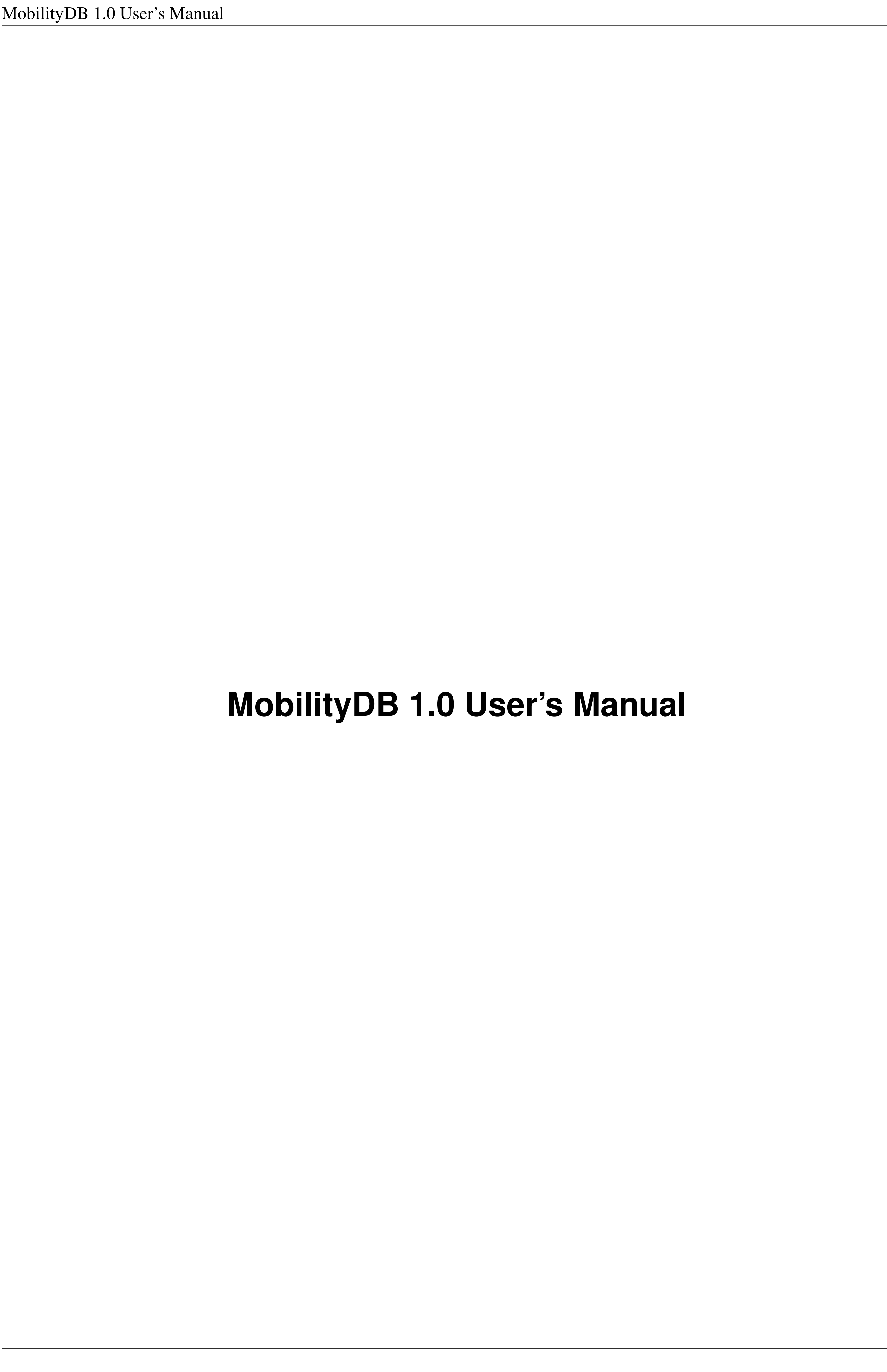

MobilityDB 1.0 User's Manual

COLLABORATORS			
	TITLE : MobilityDB 1.0 User's Manual		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Esteban Zimányi	May 27, 2022	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

1 Introduction **1**
1.1 Project Steering Committee 1
1.2 Other Code Contributors 2
1.3 Sponsors 2
1.3.1 Research Sponsors 2
1.3.2 Corporate Sponsors 2
1.4 Licenses 2
1.5 Installation 2
1.5.1 Short Version 2
1.5.2 Get the Sources 3
1.5.3 Enabling the Database 4
1.5.4 Dependencies 4
1.5.5 Configuring 5
1.5.6 Build and Install 5
1.5.7 Testing 6
1.5.8 Documentation 6
1.6 Support 7
1.6.1 Reporting Problems 7
1.6.2 Mailing Lists 7

2 Time Types and Range Types **8**
2.1 Functions and Operators for Time Types and Range Types 9
2.1.1 Constructor Functions 9
2.1.2 Casting 10
2.1.3 Accessor Functions 11
2.1.4 Modification Functions 14
2.1.5 Comparison Operators 15
2.1.6 Set Operators 16
2.1.7 Topological Operators 16
2.1.8 Relative Position Operators 17
2.1.9 Distance Operators 18
2.1.10 Aggregate Functions 19
2.2 Indexing of Time Types 20

