
- Another set of functions for sequence set values have a single argument, which is an array of values of the corresponding sequence values. For sequence values with continuous base type, the interpolation of the resulting temporal value depends on the interpolation of the composing sequences. An error is raised if the sequences composing the array have different interpolation.

We give next the constructor functions for the various subtypes.

- Constructor for temporal types of instant subtype

  `ttypeinst(base, timestamptz): ttypeinst`

```
SELECT tboolinst(true, '2001-01-01');
SELECT tfloatinst(1.5, '2001-01-01');
SELECT tgeompointinst('Point(0 0)', '2001-01-01');
```

- Constructor for temporal types of instant set subtype

  `ttypei(base, timestampset): ttypei`

  `ttypei(ttypeinst[]): ttypei`

```
SELECT tinti(2, '{2001-01-01, 2001-01-02, 2001-01-03}');
SELECT tgeompointi('Point(0 0)', '{2001-01-01, 2001-01-02}');
SELECT tbooli(ARRAY[tbool 'true@2001-01-01 08:00:00', 'false@2001-01-01 08:05:00']);
SELECT tinti(ARRAY[tint '1@2001-01-01 08:00:00', '2@2001-01-01 08:05:00']);
SELECT tfloati(ARRAY[tfloat '1.0@2001-01-01 08:00:00', '2.0@2001-01-01 08:05:00']);
SELECT ttexti(ARRAY[ttext 'AAA@2001-01-01 08:00:00', 'BBB@2001-01-01 08:05:00']);
SELECT tgeompointi(ARRAY[tgeompoint 'Point(0 0)@2001-01-01 08:00:00',
'Point(0 1)@2001-01-01 08:05:00', 'Point(1 1)@2001-01-01 08:10:00']);
SELECT tgeogpointi(ARRAY[tgeogpoint 'Point(1 1)@2001-01-01 08:00:00',
'Point(2 2)@2001-01-01 08:05:00']);
```

- Constructor for temporal types of sequence subtype

  `tdiscseq(base, period):  tdiscseq`

  `tdiscseq(ttypeinst[], left_inc = true, right_inc = true}):  tdiscseq`

  `tcontseq(base, period, linear = true):  tcontseq`

  `tcontseq(ttypeinst[], left_inc = true, right_inc = true, linear = true}):  tcontseq`

```
SELECT tfloatseq(1.5, '[2001-01-01, 2001-01-02]');
SELECT tfloatseq(2.0, '[2001-01-01, 2001-01-02]', false);
SELECT tboolseq(ARRAY[tbool 'true@2001-01-01 08:00:00', 'true@2001-01-03 08:05:00'],
true, true);
SELECT tintseq(ARRAY[tintinst(2,'2001-01-01 08:00:00'),
tintinst(2,'2001-01-01 08:10:00')], true, false);
SELECT tfloatseq(ARRAY[tfloat '2.0@2001-01-01 08:00:00', '3@2001-01-03 08:05:00',
'1@2001-01-03 08:10:00'], true, false);
SELECT tfloatseq(ARRAY[tfloat '2.0@2001-01-01 08:00:00', '3@2001-01-03 08:05:00',
'1@2001-01-03 08:10:00'], true, true, false);
SELECT ttextseq(ARRAY[ttextinst('AAA', '2001-01-01 08:00:00'),
ttextinst('BBB', '2001-01-03 08:05:00'), ttextinst('BBB', '2001-01-03 08:10:00')]);
SELECT tgeompointseq(ARRAY[tgeompoint 'Point(0 0)@2001-01-01 08:00:00',
'Point(0 1)@2001-01-03 08:05:00', 'Point(1 1)@2001-01-03 08:10:00']);
SELECT tgeogpointseq(ARRAY[tgeogpoint 'Point(0 0)@2001-01-01 08:00:00',
'Point(0 0)@2001-01-03 08:05:00'], true, true, false);
```

- Constructors for temporal types of sequence set subtype

  `tdiscs(base, periodset):  tdiscs`

  `tconts(base, periodset, linear = true):  tconts`

  `ttypes(ttypeseq[]):  ttypes`

```
SELECT ttexts('AAA', '{[2001-01-01, 2001-01-02], [2001-01-03, 2001-01-04]}');
SELECT tgeogpointseq('Point(1 1)', '[2001-01-01, 2001-01-02]', false);
SELECT tbools(ARRAY[tbool '[false@2001-01-01 08:00:00, false@2001-01-01 08:05:00)',
'[true@2001-01-01 08:05:00]','(false@2001-01-01 08:05:00, false@2001-01-01 08:10:00)']);
SELECT tints(ARRAY[tint '[1@2001-01-01 08:00:00, 2@2001-01-01 08:05:00,
2@2001-01-01 08:10:00, 2@2001-01-01 08:15:00)']);
SELECT tfloats(ARRAY[tfloat '[1.0@2001-01-01 08:00:00, 2.0@2001-01-01 08:05:00,
2.0@2001-01-01 08:10:00]', '[2.0@2001-01-01 08:15:00, 3.0@2001-01-01 08:20:00)']);
SELECT tfloats(ARRAY[tfloat 'Interp=Stepwise;[1.0@2001-01-01 08:00:00,
2.0@2001-01-01 08:05:00, 2.0@2001-01-01 08:10:00]',
'Interp=Stepwise;[3.0@2001-01-01 08:15:00, 3.0@2001-01-01 08:20:00)']);
SELECT ttexts(ARRAY[ttext '[AAA@2001-01-01 08:00:00, AAA@2001-01-01 08:05:00)',
'[BBB@2001-01-01 08:10:00, BBB@2001-01-01 08:15:00)']);
SELECT tgeompoints(ARRAY[tgeompoint '[Point(0 0)@2001-01-01 08:00:00,
Point(0 1)@2001-01-01 08:05:00, Point(0 1)@2001-01-01 08:10:00)',
'[Point(0 1)@2001-01-01 08:15:00, Point(0 0)@2001-01-01 08:20:00)']);
```

```
SELECT tgeogpoints(ARRAY[tgeogpoint
'Interp=Stepwise;[Point(0 0)@2001-01-01 08:00:00, Point(0 0)@2001-01-01 08:05:00)',
'Interp=Stepwise;[Point(1 1)@2001-01-01 08:10:00, Point(1 1)@2001-01-01 08:15:00)']);
SELECT tfloats(ARRAY[tfloat 'Interp=Stepwise;[1.0@2001-01-01 08:00:00,
2.0@2001-01-01 08:05:00, 2.0@2001-01-01 08:10:00]',
'[3.0@2001-01-01 08:15:00, 3.0@2001-01-01 08:20:00)']);
-- ERROR:  Input sequences must have the same interpolation
```

5.3 Casting

A temporal value can be converted into a temporal value of a compatible type. This can be done using the notation `CAST(ttype1 AS ttype2)` or `ttype1::ttype2`.

- Cast a temporal integer to a temporal float

  `tint::tfloat`

```
SELECT tint '[1@2001-01-01, 2@2001-01-03]'::tfloat;
-- "[1@2001-01-01 00:00:00+00, 2@2001-01-03 00:00:00+00]"
SELECT tint '[1@2000-01-01, 2@2000-01-03, 3@2000-01-05]'::tfloat;
-- "Interp=Stepwise;[1@2000-01-01, 2@2000-01-03, 3@2000-01-05]"
```

- Cast a temporal float to a temporal integer

  `tfloat::tint`

```
SELECT tfloat 'Interp=Stepwise;[1.5@2001-01-01, 2.5@2001-01-03]'::tint;
-- "[1@2001-01-01 00:00:00+00, 2@2001-01-03 00:00:00+00]"
SELECT tfloat '[1.5@2001-01-01, 2.5@2001-01-03]'::tint;
-- ERROR:  Cannot cast temporal float with linear interpolation to temporal integer
```

- Cast a temporal geometry point to a temporal geography point

  `tgeompoint::tgeogpoint`

```
SELECT asText((tgeogpoint 'Point(0 0)@2001-01-01')::tgeompoint);
-- "{POINT(0 0)@2001-01-01}"
```

- Cast a temporal geography point to a temporal geometry point

  `tgeogpoint::tgeompoint`

```
SELECT asText((tgeompoint '[Point(0 0)@2001-01-01, Point(0 1)@2001-01-02)')::tgeogpoint);
-- "{[POINT(0 0)@2001-01-01, POINT(0 1)@2001-01-02)}"
```

A common way to store temporal points in PostGIS is to represent them as geometries of type `LINESTRING M` and abuse the M dimension to encode timestamps as seconds since 1970-01-01 00:00:00. These time-enhanced geometries, called *trajectories*, can be validated with the function `ST_IsValidTrajectory` to verify that the M value is growing from each vertex to the next. Trajectories can be manipulated with the functions `ST_ClosestPointOfApproach`, `ST_DistanceCPA`, and `ST_CPAWithin`. Temporal point values can be converted to/from PostGIS trajectories.

- Cast a temporal point to a PostGIS trajectory

  `tgeompoint::geometry`

  `tgeogpoint::geography`

```
SELECT ST_AsText((tgeompoint 'Point(0 0)@2001-01-01')::geometry);
-- "POINT M (0 0 978307200)"
SELECT ST_AsText((tgeompoint '{Point(0 0)@2001-01-01, Point(1 1)@2001-01-02,
Point(1 1)@2001-01-03}')::geometry);
-- "MULTIPOINT M (0 0 978307200,1 1 978393600,1 1 978480000)"
SELECT ST_AsText((tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02)')::geometry);
-- "LINESTRING M (0 0 978307200,1 1 978393600)"
SELECT ST_AsText((tgeompoint '{[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02),
[Point(1 1)@2001-01-03, Point(1 1)@2001-01-04),
[Point(1 1)@2001-01-05, Point(0 0)@2001-01-06)}')::geometry);
-- "MULTILINESTRING M ((0 0 978307200,1 1 978393600),(1 1 978480000,1 1 978566400),
(1 1 978652800,0 0 978739200))"
SELECT ST_AsText((tgeompoint '{[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02),
[Point(1 1)@2001-01-03],
[Point(1 1)@2001-01-05, Point(0 0)@2001-01-06)}')::geometry);
-- "GEOMETRYCOLLECTION M (LINESTRING M (0 0 978307200,1 1 978393600),
POINT M (1 1 978480000),LINESTRING M (1 1 978652800,0 0 978739200))"
```

- Cast a PostGIS trajectory to a temporal point

  `geometry::tgeompoint`

  `geography::tgeogpoint`

```
SELECT asText(geometry 'LINESTRING M (0 0 978307200,0 1 978393600,
1 1 978480000)'::tgeompoint);
-- "[POINT(0 0)@2001-01-01, POINT(0 1)@2001-01-02, POINT(1 1)@2001-01-03]";
SELECT asText(geometry 'GEOMETRYCOLLECTION M (LINESTRING M (0 0 978307200,1 1 978393600),
POINT M (1 1 978480000),LINESTRING M (1 1 978652800,0 0 978739200))'::tgeompoint);
-- "{[POINT(0 0)@2001-01-01, POINT(1 1)@2001-01-02], [POINT(1 1)@2001-01-03],
[POINT(1 1)@2001-01-05, POINT(0 0)@2001-01-06]}"
```

5.4 Transformation Functions

A temporal value can be transformed to another subtype. An error is raised if the subtypes are incompatible.

- Transform a temporal value to another subtype

  `ttypeinst(ttype): ttypeinst`

  `ttypei(ttype): ttypei`

  `ttypeseq(ttype): ttypeseq`

  `ttypes(ttype): ttypes`

```
SELECT tboolinst(tbool '{[true@2001-01-01]}');
-- "t@2001-01-01 00:00:00+00"
SELECT tboolinst(tbool '{[true@2001-01-01, true@2001-01-02]}');
-- ERROR: Cannot transform input to a temporal instant
SELECT tbooli(tbool 'true@2001-01-01');
-- "{t@2001-01-01}"
SELECT tintseq(tint '1@2001-01-01');
```

```
-- "[1@2001-01-01]"
SELECT tfloats(tfloat '2.5@2001-01-01');
-- "{[2.5@2001-01-01]}"
SELECT tfloats(tfloat '{2.5@2001-01-01, 1.5@2001-01-02, 3.5@2001-01-02}');
-- "{[2.5@2001-01-01],[1.5@2001-01-02],[3.5@2001-01-03]}"
```

- Transform a temporal value with continuous base type from stepwise to linear interpolation

  `toLinear(ttype) : ttype`

```
SELECT toLinear(tfloat 'Interp=Stepwise;[1@2000-01-01, 2@2000-01-02,
1@2000-01-03, 2@2000-01-04]');
-- "{[1@2000-01-01, 1@2000-01-02), [2@2000-01-02, 2@2000-01-03),
[1@2000-01-03, 1@2000-01-04), [2@2000-01-04]}"
SELECT asText(toLinear(tgeompoint 'Interp=Stepwise;{[Point(1 1)@2000-01-01,
Point(2 2)@2000-01-02], [Point(3 3)@2000-01-05, Point(4 4)@2000-01-06]}'));
-- "{[POINT(1 1)@2000-01-01, POINT(1 1)@2000-01-02), [POINT(2 2)@2000-01-02],
[POINT(3 3)@2000-01-05, POINT(3 3)@2000-01-06), [POINT(4 4)@2000-01-06]}"
```

- Append a temporal instant to a temporal value

  `appendInstant(ttype, ttypeinst) : ttype`

```
SELECT appendInstant(tint '1@2000-01-01', tint '1@2000-01-02');
-- "{1@2000-01-01, 1@2000-01-02}"
SELECT appendInstant(tintseq(tint '1@2000-01-01'), tint '1@2000-01-02');
-- "[1@2000-01-01, 1@2000-01-02]"
SELECT asText(appendInstant(tgeompoint '{[Point(1 1 1)@2000-01-01,
Point(2 2 2)@2000-01-02], [Point(3 3 3)@2000-01-04, Point(3 3 3)@2000-01-05]}',
tgeompoint 'Point(1 1 1)@2000-01-06'));
-- "{[POINT Z (1 1 1)@2000-01-01, POINT Z (2 2 2)@2000-01-02],
[POINT Z (3 3 3)@2000-01-04, POINT Z (3 3 3)@2000-01-05, POINT Z (1 1 1)@2000-01-06]}"
```

- Merge the temporal values

  `merge(ttype, ttype) : ttype`

  `merge(ttype[]) : ttype`

  The values may share a single timestamp, in that case the temporal values are joined in the result if their value at the common timestamp is the same, otherwise an error is raised.

```
SELECT merge(tint '1@2000-01-01', tint '1@2000-01-02');
-- "{1@2000-01-01, 1@2000-01-02}"
SELECT merge(tint '[1@2000-01-01, 2@2000-01-02]', tint '[2@2000-01-02, 1@2000-01-03]');
-- "[1@2000-01-01, 2@2000-01-02, 1@2000-01-03]"
SELECT merge(tint '[1@2000-01-01, 2@2000-01-02]', tint '[3@2000-01-03, 1@2000-01-04]');
-- "{[1@2000-01-01, 2@2000-01-02], [3@2000-01-03, 1@2000-01-04]}"
SELECT merge(tint '[1@2000-01-01, 2@2000-01-02]', tint '[1@2000-01-02, 2@2000-01-03]');
-- ERROR:  Both arguments have different value at their overlapping timestamp
SELECT asText(merge(tgeompoint '{[Point(1 1 1)@2000-01-01,
Point(2 2 2)@2000-01-02], [Point(3 3 3)@2000-01-04, Point(3 3 3)@2000-01-05]}',
tgeompoint '{[Point(3 3 3)@2000-01-05, Point(1 1 1)@2000-01-06]}'));
-- "{[POINT Z (1 1 1)@2000-01-01, POINT Z (2 2 2)@2000-01-02],
[POINT Z (3 3 3)@2000-01-04, POINT Z (3 3 3)@2000-01-05, POINT Z (1 1 1)@2000-01-06]}"

SELECT merge(ARRAY[tint '1@2000-01-01', '1@2000-01-02']);
-- "{1@2000-01-01, 1@2000-01-02}"
SELECT merge(ARRAY[tint '{1@2000-01-01, 2@2000-01-02}', '{2@2000-01-02, 3@2000-01-03}']);
-- "{1@2000-01-01, 2@2000-01-02, 3@2000-01-03}"
```

```
SELECT merge(ARRAY[tint '{1@2000-01-01, 2@2000-01-02}', '{3@2000-01-03, 4@2000-01-04}']);
-- "{1@2000-01-01, 2@2000-01-02, 3@2000-01-03, 4@2000-01-04}"
SELECT merge(ARRAY[tint '[1@2000-01-01, 2@2000-01-02]', '[2@2000-01-02, 1@2000-01-03]']);
-- "[1@2000-01-01, 2@2000-01-02, 1@2000-01-03]"
SELECT merge(ARRAY[tint '[1@2000-01-01, 2@2000-01-02]', '[3@2000-01-03, 4@2000-01-04]']);
-- "{[1@2000-01-01, 2@2000-01-02], [3@2000-01-03, 4@2000-01-04]}"
SELECT merge(ARRAY[tgeompoint '{[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02],
[Point(3 3)@2000-01-03, Point(4 4)@2000-01-04]}', '{[Point(4 4)@2000-01-04,
Point(3 3)@2000-01-05], [Point(6 6)@2000-01-06, Point(7 7)@2000-01-07]}']);
-- "{[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02], [Point(3 3)@2000-01-03,
Point(4 4)@2000-01-04, Point(3 3)@2000-01-05],
[Point(6 6)@2000-01-06, Point(7 7)@2000-01-07]}"
SELECT merge(ARRAY[tgeompoint '{[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02]}',
'{[Point(2 2)@2000-01-02, Point(1 1)@2000-01-03]}']);
-- "[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02, Point(1 1)@2000-01-03]"
```

5.5 Accessor Functions

- Get the memory size in bytes

  `memSize(ttype): integer`

```
SELECT memSize(tint '{1@2012-01-01, 2@2012-01-02, 3@2012-01-03}');
-- 280
```

- Get the temporal type

  `tempSubtype(ttype): {'Instant', 'InstantSet', 'Sequence', 'SequenceSet'}`

```
SELECT tempSubtype(tint '[1@2012-01-01, 2@2012-01-02, 3@2012-01-03]');
-- "Sequence"
```

- Get the interpolation

  `interpolation(ttype): {'Discrete', 'Stepwise', 'Linear'}`

```
SELECT interpolation(tfloat '{1@2012-01-01, 2@2012-01-02, 3@2012-01-03}');
-- "Discrete"
SELECT interpolation(tint '[1@2012-01-01, 2@2012-01-02, 3@2012-01-03]');
-- "Stepwise"
SELECT interpolation(tfloat '[1@2012-01-01, 2@2012-01-02, 3@2012-01-03]');
-- "Linear"
SELECT interpolation(tfloat 'Interp=Stepwise;[1@2012-01-01, 2@2012-01-02, 3@2012-01-03]');
-- "Stepwise"
SELECT interpolation(tgeompoint 'Interp=Stepwise;[Point(1 1)@2012-01-01, Point(2 2)@2012 ↩
    -01-02, Point(3 3)@2012-01-03]');
-- "Stepwise"
```

- Get the value

  `getValue(ttypeinst): base`

```
SELECT getValue(tint '1@2012-01-01');
-- 1
SELECT ST_AsText(getValue(tgeompoint 'Point(0 0)@2012-01-01'));
-- "POINT(0 0)"
```

- Get the values

  `getValues(ttype): {base[], floatrange[], geo}`

```
SELECT getValues(tint '[1@2012-01-01, 2@2012-01-03]');
-- "{1,2}"
SELECT getValues(tfloat '[1@2012-01-01, 2@2012-01-03)');
-- "{[1,2)}"
SELECT getValues(tfloat '{[1@2012-01-01, 2@2012-01-03), [3@2012-01-03, 4@2012-01-05)}');
-- "{[1,2),[3,4)}"
SELECT getValues(tfloat 'Interp=Stepwise;{[1@2012-01-01, 2@2012-01-02],
[3@2012-01-03, 4@2012-01-05]}');
-- "{"[1,1]","[2,2]","[3,3]","[4,4]"}"
SELECT ST_AsText(getValues(tgeompoint '{[Point(0 0)@2012-01-01, Point(0 1)@2012-01-02),
[Point(0 1)@2012-01-03, Point(1 1)@2012-01-04)}'));
-- "LINESTRING(0 0,0 1,1 1)"
SELECT ST_AsText(getValues(tgeompoint '{[Point(0 0)@2012-01-01, Point(0 1)@2012-01-02),
[Point(1 1)@2012-01-03, Point(2 2)@2012-01-04)}'));
-- "MULTILINESTRING((0 0,0 1),(1 1,2 2)"
SELECT ST_AsText(getValues(tgeompoint 'Interp=Stepwise;{[Point(0 0)@2012-01-01,
Point(0 1)@2012-01-02], [Point(0 1)@2012-01-03, Point(1 1)@2012-01-04]}'));
-- "GEOMETRYCOLLECTION(MULTIPOINT(0 0,0 1),MULTIPOINT(0 1,1 1))"
SELECT ST_AsText(getValues(tgeompoint '{Point(0 0)@2012-01-01, Point(0 1)@2012-01-02}'));
-- "MULTIPOINT(0 0,0 1)"
SELECT ST_AsText(getValues(tgeompoint '{[Point(0 0)@2012-01-01, Point(0 1)@2012-01-02),
[Point(1 1)@2012-01-03, Point(1 1)@2012-01-04),
[Point(2 1)@2012-01-05, Point(2 2)@2012-01-06)}'));
-- "GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(0 0,0 1),LINESTRING(2 1,2 2))"
```

- Get the start value

  `startValue(ttype): base`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT startValue(tfloat '(1@2012-01-01, 2@2012-01-03)');
-- 1
```

- Get the end value

  `endValue(ttype): base`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT endValue(tfloat '{[1@2012-01-01, 2@2012-01-03), [3@2012-01-03, 5@2012-01-05)}');
-- 5
```

- Get the minimum value

  `minValue(torder): base`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT minValue(tfloat '{1@2012-01-01, 2@2012-01-03, 3@2012-01-05}');
-- 1
```

- Get the maximum value

  `maxValue(torder): base`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT maxValue(tfloat '{[1@2012-01-01, 2@2012-01-03), [3@2012-01-03, 5@2012-01-05)}');
-- 5
```

- Get the value range

  `valueRange(tnumber):  numrange`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT valueRange(tfloat '{[2@2012-01-01, 1@2012-01-03), [4@2012-01-03, 6@2012-01-05)}');
-- "[1,6]"
SELECT valueRange(tfloat '{1@2012-01-01, 2@2012-01-03, 3@2012-01-05}');
-- "[1,3])"
```

- Get the value at a timestamp

  `valueAtTimestamp(ttype, timestamptz):  base`

```
SELECT valueAtTimestamp(tfloat '[1@2012-01-01, 4@2012-01-04)', '2012-01-02');
-- "2"
```

- Get the timestamp

  `getTimestamp(ttypeinst):  timestamptz`

```
SELECT getTimestamp(tint '1@2012-01-01');
-- "2012-01-01"
```

- Get the time

  `getTime(ttype):  periodset`

```
SELECT getTime(tint '[1@2012-01-01, 1@2012-01-15)');
-- "{[2012-01-01, 2012-01-15)}"
```

- Get the duration

  `duration(ttype):  interval`

```
SELECT duration(period '[2012-01-01, 2012-01-03)');
-- "2 days"
SELECT duration(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-05)}');
-- "3 days"
```

- Get the timespan ignoring the potential time gaps

  `timespan(ttype):  interval`

```
SELECT timespan(timestampset '{2012-01-01, 2012-01-03}');
-- "2 days"
SELECT timespan(periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-05)}');
-- "4 days"
```

- Get the period on which the temporal value is defined ignoring the potential time gaps

  `period(ttype): period`

```
SELECT period(tint '{1@2012-01-01, 2@2012-01-03, 3@2012-01-05}');
-- "[2012-01-01, 2012-01-05]"
SELECT period(tfloat '{[1@2012-01-01, 1@2012-01-02), [2@2012-01-03, 3@2012-01-04)}');
-- "[2012-01-01, 2012-01-04)"
```

- Get the number of different instants

  `numInstants(ttype): int`

```
SELECT numInstants(tfloat '{[1@2000-01-01, 2@2000-01-02), (2@2000-01-02, 3@2000-01-03)}');
-- 3
```

- Get the start instant

  `startInstant(ttype): ttypeinst`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT startInstant(tfloat '{[1@2000-01-01, 2@2000-01-02),
(2@2000-01-02, 3@2000-01-03)}');
-- "1@2000-01-01"
```

- Get the end instant

  `endInstant(ttype): ttypeinst`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT endInstant(tfloat '{[1@2000-01-01, 2@2000-01-02), (2@2000-01-02, 3@2000-01-03)}');
-- "3@2000-01-03"
```

- Get the n-th different instant

  `instantN(ttype, int): ttypeinst`

```
SELECT instantN(tfloat '{[1@2000-01-01, 2@2000-01-02), (2@2000-01-02, 3@2000-01-03)}', 3);
-- "3@2000-01-03"
```

- Get the different instants

  `instants(ttype): ttypeinst[]`

```
SELECT instants(tfloat '{[1@2000-01-01, 2@2000-01-02), (2@2000-01-02, 3@2000-01-03)}');
-- "{"1@2000-01-01","2@2000-01-02","3@2000-01-03"}"
```

- Get the number of different timestamps

  `numTimestamps(ttype): int`

```
SELECT numTimestamps(tfloat '{[1@2012-01-01, 2@2012-01-03),
[3@2012-01-03, 5@2012-01-05)}');
-- 3
```

- Get the start timestamp

  `startTimestamp(ttype): timestamptz`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT startTimestamp(tfloat '[1@2012-01-01, 2@2012-01-03)');
-- "2012-01-01"
```

- Get the end timestamp

  `endTimestamp(ttype): timestamptz`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT endTimestamp(tfloat '{[1@2012-01-01, 2@2012-01-03),
[3@2012-01-03, 5@2012-01-05)}');
-- "2012-01-05"
```

- Get the n-th different timestamp

  `timestampN(ttype, int): timestamptz`

```
SELECT timestampN(tfloat '{[1@2012-01-01, 2@2012-01-03),
[3@2012-01-03, 5@2012-01-05)}', 3);
-- "2012-01-05"
```

- Get the different timestamps

  `timestamps(ttype): timestamptz[]`

```
SELECT timestamps(tfloat '{[1@2012-01-01, 2@2012-01-03), [3@2012-01-03, 5@2012-01-05)}');
-- "{"2012-01-01", "2012-01-03", "2012-01-05"}"
```

- Get the number of sequences

  `numSequences({ttypeseq,ttypes}): int`

```
SELECT numSequences(tfloat '{[1@2012-01-01, 2@2012-01-03),
[3@2012-01-03, 5@2012-01-05)}');
-- 2
```

- Get the start sequence

  `startSequence({ttypeseq,ttypes}): ttypeseq`

```
SELECT startSequence(tfloat '{[1@2012-01-01, 2@2012-01-03),
[3@2012-01-03, 5@2012-01-05)}');
-- "[1@2012-01-01, 2@2012-01-03)"
```

- Get the end sequence

  `endSequence({ttypeseq,ttypes}): ttypeseq`

```
SELECT endSequence(tfloat '{[1@2012-01-01, 2@2012-01-03), [3@2012-01-03, 5@2012-01-05)}');
-- "[3@2012-01-03, 5@2012-01-05)"
```

- Get the n-th sequence

  `sequenceN({ttypeseq,ttypes}, int): ttypeseq`

```
SELECT sequenceN(tfloat '{[1@2012-01-01, 2@2012-01-03),
[3@2012-01-03, 5@2012-01-05)}', 2);
-- "[3@2012-01-03, 5@2012-01-05)"
```

- Get the sequences

  `sequences({ttypeseq,ttypes}): ttypeseq[]`

```
SELECT sequences(tfloat '{[1@2012-01-01, 2@2012-01-03), [3@2012-01-03, 5@2012-01-05)}');
-- "{"[1@2012-01-01, 2@2012-01-03)", "[3@2012-01-03, 5@2012-01-05)"}"
```

- Get the segments

  `segments({ttypeseq,ttypes}): ttypeseq[]`

```
SELECT segments(tint '{[1@2012-01-01, 3@2012-01-02, 2@2012-01-03],
  (3@2012-01-03, 5@2012-01-05]}');
-- {"[1@2012-01-01, 1@2012-01-02)","[3@2012-01-02, 3@2012-01-03)","[2@2012-01-03]",
  "(3@2012-01-03, 3@2012-01-05)","[5@2012-01-05]"}
SELECT segments(tfloat '{[1@2012-01-01, 3@2012-01-02, 2@2012-01-03],
  (3@2012-01-03, 5@2012-01-05]}');
-- {"[1@2012-01-01, 3@2012-01-02)","[3@2012-01-02, 2@2012-01-03]",
  "(3@2012-01-03, 5@2012-01-05]"}
```

- Shift the timespan of the temporal value by an interval

  `shift(ttype, interval): ttype`

```
SELECT shift(tint '{1@2001-01-01, 2@2001-01-03, 1@2001-01-05}', '1 day');
-- "{1@2001-01-02, 2@2001-01-04, 1@2001-01-06}"
SELECT shift(tfloat '[1@2001-01-01, 2@2001-01-03]', '1 day');
-- "[1@2001-01-02, 2@2001-01-04]"
SELECT asText(shift(tgeompoint '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-03],
  [Point(2 2)@2001-01-04, Point(1 1)@2001-01-05]}', '1 day'));
-- "{[POINT(1 1)@2001-01-02, POINT(2 2)@2001-01-04],
  [POINT(2 2)@2001-01-05, POINT(1 1)@2001-01-06]}"
```

- Scale the time span of the temporal value to an interval. If the time span of the temporal value is zero (for example, for a temporal instant), the result is the temporal value. The given interval must be strictly greater than zero.

  `tscale(ttype, interval): ttype`

```
SELECT tscale(tint '1@2001-01-01', '1 day');
-- "1@2001-01-01"
SELECT tscale(tint '{1@2001-01-01, 2@2001-01-03, 1@2001-01-05}', '1 day');
-- "{1@2001-01-01 00:00:00+01, 2@2001-01-01 12:00:00+01, 1@2001-01-02 00:00:00+01}"
SELECT tscale(tfloat '[1@2001-01-01, 2@2001-01-03]', '1 day');
-- "[1@2001-01-01, 2@2001-01-02]"
SELECT asText(tscale(tgeompoint '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02,
  Point(1 1)@2001-01-03], [Point(2 2)@2001-01-04, Point(1 1)@2001-01-05]}', '1 day'));
-- "{[POINT(1 1)@2001-01-01 00:00:00+01, POINT(2 2)@2001-01-01 06:00:00+01,
  POINT(1 1)@2001-01-01 12:00:00+01], [POINT(2 2) @2001-01-01 18:00:00+01,
  POINT(1 1)@2001-01-02 00:00:00+01]}"
SELECT tscale(tint '1@2001-01-01', '-1 day');
-- ERROR:  The duration must be a positive interval: -1 days
```

- Shift and scale the time span of the temporal value to the two intervals. This function combines in a single step the functions shift and tscale.

```
shiftTscale(ttype, interval, interval):  ttype
```

```
SELECT shiftTscale(tint '1@2001-01-01', '1 day', '1 day');
-- "1@2001-01-02"
SELECT shiftTscale(tint '{1@2001-01-01, 2@2001-01-03, 1@2001-01-05}', '1 day', '1 day');
-- "{1@2001-01-02 00:00:00+01, 2@2001-01-02 12:00:00+01, 1@2001-01-03 00:00:00+01}"
SELECT shiftTscale(tfloat '[1@2001-01-01, 2@2001-01-03]', '1 day', '1 day');
-- "[1@2001-01-02, 2@2001-01-03]"
SELECT asText(shiftTscale(tgeompoint '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02,
  Point(1 1)@2001-01-03], [Point(2 2)@2001-01-04, Point(1 1)@2001-01-05]}',
  '1 day', '1 day'));
-- "{[POINT(1 1)@2001-01-02 00:00:00+01, POINT(2 2)@2001-01-02 06:00:00+01,
  POINT(1 1)@2001-01-02 12:00:00+01], [POINT(2 2) @2001-01-02 18:00:00+01,
  POINT(1 1)@2001-01-03 00:00:00+01]}"
```

- Does the temporal value intersect the timestamp?

```
intersectsTimestamp(ttype, timestamptz):  boolean
```

```
SELECT intersectsTimestamp(tint '[1@2012-01-01, 1@2012-01-15)', timestamptz '2012-01-03');
-- true
```

- Does the temporal value intersect the timestamp set?

```
intersectsTimestampSet(ttype, timestampset):  boolean
```

```
SELECT intersectsTimestampSet(tint '[1@2012-01-01, 1@2012-01-15)',
  timestampset '{2012-01-01, 2012-01-03}');
-- true
```

- Does the temporal value intersect the period?

```
intersectsPeriod(ttype, period):  boolean
```

```
SELECT intersectsPeriod(tint '[1@2012-01-01, 1@2012-01-04)',
  period '[2012-01-01,2012-01-05)');
-- true
```

- Does the temporal value intersect the period set?

```
intersectsPeriodSet(ttype, periodset):  boolean
```

```
SELECT intersectsPeriodSet(tbool '[t@2012-01-01, f@2012-01-15]',
  periodset '{[2012-01-01, 2012-01-03), [2012-01-05, 2012-01-07)}');
-- true
```

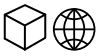
- Get the time-weighted average

```
twAvg(tnumber):  float
```

```
SELECT twAvg(tfloat '{[1@2012-01-01, 2@2012-01-03), [2@2012-01-04, 2@2012-01-06)}');
-- 1.75
```

5.6 Spatial Functions

In the following, we specify with the symbol  that the function supports 3D points and with the symbol  that the function is available for geographies.

- Get the Well-Known Text (WKT) representation 

  `asText({tpoint, tpoint[], geo[]}): {text, text[]}`

```
SELECT asText(tgeompoint 'SRID=4326;[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-02)');
-- "[POINT Z (0 0 0)@2012-01-01 00:00:00+00, POINT Z (1 1 1)@2012-01-02 00:00:00+00)"
SELECT asText(ARRAY[geometry 'Point(0 0)', 'Point(1 1)']);
-- "{"POINT(0 0)","POINT(1 1)"}"
```

- Get the Extended Well-Known Text (EWKT) representation 

  `asEWKT({tpoint, tpoint[], geo[]}): {text, text[]}`

```
SELECT asEWKT(tgeompoint 'SRID=4326;[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-02)');
-- "SRID=4326;[POINT Z (0 0 0)@2012-01-01 00:00:00+00,
POINT Z (1 1 1)@2012-01-02 00:00:00+00)"
SELECT asEWKT(ARRAY[geometry 'SRID=5676;Point(0 0)', 'SRID=5676;Point(1 1)']);
-- "{"SRID=5676;POINT(0 0)","SRID=5676;POINT(1 1)"}"
```

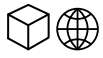
- Get the Moving Features JSON representation 

  `asMFJSON(tpoint, maxdecdigits int4 DEFAULT 15, options int4 DEFAULT 0): bytea`

  The last `options` argument could be used to add BBOX and/or CRS in MFJSON output:

  - 0: means no option (default value)
  - 1: MFJSON BBOX
  - 2: MFJSON Short CRS (e.g EPSG:4326)
  - 4: MFJSON Long CRS (e.g urn:ogc:def:crs:EPSG::4326)

```
SELECT asMFJSON(tgeompoint 'Point(1 2)@2019-01-01 18:00:00.15+02');
-- "{"type":"MovingPoint","coordinates":[1,2],"datetimes":"2019-01-01T17:00:00.15+01",
"interpolations":["Discrete"]}"
SELECT asMFJSON(tgeompoint 'SRID=4326;
Point(50.813810 4.384260)@2019-01-01 18:00:00.15+02', 2, 3);
-- "{"type":"MovingPoint","crs":{"type":"name","properties":{"name":"EPSG:4326"}},
"stBoundedBy":{"bbox":[50.81,4.38,50.81,4.38],
"period":{"begin":"2019-01-01 17:00:00.15+01","end":"2019-01-01 17:00:00.15+01"}},
"coordinates":[50.81,4.38],"datetimes":"2019-01-01T17:00:00.15+01",
"interpolations":["Discrete"]}"
```

- Get the Well-Known Binary (WKB) representation 

  `asBinary(tpoint): bytea`

  `asBinary(tpoint, text): bytea`

  The result is encoded using either the little-endian (NDR) or the big-endian (XDR) encoding. If no encoding is specified, then the encoding of the machine is used.

```
SELECT asBinary(tgeompoint 'SRID=4326;Point(1 2 3)@2012-01-01');
-- "\001\001\000\000\000\000\000\000\360?\000\000\000\000\000\000\000@ ↩
    \000\000\000\000\000\000\010@\000\374\340\023jX\001\000"
```

- Get the Extended Well-Known Binary (EWKB) representation 

  `asEWKB(tpoint): bytea`

  `asEWKB(tpoint, text): bytea`

  The result is encoded using either the little-endian (NDR) or the big-endian (XDR) encoding. If no encoding is specified, then the encoding of the machine is used.

```
SELECT asEWKB(tgeompoint 'SRID=4326;Point(1 2 3)@2012-01-01');
-- "\0011\346\020\000\000\000\000\000\000\000\360?\000\000\000\000\000\000\000@ ↩
    \000\000\000\000\000\000\010@\000\374\340\023jX\001\000"
```

- Get the Hexadecimal Extended Well-Known Binary (EWKB) representation as text 

  `asHexEWKB(tpoint): text`

  `asHexEWKB(tpoint, text): text`

  The result is encoded using either the little-endian (NDR) or the big-endian (XDR) encoding. If no encoding is specified, then NDR is used.

```
SELECT asHexEWKB(tgeompoint 'SRID=4326;Point(1 2 3)@2012-01-01');
-- "0131E6100000000000000000F03F0000000000000040000000000000084000FCE0136A580100"
```

- Input a temporal point from a Moving Features JSON representation 

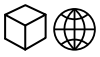
  `fromMFJSON(text): tpoint`

```
SELECT asEWKT(fromMFJSON(text '{"type":"MovingPoint","crs":{"type":"name",
"properties":{"name":"EPSG:4326"}},"coordinates":[50.81,4.38],
"datetimes":"2019-01-01T17:00:00.15+01","interpolations":["Discrete"]}'));
-- "SRID=4326;POINT(50.81 4.38)@2019-01-01 17:00:00.15+01"
```

- Input a temporal point from an Extended Well-Known Binary (EWKB) representation 

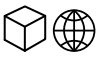
  `fromEWKB(bytea): tpoint`

```
SELECT asEWKT(fromEWKB(bytea
'\0011\346\020\000\000\000\000\000\000\000\360?\000\000\000\000\000\000\000@ ↩
    \000\000\000\000\000\000\010@\000\374\340\023jX\001\000'));
-- "SRID=4326;POINT Z (1 2 3)@2012-01-01"
```

- Get the spatial reference identifier 

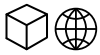
  `SRID(tpoint): integer`

```
SELECT SRID(tgeompoint 'Point(0 0)@2012-01-01');
-- 0
```

- Set the spatial reference identifier 

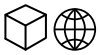
`setSRID(tpoint): tpoint`

```
SELECT asEWKT(setSRID(tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-02)', 4326));
-- "SRID=4326;[POINT(0 0)@2012-01-01 00:00:00+00, POINT(1 1)@2012-01-02 00:00:00+00)"
```

- Transform to a different spatial reference 

`transform(tpoint, integer): tpoint`

```
SELECT asEWKT(transform(tgeompoint 'SRID=4326;Point(4.35 50.85)@2012-01-01', 3812));
-- "SRID=3812;POINT(648679.018035303 671067.055638114)@2012-01-01 00:00:00+00"
```

- Round the coordinate values to a number of decimal places 

`setPrecision(tpoint, int): tpoint`

```
SELECT asText(setPrecision(tgeompoint '{Point(1.12345 1.12345 1.12345)@2000-01-01,
Point(2 2 2)@2000-01-02, Point(1.12345 1.12345 1.12345)@2000-01-03}', 2));
-- "{POINT Z (1.12 1.12 1.12)@2000-01-01, POINT Z (2 2 2)@2000-01-02,
POINT Z (1.12 1.12 1.12)@2000-01-03}"
SELECT asText(setPrecision(tgeogpoint 'Point(1.12345 1.12345)@2000-01-01', 2));
-- "POINT(1.12 1.12)@2000-01-01"
```

- Get the X coordinate values as a temporal float 

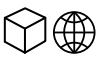
`getX(tpoint): tfloat`

```
SELECT getX(tgeompoint '{Point(1 2)@2000-01-01, Point(3 4)@2000-01-02,
  Point(5 6)@2000-01-03}');
-- "{1@2000-01-01, 3@2000-01-02, 5@2000-01-03}"
SELECT getX(tgeogpoint 'Interp=Stepwise;[Point(1 2 3)@2000-01-01, Point(4 5 6)@2000-01-02,
  Point(7 8 9)@2000-01-03]');
-- "Interp=Stepwise;[1@2000-01-01, 4@2000-01-02, 7@2000-01-03]"
```

- Get the Y coordinate values as a temporal float 

`getY(tpoint): tfloat`

```
SELECT getY(tgeompoint '{Point(1 2)@2000-01-01, Point(3 4)@2000-01-02,
  Point(5 6)@2000-01-03}');
-- "{2@2000-01-01, 4@2000-01-02, 6@2000-01-03}"
SELECT getY(tgeogpoint 'Interp=Stepwise;[Point(1 2 3)@2000-01-01, Point(4 5 6)@2000-01-02,
  Point(7 8 9)@2000-01-03]');
-- "Interp=Stepwise;[2@2000-01-01, 5@2000-01-02, 8@2000-01-03]"
```

- Get the Z coordinate values as a temporal float 

`getZ(tpoint): tfloat`

```
SELECT getZ(tgeompoint '{Point(1 2)@2000-01-01, Point(3 4)@2000-01-02,
  Point(5 6)@2000-01-03}');
-- The temporal point do not have Z dimension
SELECT getZ(tgeogpoint 'Interp=Stepwise;[Point(1 2 3)@2000-01-01, Point(4 5 6)@2000-01-02,
  Point(7 8 9)@2000-01-03]');
-- "Interp=Stepwise;[3@2000-01-01, 6@2000-01-02, 9@2000-01-03]"
```

- Returns true if the temporal point does not spatially self-intersect 

  `isSimple(tpoint): bool`

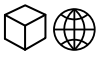
  Notice that a temporal sequence set point is simple if every composing sequence is simple.

```
SELECT isSimple(tgeompoint '[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
Point(0 0)@2000-01-03]');
-- false
SELECT isSimple(tgeompoint '[Point(0 0 0)@2000-01-01, Point(1 1 1)@2000-01-02,
Point(2 0 2)@2000-01-03, Point(0 0 0)@2000-01-04]');
-- true
SELECT isSimple(tgeompoint '{[Point(0 0 0)@2000-01-01, Point(1 1 1)@2000-01-02],
[Point(1 1 1)@2000-01-03, Point(0 0 0)@2000-01-04]}');
-- true
```

- Returns an array of fragments of the temporal point which are simple 

  `makeSimple(tpoint): tgeompoint[]`

```
SELECT asText(makeSimple(tgeompoint '[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
Point(0 0)@2000-01-03]'));
-- {"[POINT(0 0)@2000-01-01, POINT(1 1)@2000-01-02)",
   "[POINT(1 1)@2000-01-02, POINT(0 0)@2000-01-03]"}
SELECT asText(makeSimple(tgeompoint '[Point(0 0 0)@2000-01-01, Point(1 1 1)@2000-01-02,
Point(2 0 2)@2000-01-03, Point(0 0 0)@2000-01-04]'));
-- {"[POINT Z (0 0 0)@2000-01-01, POINT Z (1 1 1)@2000-01-02, POINT Z (2 0 2)@2000-01-03,
   POINT Z (0 0 0)@2000-01-04]"}
SELECT asText(makeSimple(tgeompoint '[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
Point(0 1)@2000-01-03, Point(1 0)@2000-01-04]'));
-- {"[POINT(0 0)@2000-01-01, POINT(1 1)@2000-01-02, POINT(0 1)@2000-01-03)",
   "[POINT(0 1)@2000-01-03, POINT(1 0)@2000-01-04]"}
SELECT asText(makeSimple(tgeompoint '{[Point(0 0 0)@2000-01-01, Point(1 1 1)@2000-01-02],
[Point(1 1 1)@2000-01-03, Point(0 0 0)@2000-01-04]}'));
-- {"{[POINT Z (0 0 0)@2000-01-01, POINT Z (1 1 1)@2000-01-02],
   [POINT Z (1 1 1)@2000-01-03, POINT Z (0 0 0)@2000-01-04]}"}
```

- Get the length traversed by the temporal point 

  `length(tpoint): float`

```
SELECT length(tgeompoint '[Point(0 0 0)@2000-01-01, Point(1 1 1)@2000-01-02]');
-- 1.73205080756888
SELECT length(tgeompoint '[Point(0 0 0)@2000-01-01, Point(1 1 1)@2000-01-02,
Point(0 0 0)@2000-01-03]');
-- 3.46410161513775
SELECT length(tgeompoint 'Interp=Stepwise;[Point(0 0 0)@2000-01-01,
Point(1 1 1)@2000-01-02, Point(0 0 0)@2000-01-03]');
-- 0
```

- Get the cumulative length traversed by the temporal point 

  `cumulativeLength(tpoint): tfloatseq`

```
SELECT round(cumulativeLength(tgeompoint '{[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
  Point(1 0)@2000-01-03], [Point(1 0)@2000-01-04, Point(0 0)@2000-01-05]}'), 6);
-- {[0@2000-01-01, 1.414214@2000-01-02, 2.414214@2000-01-03],
  [2.414214@2000-01-04, 3.414214@2000-01-05]}
SELECT cumulativeLength(tgeompoint 'Interp=Stepwise;[Point(0 0 0)@2000-01-01,
  Point(1 1 1)@2000-01-02, Point(0 0 0)@2000-01-03]');
-- Interp=Stepwise;[0@2000-01-01, 0@2000-01-03]
```

- Get the speed of the temporal point in units per second 

  `speed(tpoint): tfloats`

  The temporal point must have linear interpolation

```
SELECT speed(tgeompoint '{[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
  Point(1 0)@2000-01-03], [Point(1 0)@2000-01-04, Point(0 0)@2000-01-05]}') * 3600 * 24;
-- "Interp=Stepwise;{[1.4142135623731@2000-01-01, 1@2000-01-02, 1@2000-01-03],
   [1@2000-01-04, 1@2000-01-05]}"
SELECT speed(tgeompoint 'Interp=Stepwise;[Point(0 0)@2000-01-01, Point(1 1)@2000-01-02,
  Point(1 0)@2000-01-03]');
-- ERROR:  The temporal value must have linear interpolation
```

- Get the time-weighted centroid 

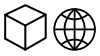
  `twCentroid(tgeompoint): point`

```
SELECT ST_AsText(twCentroid(tgeompoint '{[Point(0 0 0)@2012-01-01,
  Point(0 1 1)@2012-01-02, Point(0 1 1)@2012-01-03, Point(0 0 0)@2012-01-04)}'));
-- "POINT Z (0 0.666666666666667 0.666666666666667)"
```

- Get the temporal azimuth 

  `azimuth(tpoint): tfloat`

```
SELECT degrees(azimuth(tgeompoint '[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-02,
  Point(1 1 1)@2012-01-03, Point(0 0 0)@2012-01-04)'));
-- "Interp=Stepwise;{[45@2012-01-01, 45@2012-01-02], [225@2012-01-03, 225@2012-01-04)}"
```

- Get the instant of the first temporal point at which the two arguments are at the nearest distance 

  `nearestApproachInstant({geo, tpoint}, {geo, tpoint}): tpoint`

  The function will only return the first instant that it finds if there are more than one. The resulting instant may be at an exclusive bound.

```
SELECT asText(NearestApproachInstant(tgeompoint '(Point(1 1)@2000-01-01,
  Point(3 1)@2000-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
-- "POINT(2 1)@2000-01-02"
SELECT asText(NearestApproachInstant(tgeompoint 'Interp=Stepwise;(Point(1 1)@2000-01-01,
  Point(3 1)@2000-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
```

```
-- "POINT(1 1)@2000-01-01"
SELECT asText(NearestApproachInstant(tgeompoint '(Point(1 1)@2000-01-01,
  Point(2 2)@2000-01-03]', tgeompoint '(Point(1 1)@2000-01-01, Point(4 1)@2000-01-03]'));
-- "POINT(1 1)@2000-01-01"
SELECT asText(nearestApproachInstant(tgeompoint '[Point(0 0 0)@2012-01-01,
  Point(1 1 1)@2012-01-03, Point(0 0 0)@2012-01-05)', tgeompoint
  '[Point(2 0 0)@2012-01-02, Point(1 1 1)@2012-01-04, Point(2 2 2)@2012-01-06)'));
-- "POINT Z (0.75 0.75 0.75)@2012-01-03 12:00:00+00"
```

Function `nearestApproachInstant` generalizes the PostGIS function `ST_ClosestPointOfApproach`. First, the latter function requires both arguments to be trajectories. Second, function `nearestApproachInstant` returns both the point and the timestamp of the nearest point of approach while the PostGIS function only provides the timestamp as shown next.

```
SELECT to_timestamp(ST_ClosestPointOfApproach(
tgeompoint '[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-03,
  Point(0 0 0)@2012-01-05)'::geometry,
tgeompoint '[Point(2 0 0)@2012-01-02, Point(1 1 1)@2012-01-04,
  Point(2 2 2)@2012-01-06)'::geometry));
-- "2012-01-03 12:00:00+00"
```

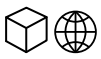
- Get the smallest distance ever 

  `nearestApproachDistance({geo, tpoint}, {geo, tpoint}): float`

```
SELECT NearestApproachDistance(tgeompoint '(Point(1 1)@2000-01-01,
  Point(3 1)@2000-01-03]', geometry 'Linestring(1 3,2 2,3 3)');
-- 1
SELECT NearestApproachDistance(tgeompoint 'Interp=Stepwise;(Point(1 1)@2000-01-01,
  Point(3 1)@2000-01-03]', geometry 'Linestring(1 3,2 2,3 3)');
-- 1.4142135623731
SELECT nearestApproachDistance(
  tgeompoint '[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-03,
  Point(0 0 0)@2012-01-05)',
tgeompoint '[Point(2 0 0)@2012-01-02, Point(1 1 1)@2012-01-04,
  Point(2 2 2)@2012-01-06)');
-- "0.5"
```

Function `nearestApproachDistance` has an associated operator `|=|` that can be used for doing nearest neighbor searches using a GiST index (see Section 5.20). This function corresponds to the function `ST_DistanceCPA` provided by PostGIS, altough the latter requires both arguments to be a trajectory.

```
SELECT ST_DistanceCPA(
tgeompoint '[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-03,
  Point(0 0 0)@2012-01-05)'::geometry,
tgeompoint '[Point(2 0 0)@2012-01-02, Point(1 1 1)@2012-01-04,
  Point(2 2 2)@2012-01-06)'::geometry);
-- "0.5"
```

- Get the line connecting the nearest approach point 

  `shortestLine({geo, tpoint}, {geo, tpoint}): geo`

  The function will only return the first line that it finds if there are more than one.

```
SELECT ST_AsText(shortestLine(tgeompoint '(Point(1 1)@2000-01-01,
  Point(3 1)@2000-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
-- "LINESTRING(2 1,2 2)"
SELECT ST_AsText(shortestLine(tgeompoint 'Interp=Stepwise;(Point(1 1)@2000-01-01,
  Point(3 1)@2000-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
-- "LINESTRING(1 1,2 2)"
SELECT ST_AsText(shortestLine(
  tgeompoint '[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-03,
    Point(0 0 0)@2012-01-05)',
  tgeompoint '[Point(2 0 0)@2012-01-02, Point(1 1 1)@2012-01-04,
    Point(2 2 2)@2012-01-06)'));
-- "LINESTRING Z (0.75 0.75 0.75,1.25 0.75 0.75)"
```

Function `shortestLine` can be used to obtain the result provided by the PostGIS function `ST_CPAWithin` when both arguments are trajectories as shown next.

```
SELECT ST_Length(shortestLine(
  tgeompoint '[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-03,
    Point(0 0 0)@2012-01-05)',
  tgeompoint '[Point(2 0 0)@2012-01-02, Point(1 1 1)@2012-01-04,
    Point(2 2 2)@2012-01-06)')) <= 0.5;
-- true
SELECT ST_CPAWithin(
  tgeompoint '[Point(0 0 0)@2012-01-01, Point(1 1 1)@2012-01-03,
    Point(0 0 0)@2012-01-05)'::geometry,
  tgeompoint '[Point(2 0 0)@2012-01-02, Point(1 1 1)@2012-01-04,
    Point(2 2 2)@2012-01-06)'::geometry, 0.5);
-- true
```

- Simplify a temporal point using a generalization of the Douglas-Peucker algorithm 

  `simplify(tpoint, float):  tpoint`

  `simplify(tpoint, float, float):  tpoint`

  The first version remove points that are less than the distance passed as second argument, which is specified in the units of the coordinate system. The second version remove points that are less than the distance passed as second argument provided that the speed difference between the point and the corresponding point in the simplified version is less than the speed passed as third argument, which is specified in units per second. Notice that simplification applies only to temporal sequences or sequence sets with linear interpolation. In all other cases, a copy of the given temporal point is returned.

```
-- Only distance specified
SELECT ST_AsText(trajectory(simplify(tgeompoint '[Point(0 4)@2000-01-01,
Point(1 1)@2000-01-02, Point(2 3)@2000-01-03, Point(3 1)@2000-01-04,
Point(4 3)@2000-01-05, Point(5 0)@2000-01-06, Point(6 4)@2000-01-07]', 1.5)));
-- "LINESTRING(0 4,1 1,4 3,5 0,6 4)"
SELECT ST_AsText(trajectory(simplify(tgeompoint '[Point(0 4)@2000-01-01,
Point(1 1)@2000-01-02, Point(2 3)@2000-01-03, Point(3 1)@2000-01-04,
Point(4 3)@2000-01-05, Point(5 0)@2000-01-06, Point(6 4)@2000-01-07]', 2)));
-- "LINESTRING(0 4,5 0,6 4)"
SELECT ST_AsText(trajectory(simplify(tgeompoint '[Point(0 4)@2000-01-01,
Point(1 1)@2000-01-02, Point(2 3)@2000-01-03, Point(3 1)@2000-01-04,
Point(4 3)@2000-01-05, Point(5 0)@2000-01-06, Point(6 4)@2000-01-07]', 4)));
-- "LINESTRING(0 4,6 4)"

-- Speed of the temporal point
SELECT round(speed(tgeompoint '[Point(0 4)@2000-01-01, Point(1 1)@2000-01-02,
Point(2 3)@2000-01-03, Point(3 1)@2000-01-04, Point(4 3)@2000-01-05,
Point(5 0)@2000-01-06, Point(6 4)@2000-01-07]') * 1e5, 2);
```

```
-- "Interp=Stepwise;[3.66@2000-01-01, 2.59@2000-01-02, 3.66@2000-01-05,
4.77@2000-01-06, 4.77@2000-01-07]"

-- Both distance and delta speed specified
SELECT ST_AsText(trajectory(simplify(tgeompoint '[Point(0 4)@2000-01-01,
Point(1 1)@2000-01-02, Point(2 3)@2000-01-03, Point(3 1)@2000-01-04,
Point(4 3)@2000-01-05, Point(5 0)@2000-01-06, Point(6 4)@2000-01-07]', 4, 1 / 1e5)));
-- "LINESTRING(0 4,1 1,2 3,3 1,4 3,5 0,6 4)"
SELECT ST_AsText(trajectory(simplify(tgeompoint '[Point(0 4)@2000-01-01,
Point(1 1)@2000-01-02, Point(2 3)@2000-01-03, Point(3 1)@2000-01-04,
Point(4 3)@2000-01-05, Point(5 0)@2000-01-06, Point(6 4)@2000-01-07]', 4, 2 / 1e5)));
-- "LINESTRING(0 4,1 1,5 0,6 4)"
SELECT ST_AsText(trajectory(simplify(tgeompoint '[Point(0 4)@2000-01-01,
Point(1 1)@2000-01-02, Point(2 3)@2000-01-03, Point(3 1)@2000-01-04,
Point(4 3)@2000-01-05, Point(5 0)@2000-01-06, Point(6 4)@2000-01-07]', 4, 3 / 1e5)));
-- "LINESTRING(0 4,6 4)"
```

A typical use for the `simplify` function is to reduce the size of a dataset, in particular for visualization purposes.

- Construct a geometry/geography with M measure from a temporal point and a temporal float 

  `geoMeasure(tpoint, tfloat, segmentize = false):  geo`

  The last `segmentize` argument states whether the resulting value is a either `Linestring M` or a `MultiLinestring M` where each component is a segment of two points.

```
SELECT st_astext(geoMeasure(tgeompoint '{Point(1 1 1)@2000-01-01,
Point(2 2 2)@2000-01-02}', '{5@2000-01-01, 5@2000-01-02}'));
-- "MULTIPOINT ZM (1 1 1 5,2 2 2 5)"
SELECT st_astext(geoMeasure(tgeogpoint '{[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02],
[Point(1 1)@2000-01-03, Point(1 1)@2000-01-04]}',
'{[5@2000-01-01, 5@2000-01-02],[7@2000-01-03, 7@2000-01-04]}'));
-- "GEOMETRYCOLLECTION M (LINESTRING M (1 1 5,2 2 5),POINT M (1 1 7))"
SELECT st_astext(geoMeasure(tgeompoint '[Point(1 1)@2000-01-01,
Point(2 2)@2000-01-02, Point(1 1)@2000-01-03]', '[5@2000-01-01, 7@2000-01-02, 5@2000 ↩
    -01-03]', true));
-- "MULTILINESTRING M ((1 1 5,2 2 5),(2 2 7,1 1 7))"
```

A typical visualization for mobility data is to show on a map the trajectory of the moving object using different colors according to the speed. Figure 5.1 shows the result of the query below using a color ramp in QGIS.

```
WITH Temp(t) AS (
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-05,
  Point(2 0)@2012-01-08, Point(3 1)@2012-01-10, Point(4 0)@2012-01-11]'
)
SELECT ST_AsText(geoMeasure(t, round(speed(t) * 3600 * 24, 2), true))
FROM Temp;
-- "MULTILINESTRING M ((0 0 0.35,1 1 0.35),(1 1 0.47,2 0 0.47),(2 0 0.71,3 1 0.71),
(3 1 1.41,4 0 1.41))"
```

The following expression is used in QGIS to achieve this. The `scale_linear` function transforms the M value of each composing segment to the range [0, 1]. This value is then passed to the `ramp_color` function.

```
ramp_color(
'RdYlBu',
scale_linear(
  m(start_point(geometry_n($geometry,@geometry_part_num))),
  0, 2, 0, 1)
)
```

Figure 5.1: Visualizing the speed of a moving object using a color ramp in QGIS.

5.7 Restriction Functions

These functions restrict the temporal value with respect to a value or a time extent.

- Restrict to a value

  `atValue(ttype, base): ttype`

```
SELECT atValue(tint '[1@2012-01-01, 1@2012-01-15)', 1);
-- "[1@2012-01-01, 1@2012-01-15)"
SELECT asText(atValue(tgeompoint '[Point(0 0 0)@2012-01-01, Point(2 2 2)@2012-01-03)',
'Point(1 1 1)'));
-- "{[POINT Z (1 1 1)@2012-01-02]}"
```

- Restrict to an array of values

  `atValues(ttype, base[]): ttype`

```
SELECT atValues(tfloat '[1@2012-01-01, 4@2012-01-4)', ARRAY[1, 3, 5]);
-- "{[1@2012-01-01], [3@2012-01-03]}"
SELECT asText(atValues(tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-03)',
ARRAY[geometry 'Point(0 0)', 'Point(1 1)']));
-- "{[POINT(0 0)@2012-01-01 00:00:00+00], [POINT(1 1)@2012-01-02 00:00:00+00]}"
```

- Restrict to a range

  `atRange(tnumber, numrange): ttype`

```
SELECT atRange(tfloat '[1@2012-01-01, 4@2012-01-4)', floatrange '[1,3]');
-- "[1@2012-01-01, 3@2012-01-03]"
```

- Restrict to an array of ranges

  `atRanges(tnumber, numrange[]): ttype`

```
SELECT atRanges(tfloat '[1@2012-01-01, 5@2012-01-05)',
ARRAY[floatrange '[1,2]', '[3,4]']);
-- "{[1@2012-01-01, 2@2012-01-02],[3@2012-01-03, 4@2012-01-04]}"
```

- Restrict to the minimum value

  `atMin(torder): torder`

  The function returns null if the minimum value only happens at exclusive bounds.

```
SELECT atMin(tint '{1@2012-01-01, 2@2012-01-03, 1@2012-01-05}');
-- "{1@2012-01-01, 1@2012-01-05}"
SELECT atMin(tint '(1@2012-01-01, 3@2012-01-03]');
-- "{(1@2012-01-01, 1@2012-01-03)}"
SELECT atMin(tfloat '(1@2012-01-01, 3@2012-01-03]');
-- NULL
SELECT atMin(ttext '{(AA@2012-01-01, AA@2012-01-03), (BB@2012-01-03, AA@2012-01-05]}');
-- "{(AA@2012-01-01, AA@2012-01-03), [AA@2012-01-05]}"
```

- Restrict to the maximum value

  `atMax(torder):  torder`

  The function returns null if the maximum value only happens at exclusive bounds.

```
SELECT atMax(tint '{1@2012-01-01, 2@2012-01-03, 3@2012-01-05}');
-- "{3@2012-01-05}"
SELECT atMax(tfloat '(1@2012-01-01, 3@2012-01-03)');
-- NULL
SELECT atMax(tfloat '{(2@2012-01-01, 1@2012-01-03), [2@2012-01-03, 2@2012-01-05)}');
-- "{[2@2012-01-03, 2@2012-01-05]}"
SELECT atMax(ttext '{(AA@2012-01-01, AA@2012-01-03), (BB@2012-01-03, AA@2012-01-05]}');
-- "{("BB"@2012-01-03, "BB"@2012-01-05)}"
```

- Restrict to a geometry

  `atGeometry(tgeompoint, geometry):  tgeompoint`

  Notice that it is allowed to mix 2D/3D geometries but the computation is only performed on 2D.

```
SELECT asText(atGeometry(tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)',
  geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
-- "{"[POINT(1 1)@2012-01-02, POINT(2 2)@2012-01-03]"}"
SELECT astext(atGeometry(tgeompoint '[Point(0 0 0)@2000-01-01, Point(4 4 4)@2000-01-05]',
  geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
-- "{[POINT Z (1 1 1)@2000-01-02, POINT Z (2 2 2)@2000-01-03]}"
```

- Restrict to a timestamp

  `atTimestamp(ttype, timestamptz):  ttypeinst`

```
SELECT atTimestamp(tfloat '[1@2012-01-01, 5@2012-01-05)', '2012-01-02');
-- "2@2012-01-02"
```

- Restrict to a timestamp set

  `atTimestampSet(ttype, timestampset):  {ttypeinst, ttypei}`

```
SELECT atTimestampSet(tint '[1@2012-01-01, 1@2012-01-15)',
timestampset '{2012-01-01, 2012-01-03}');
-- "{1@2012-01-01, 1@2012-01-03}"
```

- Restrict to a period

  `atPeriod(ttype, period):  ttype`

```
SELECT atPeriod(tfloat '{[1@2012-01-01, 3@2012-01-03), [3@2012-01-04, 1@2012-01-06)}',
'[2012-01-02,2012-01-05)');
-- "{[2@2012-01-02, 3@2012-01-03), [3@2012-01-04, 2@2012-01-05)}"
```

- Restrict to a period set

  `atPeriodSet(ttype, periodset): ttype`

```
SELECT atPeriodSet(tint '[1@2012-01-01, 1@2012-01-15)',
periodset '{[2012-01-01, 2012-01-03), [2012-01-04, 2012-01-05)}');
-- "{[1@2012-01-01, 1@2012-01-03),[1@2012-01-04, 1@2012-01-05)}"
```

- Restrict to a `tbox`

  `atTbox(tnumber, tbox): tnumber`

```
SELECT atTbox(tfloat '[0@2012-01-01, 3@2012-01-04)',
tbox 'TBOX((0, 2012-01-02), (2, 2012-01-04))');
-- "{[1@2012-01-02, 2@2012-01-03]}"
```

- Restrict to an `stbox`

  `atStbox(tgeompoint, stbox): tgeompoint`

```
SELECT asText(atStbox(tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)',
stbox 'STBOX T((0, 0, 2012-01-02), (2, 2, 2012-01-04))'));
-- "{[POINT(1 1)@2012-01-02, POINT(2 2)@2012-01-03]}"
```

5.8 Difference Functions

These functions restrict the temporal value with respect to the complement of a value or a time extent.

- Difference with a value

  `minusValue(ttype, base): ttype`

```
SELECT minusValue(tint '[1@2012-01-01, 2@2012-01-02, 2@2012-01-03)', 1);
-- "{[2@2012-01-02, 2@2012-01-03)}"
SELECT asText(minusValue(tgeompoint '[Point(0 0 0)@2012-01-01, Point(2 2 2)@2012-01-03)',
'Point(1 1 1)'));
-- "{[POINT Z (0 0 0)@2012-01-01, POINT Z (1 1 1)@2012-01-02),
(POINT Z (1 1 1)@2012-01-02, POINT Z (2 2 2)@2012-01-03)}"
```

- Difference with an array of values

  `minusValues(ttype, base[]): ttype`

```
SELECT minusValues(tfloat '[1@2012-01-01, 4@2012-01-4)', ARRAY[2, 3]);
-- "{[1@2012-01-01, 2@2012-01-02), (2@2012-01-02, 3@2012-01-03),
(3@2012-01-03, 4@2012-01-04)}"
SELECT asText(minusValues(tgeompoint '[Point(0 0 0)@2012-01-01, Point(3 3 3)@2012-01-04)',
ARRAY[geometry 'Point(1 1 1)', 'Point(2 2 2)']));
-- "{[POINT Z (0 0 0)@2012-01-01, POINT Z (1 1 1)@2012-01-02),
(POINT Z (1 1 1)@2012-01-02, POINT Z (2 2 2)@2012-01-03),
(POINT Z (2 2 2)@2012-01-03, POINT Z (3 3 3)@2012-01-04)}"
```

- Difference with a range

  `minusRange(tnumber, numrange): ttype`

```
SELECT minusRange(tfloat '[1@2012-01-01, 4@2012-01-4)', floatrange '[2,3]');
-- "{[1@2012-01-01, 2@2012-01-02), (3@2012-01-03, 4@2012-01-04)}"
```

- Difference with an array of ranges

  `minusRanges(tnumber, numrange[]): ttype`

```
SELECT minusRanges(tfloat '[1@2012-01-01, 5@2012-01-05)',
ARRAY[floatrange '[1,2]', '[3,4]']);
-- "{(2@2012-01-02, 3@2012-01-03), (4@2012-01-04, 5@2012-01-05)}"
```

- Difference with the minimum value

  `minusMin(torder): torder`

```
SELECT minusMin(tint '{1@2012-01-01, 2@2012-01-03, 1@2012-01-05}');
-- "{2@2012-01-03}"
SELECT minusMin(tfloat '[1@2012-01-01, 3@2012-01-03]');
-- "{(1@2012-01-01, 3@2012-01-03]}"
SELECT minusMin(tfloat '(1@2012-01-01, 3@2012-01-03)');
-- "{(1@2012-01-01, 3@2012-01-03)}"
SELECT minusMin(tint '{[1@2012-01-01, 1@2012-01-03), (1@2012-01-03, 1@2012-01-05)}');
-- NULL
```

- Difference with the maximum value

  `minusMax(torder): torder`

```
SELECT minusMax(tint '{1@2012-01-01, 2@2012-01-03, 3@2012-01-05}');
-- "{1@2012-01-01, 2@2012-01-03}"
SELECT minusMax(tfloat '[1@2012-01-01, 3@2012-01-03]');
-- "{[1@2012-01-01, 3@2012-01-03)}"
SELECT minusMax(tfloat '(1@2012-01-01, 3@2012-01-03)');
-- "{(1@2012-01-01, 3@2012-01-03)}"
SELECT minusMax(tfloat '{[2@2012-01-01, 1@2012-01-03), [2@2012-01-03, 2@2012-01-05)}');
-- "{(2@2012-01-01, 1@2012-01-03)}"
SELECT minusMax(tfloat '{[1@2012-01-01, 3@2012-01-03), (3@2012-01-03, 1@2012-01-05)}');
-- "{[1@2012-01-01, 3@2012-01-03), (3@2012-01-03, 1@2012-01-05)}"
```

- Difference with a geometry

  `minusGeometry(tgeompoint, geometry): tgeompoint`

  Notice that it is allowed to mix 2D/3D geometries but the computation is only performed on 2D.

```
SELECT asText(minusGeometry(tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)',
  geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
-- "{[POINT(0 0)@2012-01-01, POINT(1 1)@2012-01-02), (POINT(2 2)@2012-01-03,
  POINT(3 3)@2012-01-04)}"
SELECT astext(minusGeometry(tgeompoint '[Point(0 0 0)@2000-01-01,
  Point(4 4 4)@2000-01-05]', geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
-- "{[POINT Z (0 0 0)@2000-01-01, POINT Z (1 1 1)@2000-01-02),
  (POINT Z (2 2 2)@2000-01-03, POINT Z (4 4 4)@2000-01-05]}"
```

- Difference with a timestamp

  `minusTimestamp(ttype, timestamptz): ttype`

```
SELECT minusTimestamp(tfloat '[1@2012-01-01, 5@2012-01-05)', '2012-01-02');
-- "{[1@2012-01-01, 2@2012-01-02), (2@2012-01-02, 5@2012-01-05)}"
```

- Difference with a timestamp set

  `minusTimestampSet(ttype, timestampset): ttype`

```
SELECT minusTimestampSet(tint '[1@2012-01-01, 1@2012-01-15)',
timestampset '{2012-01-02, 2012-01-03}');
-- "{[1@2012-01-01, 1@2012-01-02), (1@2012-01-02, 1@2012-01-03),
(1@2012-01-03, 1@2012-01-15)}"
```

- Difference with a period

  `minusPeriod(ttype, period): ttype`

```
SELECT minusPeriod(tfloat '{[1@2012-01-01, 3@2012-01-03), [3@2012-01-04, 1@2012-01-06)}',
'[2012-01-02,2012-01-05)');
-- "{[1@2012-01-01, 2@2012-01-02), [2@2012-01-05, 1@2012-01-06)}"
```

- Difference with a period set

  `minusPeriodSet(ttype, periodset): ttype`

```
SELECT minusPeriodSet(tint '[1@2012-01-01, 1@2012-01-15)',
periodset '{[2012-01-02, 2012-01-03), [2012-01-04, 2012-01-05)}');
-- "{[1@2012-01-01, 1@2012-01-02), [1@2012-01-03, 1@2012-01-04),
[1@2012-01-05, 1@2012-01-15)}"
```

- Difference with a `tbox`

  `minusTbox(tnumber, tbox): tnumber`

```
SELECT minusTbox(tfloat '[0@2012-01-01, 3@2012-01-04)',
tbox 'TBOX((0, 2012-01-02), (2, 2012-01-04))');
-- "{[0@2012-01-01, 1@2012-01-02), (2@2012-01-03, 3@2012-01-04)}"
```

- Difference with an `stbox`

  `minusStbox(tgeompoint, stbox): tgeompoint`

```
SELECT asText(minusStbox(tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)',
stbox 'STBOX T((0, 0, 2012-01-02), (2, 2, 2012-01-04))'));
-- "{[POINT(0 0)@2012-01-01, POINT(1 1)@2012-01-02),
(POINT(2 2)@2012-01-03, POINT(3 3)@2012-01-04)}"
```

5.9 Comparison Operators

The traditional comparison operators (=, <, and so on) require that the left and right operands be of the same base type. Excepted equality and inequality, the other comparison operators are not useful in the real world but allow B-tree indexes to be constructed on temporal types. These operators compare the bounding periods (see Section 2.1.4), then the bounding boxes (see Section 4.7) and if those are equal, then the comparison depends on the subtype. For instant values, they compare first the timestamps and if those are equal, compare the values. For instant set and sequence values, they compare the first N instants, where N is the minimum of the number of composing instants of both values. Finally, for sequence set values, they compare the first N sequence values, where N is the minimum of the number of composing sequences of both values.

The equality and inequality operators consider the equivalent representation for different subtypes as shown next.

```
SELECT tint '1@2001-01-01' = tint '{1@2001-01-01}';
-- true
SELECT tfloat '1.5@2001-01-01' = tfloat '[1.5@2001-01-01]';
-- true
SELECT ttext 'AAA@2001-01-01' = ttext '{[AAA@2001-01-01]}';
-- true
SELECT tgeompoint '{Point(1 1)@2001-01-01, Point(2 2)@2001-01-02}' =
tgeompoint '{[Point(1 1)@2001-01-01], [Point(2 2)@2001-01-02]}';
-- true
SELECT tgeogpoint '[Point(1 1 1)@2001-01-01, Point(2 2 2)@2001-01-02]' =
tgeogpoint '{[Point(1 1 1)@2001-01-01], [Point(2 2 2)@2001-01-02]}';
-- true
```

- Are the temporal values equal?

  `ttype = ttype: boolean`

```
SELECT tint '[1@2012-01-01, 1@2012-01-04)' = tint '[2@2012-01-03, 2@2012-01-05)';
-- false
```

- Are the temporal values different?

  `ttype <> ttype: boolean`

```
SELECT tint '[1@2012-01-01, 1@2012-01-04)' <> tint '[2@2012-01-03, 2@2012-01-05)'
-- true
```

- Is the first temporal value less than the second one?

  `ttype < ttype: boolean`

```
SELECT tint '[1@2012-01-01, 1@2012-01-04)' < tint '[2@2012-01-03, 2@2012-01-05)'
-- true
```

- Is the first temporal value greater than the second one?

  `ttype > ttype: boolean`

```
SELECT tint '[1@2012-01-01, 1@2012-01-04)' > tint '[2@2012-01-03, 2@2012-01-05)'
-- false
```

- Is the first temporal value less than or equal to the second one?

  `ttype <= ttype: boolean`

```
SELECT tint '[1@2012-01-01, 1@2012-01-04)' <= tint '[2@2012-01-03, 2@2012-01-05)'
-- true
```

- Is the first temporal value greater than or equal to the second one?

  `ttype >= ttype: boolean`

```
SELECT tint '[1@2012-01-01, 1@2012-01-04)' >= tint '[2@2012-01-03, 2@2012-01-05)'
-- false
```

5.10 Ever and Always Comparison Operators

A possible generalization of the traditional comparison operators (=, <>, <, <=, etc.) to temporal types consists in determining whether the comparison is ever or always true. In this case, the result is a Boolean value. MobilityDB provides operators to test whether the comparison of a temporal value and a value of the base type is ever or always true. These operators are denoted by prefixing the traditional comparison operators with, respectively, `?` (ever) and `%` (always). Some examples are `?=`, `%<>`, or `?<=`. Ever/always equality and non-equality are available for all temporal types, while ever/always inequalities are only available for temporal types whose base type has a total order defined, that is, `tint`, `tfloat`, or `ttext`. The ever and always comparisons are inverse operators: for example, `?=` is the inverse of `%<>`, and `?>` is the inverse of `%<=`.

- Is the temporal value ever equal to the value?

  `ttype ?= base: bool`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT tfloat '[1@2012-01-01, 3@2012-01-04)' ?= 2;
-- true
SELECT tfloat '[1@2012-01-01, 3@2012-01-04)' ?= 3;
-- true
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-04)' ?=
geometry 'Point(1 1)';
-- true
```

- Is the temporal value ever different from the value?

  `ttype ?<> base: bool`

```
SELECT tfloat '[1@2012-01-01, 3@2012-01-04)' ?<> 2;
-- false
SELECT tfloat '[2@2012-01-01, 2@2012-01-04)' ?<> 2;
-- true
SELECT tgeompoint '[Point(1 1)@2012-01-01, Point(1 1)@2012-01-04)' ?<>
geometry 'Point(1 1)';
-- true
```

- Is the temporal value ever less than the value?

  `tnumber ?< number: bool`

```
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' ?< 2;
-- "{[t@2012-01-01, f@2012-01-02, f@2012-01-04)}"
SELECT tint '[2@2012-01-01, 2@2012-01-05)' ?< tfloat '[1@2012-01-03, 3@2012-01-05)';
-- "{[f@2012-01-03, f@2012-01-04], (t@2012-01-04, t@2012-01-05)}"
```

- Is the temporal value ever greater than the value?

  `tnumber ?> number:  bool`

```
SELECT tint '[1@2012-01-03, 1@2012-01-05)' ?> 1;
-- "[f@2012-01-03, f@2012-01-05)"
```

- Is the temporal value ever less than or equal to the value?

  `tnumber ?<= number:  bool`

```
SELECT tint '[1@2012-01-01, 1@2012-01-05)' ?<= tfloat '{2@2012-01-03, 3@2012-01-04}';
-- "{t@2012-01-03, t@2012-01-04}"
```

- Is the temporal value ever greater than or equal to the value?

  `tnumber ?>= number:  bool`

```
SELECT 'AAA'::text ?> ttext '{[AAA@2012-01-01, AAA@2012-01-03),
[BBB@2012-01-04, BBB@2012-01-05)}';
-- "{[f@2012-01-01, f@2012-01-03), [t@2012-01-04, t@2012-01-05)}"
```

- Is the temporal value always equal to the value?

  `ttype %= base:  bool`

  The function does not take into account whether the bounds are inclusive or not.

```
SELECT tfloat '[1@2012-01-01, 3@2012-01-04)' %= 2;
-- true
SELECT tfloat '[1@2012-01-01, 3@2012-01-04)' %= 3;
-- true
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-04)' %=
geometry 'Point(1 1)';
-- true
```

- Is the temporal value always different to the value?

  `ttype %<> base:  bool`

```
SELECT tfloat '[1@2012-01-01, 3@2012-01-04)' %<> 2;
-- false
SELECT tfloat '[2@2012-01-01, 2@2012-01-04)' %<> 2;
-- true
SELECT tgeompoint '[Point(1 1)@2012-01-01, Point(1 1)@2012-01-04)' %<>
geometry 'Point(1 1)';
-- true
```

- Is the temporal value always less than the value?

  `tnumber %< number:  bool`

```
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' %< 2;
-- "{[t@2012-01-01, f@2012-01-02, f@2012-01-04)}"
SELECT tint '[2@2012-01-01, 2@2012-01-05)' %< tfloat '[1@2012-01-03, 3@2012-01-05)';
-- "{[f@2012-01-03, f@2012-01-04], (t@2012-01-04, t@2012-01-05)}"
```

- Is the temporal value always greater than the value?

```
tnumber %> number:  bool
```

```
SELECT tint '[1@2012-01-03, 1@2012-01-05)' %> 1;
-- "[f@2012-01-03, f@2012-01-05)"
```

- Is the temporal value always less than or equal to the value?

```
tnumber %<= number:  bool
```

```
SELECT tint '[1@2012-01-01, 1@2012-01-05)' %<= tfloat '{2@2012-01-03, 3@2012-01-04}';
-- "{t@2012-01-03, t@2012-01-04}"
```

- Is the temporal value always greater than or equal to the value?

```
tnumber %>= number:  bool
```

```
SELECT 'AAA'::text %> ttext '{[AAA@2012-01-01, AAA@2012-01-03),
[BBB@2012-01-04, BBB@2012-01-05)}';
-- "{[f@2012-01-01, f@2012-01-03), [t@2012-01-04, t@2012-01-05)}"
```

5.11 Temporal Comparison Operators

Another possible generalization of the traditional comparison operators (=, <>, <, <=, etc.) to temporal types consists in determining whether the comparison is true or false at each instant. In this case, the result is a temporal Boolean. The temporal comparison operators are denoted by prefixing the traditional comparison operators with #. Some examples are #= or #<=. Temporal equality and non-equality are available for all temporal types, while temporal inequalities are only available for temporal types whose base type has a total order defined, that is, `tint`, `tfloat`, or `ttext`.

- Temporal equal

```
{base, ttype} #= {base, ttype}:  tbool
```

```
SELECT tfloat '[1@2012-01-01, 2@2012-01-04)' #= 3;
-- "{[f@2012-01-01, f@2012-01-04)}"
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' #= tint '[1@2012-01-01, 1@2012-01-04)';
-- "{[t@2012-01-01], (f@2012-01-01, f@2012-01-04)}"
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' #= tfloat '[4@2012-01-02, 1@2012-01-05)';
-- "{[f@2012-01-02, t@2012-01-03], (f@2012-01-03, f@2012-01-04)}"
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-03)' #=
geometry 'Point(1 1)';
-- "{[f@2012-01-01, t@2012-01-02], (f@2012-01-02, f@2012-01-03)}"
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-03)' #=
tgeompoint '[Point(0 2)@2012-01-01, Point(2 0)@2012-01-03)';
-- "{[f@2012-01-01], (t@2012-01-01, t@2012-01-03)}"
```

- Temporal different

```
{base, ttype} #<> {base, ttype}:  tbool
```

```
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' #<> 2;
-- "{[t@2012-01-01, f@2012-01-02], (t@2012-01-02, 2012-01-04)}"
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' #<> tint '[2@2012-01-02, 2@2012-01-05)';
-- "{[f@2012-01-02], (t@2012-01-02, t@2012-01-04)}"
```

- Temporal less than

```
{base, torder} #< {base, torder}:  tbool
```

```
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' #< 2;
-- "{[t@2012-01-01, f@2012-01-02, f@2012-01-04)}"
SELECT tint '[2@2012-01-01, 2@2012-01-05)' #< tfloat '[1@2012-01-03, 3@2012-01-05)';
-- "{[f@2012-01-03, f@2012-01-04], (t@2012-01-04, t@2012-01-05)}"
```

- Temporal greater than

```
{base, torder} #> {base, torder}:  tbool
```

```
SELECT 1 #> tint '[1@2012-01-03, 1@2012-01-05)';
-- "[f@2012-01-03, f@2012-01-05)"
```

- Temporal less than or equal to

```
{base, torder} #<= {base, torder}:  tbool
```

```
SELECT tint '[1@2012-01-01, 1@2012-01-05)' #<= tfloat '{2@2012-01-03, 3@2012-01-04}';
-- "{t@2012-01-03, t@2012-01-04}"
```

- Temporal greater than or equal to

```
{base, torder} #>= {base, torder}:  tbool
```

```
SELECT 'AAA'::text #> ttext '{[AAA@2012-01-01, AAA@2012-01-03),
[BBB@2012-01-04, BBB@2012-01-05)}';
-- "{[f@2012-01-01, f@2012-01-03), [t@2012-01-04, t@2012-01-05)}"
```

5.12 Mathematical Functions and Operators

- Temporal addition

```
{number, tnumber} + {number, tnumber}:  tnumber
```

```
SELECT tint '[2@2012-01-01, 2@2012-01-04)' + 1.5;
-- "[3.5@2012-01-01, 3.5@2012-01-04)"
SELECT tint '[2@2012-01-01, 2@2012-01-04)' + tfloat '[1@2012-01-01, 4@2012-01-04)';
-- "[3@2012-01-01, 6@2012-01-04)"
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' +
tfloat '{[1@2012-01-01, 2@2012-01-02), [1@2012-01-02, 2@2012-01-04)}';
-- "{[2@2012-01-01, 4@2012-01-04), [3@2012-01-02, 6@2012-01-04)}"
```

- Temporal subtraction

  `{number, tnumber} - {number, tnumber}: tnumber`

```
SELECT tint '[1@2012-01-01, 1@2012-01-04)' - tint '[2@2012-01-03, 2@2012-01-05)';
-- "[-1@2012-01-03, -1@2012-01-04)"
SELECT tfloat '[3@2012-01-01, 6@2012-01-04)' - tint '[2@2012-01-01, 2@2012-01-04)';
-- "[1@2012-01-01, 4@2012-01-04)"
```

- Temporal multiplication

  `{number, tnumber} * {number, tnumber}: tnumber`

```
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' * 2;
-- "[2@2012-01-01, 8@2012-01-04)"
SELECT tfloat '[1@2012-01-01, 4@2012-01-04)' * tint '[2@2012-01-01, 2@2012-01-04)';
-- "[2@2012-01-01, 8@2012-01-04)"
SELECT tfloat '[1@2012-01-01, 3@2012-01-03)' * '[3@2012-01-01, 1@2012-01-03)'
-- "{[3@2012-01-01, 4@2012-01-02, 3@2012-01-03)}"
```

- Temporal division

  `{number, tnumber} / {number, tnumber}: tnumber`

  The function will raise an error if the denominator will ever be equal to zero during the common timespan of the arguments.

```
SELECT 2 / tfloat '[1@2012-01-01, 3@2012-01-04)';
-- "[2@2012-01-01, 1@2012-01-02 12:00:00+00, 0.66666666666667@2012-01-04)"
SELECT tfloat '[1@2012-01-01, 5@2012-01-05)' / '[5@2012-01-01, 1@2012-01-05)'
-- "{[0.2@2012-01-01, 1@2012-01-03,2012-01-03, 5@2012-01-03,2012-01-05)}"
select 2 / tfloat '[-1@2000-01-01, 1@2000-01-02]'
-- ERROR:  Division by zero
select tfloat '[-1@2000-01-04, 1@2000-01-05]' / tfloat '[-1@2000-01-01, 1@2000-01-05]'
-- "[-2@2000-01-04, 1@2000-01-05]"
```

- Round the values to a number of decimal places

  `round(tfloat, integer): tfloat`

```
SELECT round(tfloat '[0.7853981633974448@2000-01-01, 2.356194490192346@2000-01-02]', 2);
-- "[0.79@2000-01-01, 2.36@2000-01-02]"
```

- Convert from radians to degrees

  `degrees(tfloat): tfloat`

```
SELECT degrees(tfloat '[0.7853981633974448@2000-01-01, 2.356194490192346@2000-01-02]');
-- "[45@2000-01-01, 135@2000-01-02]"
```

- Get the derivative over time of the temporal float in units per second

  `derivative(tfloat): tfloat`

  The temporal float must have linear interpolation

```
SELECT derivative(tfloat '{[0@2000-01-01, 10@2000-01-02, 5@2000-01-03],
  [1@2000-01-04, 0@2000-01-05]}') * 3600 * 24;
--  Interp=Stepwise;{[-10@2000-01-01, 5@2000-01-02, 5@2000-01-03],
  [1@2000-01-04, 1@2000-01-05]}
SELECT derivative(tfloat 'Interp=Stepwise;[0@2000-01-01, 10@2000-01-02, 5@2000-01-03]');
-- ERROR:  The temporal value must have linear interpolation
```

5.13 Boolean Operators

- Temporal and

  `{bool, tbool} & {bool, tbool}:  tbool`

```
SELECT tbool '[true@2012-01-03, true@2012-01-05)' &
tbool '[false@2012-01-03, false@2012-01-05)';
-- "[f@2012-01-03, f@2012-01-05)"
SELECT tbool '[true@2012-01-03, true@2012-01-05)' &
tbool '{[false@2012-01-03, false@2012-01-04),
[true@2012-01-04, true@2012-01-05)}';
-- "{[f@2012-01-03, t@2012-01-04, t@2012-01-05)}"
```

- Temporal or

  `{bool, tbool} | {bool, tbool}:  tbool`

```
SELECT tbool '[true@2012-01-03, true@2012-01-05)' |
tbool '[false@2012-01-03, false@2012-01-05)';
-- "[t@2012-01-03, t@2012-01-05)"
```

- Temporal not

  `~ tbool:  tbool`

```
SELECT ~ tbool '[true@2012-01-03, true@2012-01-05)';
-- "[f@2012-01-03, f@2012-01-05)"
```

5.14 Text Functions and Operators

- Temporal text concatenation

  `{text, ttext} || {text, ttext}:  ttext`

```
SELECT ttext '[AA@2012-01-01, AA@2012-01-04)' || text 'B';
-- "["AAB"@2012-01-01, "AAB"@2012-01-04)"
SELECT ttext '[AA@2012-01-01, AA@2012-01-04)' || ttext '[BB@2012-01-02, BB@2012-01-05)';
-- "["AABB"@2012-01-02, "AABB"@2012-01-04)"
SELECT ttext '[A@2012-01-01, B@2012-01-03, C@2012-01-04]' ||
ttext '{[D@2012-01-01, D@2012-01-02), [E@2012-01-02, E@2012-01-04)}';
-- "{["DA"@2012-01-01, "EA"@2012-01-02, "EB"@2012-01-03, "EB"@2012-01-04)}"
```

- Transform to uppercase

  `upper(ttext): ttext`

```
SELECT upper(ttext '[AA@2000-01-01, bb@2000-01-02]');
-- "["AA"@2000-01-01, "BB"@2000-01-02]"
```

- Transform to lowercase

  `lower(ttext): ttext`

```
SELECT lower(ttext '[AA@2000-01-01, bb@2000-01-02]');
-- "["aa"@2000-01-01, "bb"@2000-01-02]"
```

5.15 Bounding Box Operators

These operators test whether the bounding boxes of their arguments satisfy the predicate and result in a Boolean value. As stated in Chapter 3, the bounding box associated to a temporal type depends on the base type: It is the `period` type for the `tbool` and `ttext` types, the `tbox` type for the `tint` and `tfloat` types, and the `stbox` type for the `tgeompoint` and `tgeogpoint` types. Furthermore, as seen in Section 4.3, many PostgreSQL, PostGIS, or MobilityDB types can be cast to the `tbox` and `stbox` types. For example, numeric and range types can be casted to type `tbox`, types `geometry` and `geography` can be casted to type `stbox`, and time types and temporal types can be casted to types `tbox` and `stbox`.

A first set of operators consider the topological relationships between the bounding boxes. There are five topological operators: overlaps (`&&`), contains (`@>`), contained (`<@`), same (`~=`), and adjacent (`-|-`). The arguments of these operators can be a base type, a box, or a temporal type and the operators verify the topological relationship taking into account the value and/or the time dimension depending on the type of the arguments.

Another set of operators consider the relative position of the bounding boxes. The operators `<<`, `>>`, `&<`, and `&>` consider the value dimension for `tint` and `tfloat` types and the X coordinates for the `tgeompoint` and `tgeogpoint` types, the operators `<<|`, `|>>`, `&<|`, and `|&>` consider the Y coordinates for the `tgeompoint` and `tgeogpoint` types, the operators `<</`, `/>>`, `&</`, and `/&>` consider the Z coordinates for the `tgeompoint` and `tgeogpoint` types, and the operators `<<#`, `#>>`, `#&<`, and `#&>` consider the time dimension for all temporal types.

Finally, it is worth noting that the bounding box operators allow to mix 2D/3D geometries but in that case, the computation is only performed on 2D.

We refer to Section 4.9 and Section 4.10 for the bounding box operators.

5.16 Distance Operators

There are two distance operators. The first one, denoted `|=|`, computes the distance between either a temporal point and a geometry or between two temporal points at their nearest point of approach, which is a float. This is the same as the function `nearestApproachDistance` discussed before but as an operator it can be used for doing nearest neightbor searches using a GiST index (see Section 5.20).

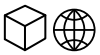
On the other hand, the temporal distance operator, denoted `<->`, computes the distance at each instant of the intersection of the temporal extents of their arguments and results in a temporal float. Computing temporal distance is useful in many mobility applications. For example, a moving cluster (also known as convoy or flock) is defined as a set of objects that move close to each other for a long time interval. This requires to compute temporal distance between two moving objects.

The temporal distance operator accepts a geometry/geography restricted to a point or a temporal point as arguments. Notice that the temporal types only consider linear interpolation between values, while the distance is a root of a quadratic function. Therefore, the temporal distance operator gives a linear approximation of the actual distance value for temporal sequence points. In this case, the arguments are synchronized in the time dimension, and for each of the composing line segments of the arguments, the spatial distance between the start point, the end point, and the nearest point of approach is computed, as shown in the examples below.

- Get the smallest distance ever 

```
{geometry, tgeompoint} |=| {geometry, tgeompoint}:  float
```

```
SELECT tgeompoint '[Point(0 0)@2012-01-02, Point(1 1)@2012-01-04, Point(0 0)@2012-01-06)'
|=| geometry 'Linestring(2 2,2 1,3 1)';
-- "1"
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03, Point(0 0)@2012-01-05)'
|=| tgeompoint '[Point(2 0)@2012-01-02, Point(1 1)@2012-01-04, Point(2 2)@2012-01-06)';
-- "0.5"
```

- Get the temporal distance 

```
{point, tpoint} <-> {point, tpoint}:  tfloat
```

```
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)' <->
geometry 'Point(0 1)';
-- "[1@2012-01-01, 0.707106781186548@2012-01-02, 1@2012-01-03)"
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)' <->
tgeompoint '[Point(0 1)@2012-01-01, Point(1 0)@2012-01-03)';
-- "[1@2012-01-01, 0@2012-01-02, 1@2012-01-03)"
SELECT tgeompoint '[Point(0 1)@2012-01-01, Point(0 0)@2012-01-03)' <->
tgeompoint '[Point(0 0)@2012-01-01, Point(1 0)@2012-01-03)';
-- "[1@2012-01-01, 0.707106781186548@2012-01-02, 1@2012-01-03)"
SELECT tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-02)' <->
tgeompoint '[Point(0 1)@2012-01-01, Point(1 2)@2012-01-02)';
-- "[1@2012-01-01,1@2012-01-02)"
```

5.17 Topological Relationships for Temporal Points

The topological relationships such as `ST_Intersects` and `ST_Relate` can be generalized for temporal points. The arguments of these generalized functions are either a temporal point or a base type (that is, a `geometry` or `geography`), but these functions do not allow a base type in both arguments. Furthermore, both arguments must be of the same base type, that is, these functions do not allow to have a temporal geometry point (or a geometry) and a temporal geography point (or a geography) as arguments.

There are two versions of the temporal topological relationships:

- The first version applies the traditional topological function to the union of all values taken by the temporal point (which is a `geometry` or `geography`) and returns a `boolean` or a `text`. Examples are the `intersects` and `relate` functions.
- The second version is defined with the temporal semantics, that is, the traditional topological function is computed at each instant and results in a `tbool` or a `ttext`. Examples are the `tintersects` and `trelate` functions.

All spatial relationships in the two versions are defined for temporal geometry points, while only four of them are defined for temporal geography points, namely, `covers`, `coveredby`, `intersects`, and `dwithin`, and the corresponding temporal versions.

The semantics conveyed by the first version of the relationships varies depending on the relationship and the type of the arguments. For example, the following query

```
SELECT intersects(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
  tgeompoint '[Point(0 1)@2012-01-01, Point(1 1)@2012-01-03)');
```

tests whether the temporal point ever intersected the geometry, since the query is conceptually equivalent to the following one

```
SELECT ST_Intersects(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
  geometry 'Linestring(0 1,1 1)');
```

where the second geometry is obtained by applying the `trajectory` function to the temporal point. On the other hand, the query

```
SELECT contains(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
  tgeompoint '[Point(0 1)@2012-01-01, Point(1 1)@2012-01-03)');
```

tests whether the geometry always contains the temporal point. Finally, the following query

```
SELECT intersects(tgeompoint '[Point(0 1)@2012-01-01, Point(1 0)@2012-01-03)',
  tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
```

tests whether the temporal points may intersect, since the query above is conceptually equivalent to the following one

```
SELECT ST_Intersects('Linestring(0 1,1 0)', 'Linestring(0 0,1 1)');
```

The first versions of the relationships are typically used in combination with a spatio-temporal index when computing the temporal relationships. For example, the following query

```
SELECT T.TripId, R.RegionId, tintersects(T.Trip, R.Geom)
FROM Trips T, Regions R
WHERE intersects(T.Trip, R.Geom)
```

which verifies whether a trip `T` (which is a temporal point) intersects a region `R` (which is a geometry), will benefit rom a spatio-temporal index on the column `T.Trip` since the `intersects` function will automatically perform the bounding box comparison `T.Trip && R.Geom`. This is further explained later in this document.

Three topological relationships available in PostGIS are not provided in the temporal version.

- `tcontainsproperly` since it would always be equal to `tcontains`: `ST_Contains` returns true if and only if no points of B lie in the exterior of A, and at least one point of the interior of B lies in the interior of A. `ST_ContainsProperly` returns true if B intersects the interior of A but not the boundary (or exterior).
- `tcrosses` since it would always returns false: `ST_Crosses` returns true if the supplied geometries have some, but not all, interior points in common.
- `toverlaps` since it would always returns false: `ST_Overlaps` returns true if the geometries share space, are of the same dimension, but are not completely contained by each other.

Similarly, only a few temporal topological relationships are meaningful when the two arguments are temporal points. Therefore, the relationships supported for two temporal geometry points are `tdisjoint`, `tequals`, `tintersects`, `tdwithin`, and `trelate` (with 2 and 3 arguments), while only `tintersects` and `tdwithin` are supported for two temporal geography points.

The `relate` and the `trelate` functions have two forms with either two or three arguments. The two-argument forms consider the spatial relationship between the interior, the boundary, and the exterior of the arguments and return a `text` or a `ttext` value representing the maximum intersection matrix pattern. This pattern is defined using the Dimensionally Extended 9 Intersection Model or DE-9IM (see the PostGIS documentation for more details). The three-argument forms determine whether the first two arguments satisfy the intersection matrix pattern given as third argument (a `text` value) and return a Boolean or a temporal Boolean.

Finally, it is worth noting that the topological relationships allow to mix 2D/3D geometries but in that case, the computation is only performed on 2D.

5.17.1 Possible Spatial Relationships

- May contain

```
contains({geo, tgeompoint}, {geo, tgeompoint}):  boolean
```

```
SELECT contains(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
-- true
```

- May contain properly

```
containsproperly({geo, tgeompoint}, {geo, tgeompoint}):  boolean
```

```
SELECT containsproperly(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
-- false
```

- May cover

```
covers({geo, tpoint}, {geo, tpoint}):  boolean
```

```
SELECT covers(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
-- true
```

- May be covered by

```
coveredby({geo, tpoint}, {geo, tpoint}):  boolean
```

```
SELECT coveredby(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
-- false
```

- May cross

```
crosses({geo, tgeompoint}, {geo, tgeompoint}):  boolean
```

```
SELECT crosses(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-03)');
-- true
```

- May be disjoint

```
disjoint({geo, tgeompoint}, {geo, tgeompoint}):  boolean
```

```
SELECT disjoint(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
-- false
```

- May be equal

```
equals({geo, tgeompoint}, {geo, tgeompoint}):  boolean
```

```
SELECT equals(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
-- false
```

- May intersect 

  `intersects({geo, tpoint}, {geo, tpoint}): boolean`

```
SELECT intersects(geometry 'Polygon((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0))',
tgeompoint '[Point(0 0 1)@2012-01-01, Point(1 1 1)@2012-01-03)');
-- false
SELECT intersects(geometry 'Polygon((0 0 0,0 1 1,1 1 1,1 0 0,0 0 0))',
tgeompoint '[Point(0 0 1)@2012-01-01, Point(1 1 1)@2012-01-03)');
-- true
```

- May overlap

  `overlaps({geo, tgeompoint}, {geo, tgeompoint}): boolean`

```
SELECT overlaps(geometry 'Linestring(1 1,3 3)',
tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-03)');
-- true
```

- May touch

  `touches({geo, tgeompoint}, {geo, tgeompoint}): boolean`

```
SELECT touches(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(0 1)@2012-01-03)');
-- true
```

- May be within

  `within({geo, tgeompoint}, {geo, tgeompoint}): boolean`

```
SELECT within(geometry 'LineString(1 1,2 2)',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-03)');
-- true
```

- May be at distance within 

  `dwithin({geo, tpoint}, {geo, tpoint}, double): boolean`

```
SELECT dwithin(geometry 'Polygon((0 0 0,0 1 1,1 1 1,1 0 0,0 0 0))',
tgeompoint 'Point(0 2 1)@2000-01-01,Point(2 2 1)@2000-01-02', 1)
-- true
SELECT dwithin(geometry 'Polygon((0 0 0,0 1 1,1 1 1,1 0 0,0 0 0))',
tgeompoint 'Point(0 2 2)@2000-01-01,Point(2 2 2)@2000-01-02', 1)
--  false
```

- May relate

  `relate({geo, tgeompoint}, {geo, tgeompoint}): text`

  `relate({geo, tgeompoint}, {geo, tgeompoint}, text): boolean`

```
SELECT relate(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)');
-- "1F2F01FF2"
SELECT relate(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(1 1)@2012-01-03)','1F2F01FF2');
-- true
```

5.17.2 Temporal Spatial Relationships

- Temporal contains

  `tcontains({geo, tgeompoint}, {geo, tgeompoint}): tbool`

```
SELECT tcontains(geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)');
-- "{[f@2012-01-01, f@2012-01-02], (t@2012-01-02, f@2012-01-03, f@2012-01-04)}"
```

- Temporal covers 

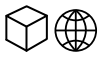
  `tcovers({geo, tpoint}, {geo, tpoint}): tbool`

```
SELECT tcovers(geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)');
-- "{[f@2012-01-01, t@2012-01-02, t@2012-01-03], (f@2012-01-03, f@2012-01-04]}"
```

- Temporal covered by 

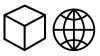
  `tcoveredby({geo, tpoint}, {geo, tpoint}): tbool`

```
SELECT tcoveredby(geometry 'Point(1 1)',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)');
-- "{[f@2012-01-01, t@2012-01-02], (f@2012-01-02, f@2012-01-04)}"
```

- Temporal disjoint 

  `tdisjoint({geo, tgeompoint}, {geo, tgeompoint}): tbool`

```
SELECT tdisjoint(geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)');
-- "{[t@2012-01-01, f@2012-01-02, f@2012-01-03], (t@2012-01-03, t@2012-01-04]}"
SELECT tdisjoint(tgeompoint '[Point(0 3)@2012-01-01, Point(3 0)@2012-01-05)',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-05)');
-- "{[t@2012-01-01, f@2012-01-03], (t@2012-01-03, t@2012-01-05)}"
```

- Temporal equals 

  `tequals({point, tgeompoint}, {point, tgeompoint}): tbool`

```
SELECT tequals(geometry 'Point(1 1)',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)');
-- "{[f@2012-01-01, t@2012-01-02], (f@2012-01-02, f@2012-01-04]}"
SELECT tequals(tgeompoint '[Point(0 3)@2012-01-01, Point(3 0)@2012-01-05)',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-05)');
-- "{[f@2012-01-01, t@2012-01-03], (f@2012-01-03, f@2012-01-05)}"
```

- Temporal intersects 

  `tintersects({geo, tpoint}, {geo, tpoint}): tbool`

```
SELECT tintersects(geometry 'MultiPoint(1 1,2 2)',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-04)');
-- "{[f@2012-01-01, t@2012-01-02], (f@2012-01-02, t@2012-01-03],
(f@2012-01-03, f@2012-01-04]}"
SELECT tintersects(tgeompoint '[Point(0 3)@2012-01-01, Point(3 0)@2012-01-05)',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 3)@2012-01-05)');
-- "{[f@2012-01-01, t@2012-01-03], (f@2012-01-03, f@2012-01-05)}"
```

- Temporal touches

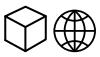
  `ttouches({geo, tgeompoint}, {geo, tgeompoint}): tbool`

```
SELECT ttouches(geometry 'Polygon((1 0,1 2,2 2,2 0,1 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 0)@2012-01-04)');
-- "{[f@2012-01-01, t@2012-01-02, t@2012-01-03], (f@2012-01-03, f@2012-01-04]}"
```

- Temporal within

  `twithin({geo, tgeompoint}, {geo, tgeompoint}): tbool`

```
SELECT twithin(geometry 'Point(1 1)',
tgeompoint '[Point(0 0)@2012-01-01, Point(2 2)@2012-01-03)');
-- "{[f@2012-01-01, t@2012-01-02], (f@2012-01-02, f@2012-01-03]}"
```

- Temporal distance within 

  `tdwithin({geo, tpoint}, {geo, tpoint}, double): tbool`

```
SELECT tdwithin(geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 0)@2012-01-04)', 1);
-- "{[f@2012-01-01, t@2012-01-02, t@2012-01-03], (f@2012-01-03, f@2012-01-04)}"
SELECT tdwithin(tgeompoint '[Point(1 0)@2000-01-01, Point(1 4)@2000-01-05]',
tgeompoint 'Interp=Stepwise;[Point(1 2)@2000-01-01, Point(1 3)@2000-01-05]', 1);
-- "{[f@2000-01-01, t@2000-01-02, t@2000-01-04], (f@2000-01-04, t@2000-01-05]}"
```

- Temporal relate

  `trelate({geo, tgeompoint}, {geo, tgeompoint}, text): tbool`

  `trelate({geo, tgeompoint}, {geo, tgeompoint}): ttext`

```
SELECT trelate(geometry 'Polygon((1 0,1 2,2 2,2 0,1 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 0)@2012-01-04)');
-- "{[FF2FF10F2@2012-01-01, FF20F1FF2@2012-01-02, FF20F1FF2@2012-01-03],
(FF2FF10F2@2012-01-03, FF2FF10F2@2012-01-04]}"
SELECT trelate(geometry 'Polygon((1 0,1 2,2 2,2 0,1 0))',
tgeompoint '[Point(0 0)@2012-01-01, Point(3 0)@2012-01-04)', 'FF20F1FF2');
-- "{[f@2012-01-01, t@2012-01-02, t@2012-01-03], (f@2012-01-03, f@2012-01-04]}"
2012-01-04)}"
```

5.18 Aggregate Functions

The temporal aggregate functions generalize the traditional aggregate functions. Their semantics is that they compute the value of the function at every instant in the *union* of the temporal extents of the values to aggregate. In contrast, recall that all other functions manipulating temporal types compute the value of the function at every instant in the *intersection* of the temporal extents of the arguments.

The temporal aggregate functions are the following ones:

- For all temporal types, the function `tcount` generalize the traditional function `count`. The temporal count can be used to compute at each point in time the number of available objects (for example, number of cars in an area).
- For all temporal types, function `extent` returns a bounding box that encloses a set of temporal values. Depending on the base type, the result of this function can be a `period`, a `tbox` or an `stbox`.
- For the temporal Boolean type, the functions `tand` and `tor` generalize the traditional functions `and` and `or`.
- For temporal numeric types, there are two types of temporal aggregate functions. The functions `tmin`, `tmax`, `tsum`, and `tavg` generalize the traditional functions `min`, `max`, `sum`, and `avg`. Furthermore, the functions `wmin`, `wmax`, `wcount`, `wsum`, and `wavg` are window (or cumulative) versions of the traditional functions that, given a time interval w, compute the value of the function at an instant t by considering the values during the interval [t-w, t]. All window aggregate functions are available for temporal integers, while for temporal floats only window minimum and maximum are meaningful.
- For the temporal text type, the functions `tmin` y `tmax` generalize the traditional functions `min` and `max`.
- Finally, for temporal point types, the function `tcentroid` generalizes the function `ST_Centroid` provided by PostGIS. For example, given set of objects that move together (that is, a convoy or a flock) the temporal centroid will produce a temporal point that represents at each instant the geometric center (or the center of mass) of all the moving objects.

In the examples that follow, we suppose the tables `Department` and `Trip` contain the two tuples introduced in Section 3.1.

- Temporal count

  `tcount(ttype): {tinti, tints}`

```
SELECT tcount(NoEmps) FROM Department;
-- "{[1@2012-01-01, 2@2012-02-01, 1@2012-08-01, 1@2012-10-01)}"
```

- Bounding box extent

  `extent(temp): {period, tbox, stbox}`

```
SELECT extent(noEmps) FROM Department;
-- "TBOX((4,2012-01-01 00:00:00+01),(12,2012-10-01 00:00:00+02))"
SELECT extent(Trip) FROM Trips;
-- "STBOX T((0,0,2012-01-01 08:00:00+01),(3,3,2012-01-01 08:20:00+01))"
```

- Temporal and

  `tand(tbool): tbool`

```
SELECT tand(NoEmps #> 6) FROM Department;
-- "{[t@2012-01-01, f@2012-04-01, f@2012-10-01)}"
```

- Temporal or

  `tor(tbool): tbool`

```
SELECT tor(NoEmps #> 6) FROM Department;
-- "{[t@2012-01-01, f@2012-08-01, f@2012-10-01)}"
```

- Temporal minimum

  `tmin(ttype): {ttypei, ttypes}`

```
SELECT tmin(NoEmps) FROM Department;
-- "{[10@2012-01-01, 4@2012-02-01, 6@2012-06-01, 6@2012-10-01)}"
```

- Temporal maximum

  `tmax(ttype): {ttypei, ttypes}`

```
SELECT tmax(NoEmps) FROM Department;
-- "{[10@2012-01-01, 12@2012-04-01, 6@2012-08-01, 6@2012-10-01)}"
```

- Temporal sum

  `tsum(tnumber): {tnumi, tnums}`

```
SELECT tsum(NoEmps) FROM Department;
-- "{[10@2012-01-01, 14@2012-02-01, 16@2012-04-01, 18@2012-06-01, 6@2012-08-01,
6@2012-10-01)}"
```

- Temporal average

  `tavg(tnumber): {tfloati, tfloats}`

```
SELECT tavg(NoEmps) FROM Department;
-- "{[10@2012-01-01, 10@2012-02-01), [7@2012-02-01, 7@2012-04-01),
[8@2012-04-01, 8@2012-06-01), [9@2012-06-01, 9@2012-08-01),
[6@2012-08-01, 6@2012-10-01)"
```

- Window minimum

  `wmin(tnumber, interval): {tnumi, tnums}`

```
SELECT wmin(NoEmps, interval '2 days') FROM Department;
-- "{[10@2012-01-01, 4@2012-04-01, 6@2012-06-03, 6@2012-10-03)}"
```

- Window maximum

  `wmax(tnumber, interval): {tnumi, tnums}`

```
SELECT wmax(NoEmps, interval '2 days') FROM Department;
-- "{[10@2012-01-01, 12@2012-04-01, 6@2012-08-03, 6@2012-10-03)}"
```

- Window count

  `wcount(tnumber, interval): {tinti, tints}`

```
SELECT wcount(NoEmps, interval '2 days') FROM Department;
-- "{[1@2012-01-01, 2@2012-02-01, 3@2012-04-01, 2@2012-04-03, 3@2012-06-01, 2@2012-06-03,
1@2012-08-03, 1@2012-10-03)}"
```

- Window sum

  `wsum(tint, interval): {tinti, tints}`

```
SELECT wsum(NoEmps, interval '2 days') FROM Department;
-- "{[10@2012-01-01, 14@2012-02-01, 26@2012-04-01, 16@2012-04-03, 22@2012-06-01,
18@2012-06-03, 6@2012-08-03, 6@2012-10-03)}"
```

- Window average

  `wavg(tint, interval): {tfloati, tfloats}`

```
SELECT wavg(NoEmps, interval '2 days') FROM Department;
-- "{[10@2012-01-01, 10@2012-02-01), [7@2012-02-01, 7@2012-04-01),
[8.6666666666667@2012-04-01, 8.66666666666667@2012-04-03),
[8@2012-04-03, 8@2012-06-01),
[7.33333333333333@2012-06-01, 7.33333333333333@2012-06-03),
[9@2012-06-03, 9@2012-08-03), [6@2012-08-03, 6@2012-10-03)}"
```

- Temporal centroid

  `tcentroid(tgeompoint): tgeompoint`

```
SELECT tcentroid(Trip) FROM Trips;
-- "{[POINT(0 0)@2012-01-01 08:00:00+00, POINT(1 0)@2012-01-01 08:05:00+00),
[POINT(0.5 0)@2012-01-01 08:05:00+00, POINT(1.5 0.5)@2012-01-01 08:10:00+00,
POINT(2 1.5)@2012-01-01 08:15:00+00),
[POINT(2 2)@2012-01-01 08:15:00+00, POINT(3 3)@2012-01-01 08:20:00+00)}"
```

5.19 Utility Functions

- Version of the MobilityDB extension

  `mobilitydb_version(): text`

```
SELECT mobilitydb_version();
-- "MobilityDB 1.0"
```

- Versions of the MobilityDB extension and its dependencies

  `mobilitydb_full_version(): text`

```
SELECT mobilitydb_full_version();
-- "MobilityDB 1.0 PostgreSQL 12.3 PostGIS 2.5"
```

5.20 Indexing of Temporal Types

GiST and SP-GiST indexes can be created for table columns of temporal types. The GiST index implements an R-tree for temporal alphanumeric types and for temporal point types. The SP-GiST index implements a Quad-tree for temporal alphanumeric types and an Oct-tree for temporal point types. Examples of index creation are as follows:

```
CREATE INDEX Department_NoEmps_Gist_Idx ON Department USING Gist(NoEmps);
CREATE INDEX Trips_Trip_SPGist_Idx ON Trips USING SPGist(Trip);
```

The GiST and SP-GiST indexes store the bounding box for the temporal types. As explained in Chapter 3, these are

- the period `period` type for the `tbool` and `ttext` types,
- the `tbox` type for the `tint` and `tfloat` types,
- the `stbox` type for the `tgeompoint` and `tgeogpoint` types.

A GiST or SP-GiST index can accelerate queries involving the following operators (see Section 5.9 for more information):

- `<<`, `&<`, `&>`, `>>`, which only consider the value dimension in temporal alphanumeric types,
- `<<`, `&<`, `&>`, `>>`, `<<|`, `&<|`, `|&>`, `|>>`, `&</`, `<</`, `/>>`, and `/&>`, which only consider the spatial dimension in temporal point types,
- `&<#`, `<<#`, `#>>`, `#&>`, which only consider the time dimension for all temporal types,
- `&&`, `@>`, `<@`, and `~=`, which consider as many dimensions as they are shared by the indexed column and the query argument. These operators work on bounding boxes (that is, `period`, `tbox`, or `stbox`), not the entire values.

In addition, a GiST index can accelerate nearest neighbor queries involving the `|=|` operator.

For example, given the index defined above on the `Department` table and a query that involves a condition with the `&&` (overlaps) operator, if the right argument is a temporal float then both the value and the time dimensions are considered for filtering the tuples of the relation, while if the right argument is a float value, a float range, or a time type, then either the value or the time dimension will be used for filtering the tuples of the relation. Furthermore, a bounding box can be constructed from a value/range and/or a timestamp/period, which can be used for filtering the tuples of the relation. Examples of queries using the index on the `Department` table defined above are given next.

```
SELECT * FROM Department WHERE NoEmps && 5;
SELECT * FROM Department WHERE NoEmps && intrange '[1, 5)';
SELECT * FROM Department WHERE NoEmps && timestamptz '2012-04-01';
SELECT * FROM Department WHERE NoEmps && period '[2012-04-01, 2012-05-01)';
SELECT * FROM Department WHERE NoEmps &&
tbox(intrange '[1, 5)', period '[2012-04-01, 2012-05-01)');
SELECT * FROM Department WHERE NoEmps &&
tfloat '{[1@2012-01-01, 1@2012-02-01), [5@2012-04-01, 5@2012-05-01)}';
```

Similarly, examples of queries using the index on the `Trips` table defined above are given next.

```
SELECT * FROM Trips WHERE Trip && geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))';
SELECT * FROM Trips WHERE Trip && timestamptz '2001-01-01';
SELECT * FROM Trips WHERE Trip && period '[2001-01-01, 2001-01-05)';
SELECT * FROM Trips WHERE Trip &&
stbox(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))', period '[2001-01-01, 2001-01-05]');
SELECT * FROM Trips WHERE Trip &&
tgeompoint '{[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02, Point(1 1)@2001-01-05)}';
```

Finally, B-tree indexes can be created for table columns of all temporal types. For this index type, the only useful operation is equality. There is a B-tree sort ordering defined for values of temporal types, with corresponding <, <=, >, >= and operators, but the ordering is rather arbitrary and not usually useful in the real world. B-tree support for temporal types is primarily meant to allow sorting internally in queries, rather than creation of actual indexes.

In order to speed up several of the functions in Chapter 5, we can add in the `WHERE` clause of queries a bounding box comparison that make uses of the available indexes. For example, this would be typically the case for the functions that project the temporal types to the value/spatial and/or time dimensions. This will filter out the tuples with an index as shown in the following query.

```
SELECT atPeriod(T.Trip, period(2001-01-01, 2001-01-02))
FROM Trips T
-- Bouding box index filtering
WHERE T.Trip && period(2001-01-01, 2001-01-02)
```

In the case of temporal points, all spatial relationships with the possible semantics (see Section 5.17), excepted `disjoint`and `relate`, automatically include a bounding box comparison that will make use of any indexes that are available on the temporal points. For this reason, the first version of the relationships is typically used for filtering the tuples with the help of an index when computing the temporal relationships as shown in the following query.

```
SELECT tintersects(T.Trip, R.Geom)
FROM Trips T, Regions R
-- Bouding box index filtering
WHERE intersects(T.Trip, R.Geom);
```

5.21 Statistics and Selectivity for Temporal Types

5.21.1 Statistics Collection

The PostgreSQL planner relies on statistical information about the contents of tables in order to generate the most efficient execution plan for queries. These statistics include a list of some of the most common values in each column and a histogram showing the approximate data distribution in each column. For large tables, a random sample of the table contents is taken, rather than examining every row. This enables large tables to be analyzed in a small amount of time. The statistical information is gathered by the `ANALYZE` command and stored in the `pg_statistic` catalog table. Since different kinds of statistics may be appropriate for different kinds of data, the table only stores very general statistics (such as number of null values) in dedicated columns. Everything else is stored in five "slots", which are couples of array columns that store the statistics for a column of an arbitrary type.

The statistics collected for time types and temporal types are based on those collected by PostgreSQL for scalar types and range types. For scalar types, such as `float`, the following statistics are collected:

1. fraction of null values,
2. average width, in bytes, of non-null values,

3. number of different non-null values,
4. array of most common values and array of their frequencies,
5. histogram of values, where the most common values are excluded,
6. correlation between physical and logical row ordering.

For range types, like `tstzrange`, three additional histograms are collected:

7. length histogram of non-empty ranges,
8. histograms of lower and upper bounds.

For geometries, in addition to (1)–(3), the following statistics are collected:

9. number of dimensions of the values, N-dimensional bounding box, number of rows in the table, number of rows in the sample, number of non-null values,
10. N-dimensional histogram that divides the bounding box into a number of cells and keeps the proportion of values that intersects with each cell.

The statistics collected for columns of the new time types `timestampset`, `period`, and `periodset` replicate those collected by PostgreSQL for the `tstzrange`. This is clear for the `period` type, which is equivalent to `tszrange`, excepted that periods cannot be empty. For the `timestampset` and the `periodset` types, a value is converted into its bounding box which is a `period`, then the statistics for the `period` type are collected.

The statistics collected for columns of temporal types depend on their subtype and their base type. In addition to statistics (1)–(3) that are collected for all temporal types, statistics are collected for the time and the value dimensions independently. More precisely, the following statistics are collected for the time dimension:

- For columns of instant subtype, the statistics (4)–(6) are collected for the timestamps.
- For columns of other subtype, the statistics (7)–(8) are collected for the (bounding box) periods.

The following statistics are collected for the value dimension:

- For columns of temporal types with stepwise interpolation (that is, `tbool`, `ttext`, or `tint`):
  - For the instant subtype, the statistics (4)–(6) are collected for the values.
  - For all other subtypes, the statistics (7)–(8) are collected for the values.
- For columns of the temporal float type (that is, `tfloat`):
  - For the instant subtype, the statistics (4)–(6) are collected for the values.
  - For all other subtype, the statistics (7)–(8) are collected for the (bounding) value ranges.
- For columns of temporal point types (that is, `tgeompoint` and `tgeogpoint`) the statistics (9)–(10) are collected for the points.

5.21.2 Selectivity Estimation of Operators

Boolean operators in PostgreSQL can be associated with two selectivity functions, which compute how likely a value of a given type will match a given criterion. These selectivity functions rely on the statistics collected. There are two types of selectivity functions. The *restriction* selectivity functions try to estimate the percentage of the rows in a table that satisfy a `WHERE`-clause condition of the form `column OP constant`. On the other hand, the *join* selectivity functions try to estimate the percentage of the rows in a table that satisfy a `WHERE`-clause condition of the form `table1.column1 OP table2.column2`.

MobilityDB defines 23 classes of Boolean operators (such as =, <, &&, <<, etc.), each of which can have as left or right arguments a PostgreSQL type (such as `int`, `timestamptz`, etc.) or a new MobilityDB type (such as `period`, `tintseq`, etc.). As a consequence, there is a very high number of operators with different arguments to be considered for the selectivity functions. The approach taken was to group these combinations into classes corresponding to the value and time dimensions. The classes correspond to the type of statistics collected as explained in the previous section.

Currently, only restriction selectivity functions are implemented for temporal types, while join selectivity functions give a default selectivity value depending on the operator. It is planned to implement joint selectivity functions in the future.

Appendix A

MobilityDB Reference

A.1 Functions and Operators for Time Types and Range Types

A.1.1 Constructor Functions

- period: Constructor for `period`
- timestampset: Constructor for `timestampset`
- periodset: Constructor for `periodset`

A.1.2 Casting

- timestamptz::time: Cast a `timestamptz` to another time type
- timestampset::periodset: Cast a `timestampset` to a `periodset`
- period::type: Cast a `period` to another time type
- tstzrange::period: Cast a `tstzrange` to a `period`

A.1.3 Accessor Functions

- memSize: Get the memory size in bytes
- lower: Get the lower bound
- upper: Get the upper bound
- lower_inc: Is the lower bound inclusive?
- upper_inc: Is the upper bound inclusive?
- duration: Get the duration
- timespan: Get the timespan ignoring the potential time gaps
- period: Get the period on which the timestamp set or period set is defined ignoring the potential time gaps
- numTimestamps: Get the number of different timestamps
- startTimestamp: Get the start timestamp
- endTimestamp: Get the end timestamp

- timestampN: Get the n-th different timestamp
- timestamps: Get the different timestamps
- numPeriods: Get the number of periods
- startPeriod: Get the start period
- endPeriod: Get the end period
- periodN: Get the n-th period
- periods: Get the periods
- shift: Shift the time value by an interval

A.1.4 Comparison Operators

- =: Are the time values equal?
- <>: Are the time values different?
- <: Is the first time value less than the second one?
- >: Is the first time value greater than the second one?
- <=: Is the first time value less than or equal to the second one?
- >=: Is the first time value greater than or equal to the second one?

A.1.5 Set Operators

- +: Union of the time values
- *: Intersection of the time values
- -: Difference of the time values

A.1.6 Topological and Relative Position Operators

- &&: Do the time values overlap (have instants in common)?
- @>: Does the first time value contain the second one?
- <@: Is the first time value contained by the second one?
- -|-: Is the first time value adjacent to the second one?
- <<: Is the first number or range value scritly left of the second one?
- >>: Is the first number or range value strictly right of the second one?
- &<: Is the first number or range value not to the right of the second one?
- &>: Is the first number or range value not to the left of the second one?
- -|-: Is the first number or range value adjacent to the second one?
- <<#: Is the first time value scritly before the second one?
- #>>: Is the first time value strictly after the second one?
- &<#: Is the first time value not after the second one?
- #&>: Is the first time value not before the second one?

A.1.7 Aggregate Functions

- tcount: Temporal count
- extent: Bounding period
- tunion: Temporal union

A.2 Functions and Operators for Box Types

A.2.1 Constructor Functions

- tbox: Constructor for `tbox`
- stbox, stboxt: Constructor for `stbox`

A.2.2 Casting

- tbox::type: Cast a `tbox` to another type
- type::tbox: Cast another type to a `tbox`
- stbox::type: Cast an `stbox` to anoter type
- type::stbox: Cast another type to an `stbox`

A.2.3 Accessor Functions

- hasX: Has X dimension?
- hasZ: Has Z dimension?
- hasT: Has T dimension?
- Xmin: Get the minimum X value
- Xmax: Get the maximum X value
- Ymin: Get the minimum Y value
- Ymax: Get the maximum Y value
- Zmin: Get the minimum Z value
- Zmax: Get the maximum Z value
- Tmin: Get the minimum T value
- Tmax: Get the maximum T value

A.2.4 Modification Functions

- expandValue: Expand the numeric value dimension of the bounding box by a float value
- expandSpatial: Expand the spatial value dimension of the bounding box by a float value
- expandTemporal: Expand the temporal dimension of the bounding box by a time interval
- setPrecision: Round the value or the coordinates of the bounding box to a number of decimal places

A.2.5 Spatial Reference System Functions

- SRID: Get the spatial reference identifier
- setSRID: Set the spatial reference identifier
- transform: Transform to a different spatial reference

A.2.6 Comparison Operators

- =: Are the bounding boxes equal?
- <>: Are the bounding boxes different?
- <: Is the first bouding box less than the second one?
- >: Is the first bouding box greater than the second one?
- <=: Is the first bouding box less than or equal to the second one?
- >=: Is the first bouding box greater than or equal to the second one?

A.2.7 Set Operators

- +: Union of the bounding boxes
- *: Intersection of the bounding boxes

A.2.8 Topological Operators

- &&: Do the bounding boxes overlap?
- @>: Does the first bounding box contain the second one?
- <@: Is the first bounding box contained in the second one?
- ~=: Are the bounding boxes equal in their common dimensions?
- -|-: Are the bounding boxes adjacent?

A.2.9 Relative Position Operators

- <<: Are the X values of the first bounding box strictly less than those of the second one?
- >>: Are the X values of the first bounding box strictly greater than those of the second one?
- &<: Are the X values of the first bounding box not greater than those of the second one?
- &>: Are the X values of the first bounding box not less than those of the second one?
- <<: Are the X values of the first bounding box strictly to the left of those of the second one?
- >>: Are the X values of the first bounding box strictly to the right of those of the second one?
- &<: Are the X values of the first bounding box not to the right of those of the second one?
- &>: Are the X values of the first bounding box not to the left of those of the second one?
- <<|: Are the Y values of the first bounding box strictly below of those of the second one?
- |>>: Are the Y values of the first bounding box strictly above of those of the second one?

- &<|: Are the Y values of the first bounding box not above of those of the second one?
- |&>: Are the Y values of the first bounding box not below of those of the second one?
- <</: Are the Z values of the first bounding box strictly in front of those of the second one?
- />>: Are the Z values of the first bounding box strictly back of those of the second one?
- &</: Are the Z values of the first bounding box not back of those of the second one?
- /&>: Are the Z values of the first bounding box not in front of those of the second one?
- <<#: Are the T values of the first bounding box strictly before those of the second one?
- #>>: Are the T values of the first bounding box strictly after those of the second one?
- &<#: Are the T values of the first bounding box not after those of the second one?
- #&>: Are the T values of the first bounding box not before those of the second one?

A.3 Functions and Operators for Temporal Types

A.3.1 Constructor Functions

- ttypeinst: Constructor for temporal types of instant subtype
- ttypei: Constructor for temporal types of instant set subtype
- ttypeseq: Constructor for temporal types of sequence subtype
- ttypes: Constructor for temporal types of sequence set subtype

A.3.2 Casting

- tint::tfloat: Cast a temporal integer to a temporal float
- tfloat::tint: Cast a temporal float to a temporal integer
- tgeompoint::tgeogpoint: Cast a temporal geometry point to a temporal geography point
- tgeogpoint::tgeompoint: Cast a temporal geography point to a temporal geometry point
- tgeompoint::geometry, tgeogpoint::geography: Cast a temporal point to a PostGIS trajectory
- geometry::tgeompoint, geography::tgeogpoint: Cast a PostGIS trajectory to a temporal point

A.3.3 Transformation Functions

- ttypeinst, ttypei, ttypeseq, ttypes: Transform a temporal value to another subtype
- toLinear: Transform a temporal value with continuous base type from stepwise to linear interpolation
- appendInstant: Append a temporal instant to a temporal value
- merge: Merge temporal values

A.3.4 Accessor Functions

- memSize: Get the memory size in bytes
- tempSubtype: Get the temporal subtype
- interpolation: Get the interpolation
- getValue: Get the value
- getValues: Get the values
- startValue: Get the start value
- endValue: Get the end value
- minValue: Get the minimum value
- maxValue: Get the maximum value
- valueRange: Get the value range
- valueAtTimestamp: Get the value at a timestamp
- getTimestamp: Get the timestamp
- getTime: Get the time
- duration: Get the duration
- timespan: Get the timespan ignoring the potential time gaps
- period: Get the period on which the temporal value is defined ignoring the potential time gaps
- numInstants: Get the number of different instants
- startInstant: Get the start instant
- endInstant: Get the end instant
- instantN: Get the n-th different instant
- instants: Get the different instants
- numTimestamps: Get the number of different timestamps
- startTimestamp: Get the start timestamp
- endTimestamp: Get the end timestamp
- timestampN: Get the n-th different timestamp
- timestamps: Get the different timestamps
- numSequences: Get the number of sequences
- startSequence: Get the start sequence
- endSequence: Get the end sequence
- sequenceN: Get the n-th sequence
- sequences: Get the sequences
- segments: Get the segments
- shift: Shift the time span of the temporal value by an interval
- tscale: Scale the time span of the temporal value to an interval

- shiftTscale: Shift and scale the time span the temporal value with the intervals
- intersectsTimestamp: Does the temporal value intersect the timestamp?
- intersectsTimestampSet: Does the temporal value intersect the timestamp set?
- intersectsPeriod: Does the temporal value intersect the period?
- intersectsPeriodSet: Does the temporal value intersect the period set?
- twAvg: Get the time-weighted average

A.3.5 Spatial Functions

- asText: Get the Well-Known Text (WKT) representation
- asEWKT: Get the Extended Well-Known Text (EWKT) representation
- asMFJSON: Get the Moving Features JSON representation
- asBinary: Get the Well-Known Binary (WKB) representation
- asEWKB: Get the Extended Well-Known Binary (EWKB) representation
- asHexEWKB: Get the Hexadecimal Extended Well-Known Binary (EWKB) representation as text
- fromMFJSON: Input a temporal point from a Moving Features JSON representation
- fromEWKB: Input a temporal point from an Extended Well-Known Binary (EWKB) representation
- SRID: Get the spatial reference identifier
- setSRID: Set the spatial reference identifier
- transform: Transform to a different spatial reference
- setPrecision: Round the coordinate values to a number of decimal places
- getX: Get the X coordinate values as a temporal float
- getY: Get the Y coordinate values as a temporal float
- getZ: Get the Z coordinate values as a temporal float
- length: Get the length traversed by the temporal point
- cumulativeLength: Get the cumulative length traversed by the temporal point
- speed: Get the speed of the temporal point in units per second
- twCentroid: Get the time-weighted centroid
- azimuth: Get the temporal azimuth
- nearestApproachInstant: Get the instant of the first temporal point at which the two arguments are at the nearest distance
- nearestApproachDistance: Get the smallest distance ever
- shortestLine: Get the line connecting the nearest approach point
- simplify: Simplify a temporal point using a generalization of the Douglas-Peucker algorithm
- geoMeasure: Construct a geometry/geography with M measure from a temporal point and a temporal float

A.3.6 Restriction Functions

- atValue: Restrict to a value
- atValues: Restrict to an array of values
- atRange: Restrict to a range
- atRanges: Restrict to an array of ranges
- atMin: Restrict to the minimum value
- atMax: Restrict to the maximum value
- atGeometry: Restrict to a geometry
- atTimestamp: Restrict to a timestamp
- atTimestampSet: Restrict to a timestamp set
- atPeriod: Restrict to a period
- atPeriodSet: Restrict to a period set
- atTbox: Restrict to a `tbox`
- atStbox: Restrict to an `stbox`

A.3.7 Difference Functions

- minusValue: Difference with a value
- minusValues: Difference with an array of values
- minusRange: Difference with a range
- minusRanges: Difference with an array of ranges
- minusMin: Difference with the minimum value
- minusMax: Difference with the maximum value
- minusGeometry: Difference with a geometry
- minusTimestamp: Difference with a timestamp
- minusTimestampSet: Difference with a timestamp set
- minusPeriod: Difference with period
- minusPeriodSet: Difference with a period set
- minusTbox: Difference with a `tbox`
- minusStbox: Difference with an `stbox`

A.3.8 Comparison Operators

- =: Are the temporal values equal?
- <>: Are the temporal values different?
- <: Is the first temporal value less than the second one?
- >: Is the first temporal value greater than the second one?
- <=: Is the first temporal value less than or equal to the second one?
- >=: Is the first temporal value greater than or equal to the second one?

A.3.9 Ever and Always Comparison Operators

- ?=: Is the temporal value ever equal to the value?
- ?<>: Is the temporal value ever different from the value?
- ?<: Is the temporal value ever less than the value?
- ?>: Is the temporal value ever greater than the value?
- ?<=: Is the temporal value ever less than or equal to the value?
- ?>=: Is the temporal value ever greater than or equal to the value?
- %=: Is the temporal value always equal to the value?
- %<>: Is the temporal value always different to the value?
- %<: Is the temporal value always less than the value?
- %>: Is the temporal value always greater than the value?
- %<=: Is the temporal value always less than or equal to the value?
- %>=: Is the temporal value always greater than or equal to the value?

A.3.10 Temporal Comparison Operators

- #=: Temporal equal
- #<>: Temporal different
- #<: Temporal less than
- #>: Temporal greater than
- #<=: Temporal less than or equal to
- #>=: Temporal greater than or equal to

A.3.11 Mathematical Functions and Operators

- +: Temporal addition
- -: Temporal subtraction
- *: Temporal multiplication
- /: Temporal division
- round: Round the values to a number of decimal places
- degrees: Convert from radians to degrees
- derivative: Get the derivative over time of the temporal float in units per second

A.3.12 Boolean Operators

- &: Temporal and
- |: Temporal or
- ~: Temporal not

A.3.13 Text Functions and Operators

- ||: Temporal text concatenation
- upper: Transform to uppercase
- lower: Transform to lowercase

A.3.14 Distance Operators

- |=|: Get the smallest distance ever
- <->: Get the temporal distance

A.3.15 Spatial Relationships for Temporal Points

A.3.15.1 Possible Spatial Relationships

- contains: May contain
- containsproperly: May contain properly
- covers: May cover
- coveredby: May be covered by
- crosses: May cross
- disjoint: May be disjoint
- equals: May be equal
- intersects: May intersect
- overlaps: May overlap
- touches: May touch
- within: May be within
- dwithin: May be at distance within
- relate: May relate

A.3.15.2 Temporal Spatial Relationships

- tcontains: Temporal contains
- tcovers: Temporal covers
- tcoveredby: Temporal covered by
- tdisjoint: Temporal disjoint
- tequals: Temporal equals
- tintersects: Temporal intersects
- ttouches: Temporal touches
- twithin: Temporal within
- tdwithin: Temporal distance within
- trelate: Temporal relate

A.3.16 Aggregate Functions

- tcount: Temporal count
- extent: Bounding box extent
- tand: Temporal and
- tor: Temporal or
- tmin: Temporal minimum
- tmax: Temporal maximum
- tsum: Temporal sum
- tavg: Temporal average
- wmin: Window minimum
- wmax: Window maximum
- wcount: Window count
- wsum: Window sum
- wavg: Window average
- tcentroid: Temporal centroid

A.3.17 Utility Functions

- mobilitydb_version: Get the version of the MobilityDB extension
- mobilitydb_full_version: Get the versions of the MobilityDB extension and its dependencies

Chapter 6

Index