3 Temporal Types 22
3.1 Examples of Temporal Types . . . 24
3.2 Validity of Temporal Types . . . 26

4 Manipulating Bounding Box Types 27
4.1 Input/Output of Bounding Box Types . . . 27
4.2 Constructor Functions . . . 28
4.3 Casting . . . 29
4.4 Accessor Functions . . . 30
4.5 Modification Functions . . . 32
4.6 Spatial Reference System Functions . . . 32
4.7 Aggregate Functions . . . 33
4.8 Comparison Operators . . . 33
4.9 Set Operators . . . 34
4.10 Topological Operators . . . 35
4.11 Relative Position Operators . . . 36
4.12 Indexing of Box Types . . . 39

5 Manipulating Temporal Types 40
5.1 Input/Output of Temporal Types . . . 41
5.2 Constructor Functions . . . 43
5.3 Casting . . . 45
5.4 Accessor Functions . . . 47
5.5 Modification Functions . . . 54
5.6 Restriction Functions . . . 56
5.6.1 Selection Functions . . . 56
5.6.2 Difference Functions . . . 59
5.7 Comparison Operators . . . 61
5.7.1 Traditional Comparison Operators . . . 61
5.7.2 Ever and Always Comparison Operators . . . 63
5.7.3 Temporal Comparison Operators . . . 65
5.8 Bounding Box Operators . . . 66
5.9 Mathematical Functions and Operators . . . 66
5.10 Boolean Operators . . . 68
5.11 Text Functions and Operators . . . 68
5.12 Spatial Functions and Operators . . . 69
5.12.1 Input/Output Functions . . . 69
5.12.2 Spatial Reference System Functions . . . 72
5.12.3 Accessor Functions . . . 73

5.12.4 Manipulation Functions . . . 75
5.12.5 Distance Functions and Operators . . . 77
5.12.6 Spatial Relationships . . . 80
5.12.7 Ever Spatial Relationships . . . 81
5.12.8 Temporal Spatial Relationships . . . 82
5.13 Similarity Functions . . . 83
5.14 Multidimensional Tiling . . . 85
5.14.1 Bucket Functions . . . 86
5.14.2 Grid Functions . . . 87
5.14.3 Split Functions . . . 89
5.15 Aggregate Functions . . . 91
5.16 Utility Functions . . . 94
5.17 Indexing of Temporal Types . . . 94
5.18 Statistics and Selectivity for Temporal Types . . . 96
5.18.1 Statistics Collection . . . 96
5.18.2 Selectivity Estimation of Operators . . . 97

6 Temporal Network Points 98
6.1 Static Network Types . . . 98
6.1.1 Constructor Functions . . . 100
6.1.2 Modification Functions . . . 100
6.1.3 Accessor Functions . . . 100
6.1.4 Spatial Functions . . . 101
6.1.5 Comparison Operators . . . 102
6.2 Temporal Network Points . . . 103
6.3 Validity of Temporal Network Points . . . 104
6.4 Constructors for Temporal Network Points . . . 104
6.5 Casting for Temporal Network Points . . . 105
6.6 Functions and Operators for Temporal Network Points . . . 105
6.7 Aggregate Functions . . . 111
6.8 Indexing of Temporal Network Points . . . 112

A MobilityDB Reference 113
A.1 Functions and Operators for Time Types and Range Types . . . 113
A.1.1 Constructor Functions . . . 113
A.1.2 Casting . . . 113
A.1.3 Accessor Functions . . . 113
A.1.4 Modification Functions . . . 114
A.1.5 Comparison Operators . . . 114

A.1.6 Set Operators . . . 114
A.1.7 Topological and Relative Position Operators . . . 114
A.1.8 Distance Operators . . . 115
A.1.9 Aggregate Functions . . . 115
A.2 Functions and Operators for Box Types . . . 115
A.2.1 Constructor Functions . . . 115
A.2.2 Casting . . . 115
A.2.3 Accessor Functions . . . 115
A.2.4 Modification Functions . . . 116
A.2.5 Spatial Reference System Functions . . . 116
A.2.6 Aggregate Functions . . . 116
A.2.7 Comparison Operators . . . 116
A.2.8 Set Operators . . . 116
A.2.9 Topological Operators . . . 117
A.2.10 Relative Position Operators . . . 117
A.3 Functions and Operators for Temporal Types . . . 117
A.3.1 Constructor Functions . . . 117
A.3.2 Casting . . . 118
A.3.3 Accessor Functions . . . 118
A.3.4 Modification Functions . . . 119
A.3.5 Restriction Functions . . . 119
A.3.5.1 Selection Functions . . . 119
A.3.5.2 Difference Functions . . . 120
A.3.6 Comparison Operators . . . 120
A.3.6.1 Traditional Comparison Operators . . . 120
A.3.6.2 Ever and Always Comparison Operators . . . 121
A.3.6.3 Temporal Comparison Operators . . . 121
A.3.7 Mathematical Functions and Operators . . . 121
A.3.8 Boolean Operators . . . 121
A.3.9 Text Functions and Operators . . . 122
A.3.10 Spatial Functions and Operators . . . 122
A.3.10.1 Input/Output Functions . . . 122
A.3.10.2 Spatial Reference System Functions . . . 122
A.3.10.3 Accessor Functions . . . 123
A.3.10.4 Manipulation Functions . . . 123
A.3.10.5 Distance Functions and Operators . . . 123
A.3.10.6 Possible Spatial Relationships . . . 123
A.3.10.7 Temporal Spatial Relationships . . . 124
A.3.11 Similarity Functions . . . 124

A.3.12 Multidimensional Tiling . . . 124
A.3.12.1 Bucket Functions . . . 124
A.3.12.2 Grid Functions . . . 124
A.3.12.3 Split Functions . . . 124
A.3.13 Aggregate Functions . . . 125
A.3.14 Utility Functions . . . 125
A.4 Functions and Operators for Temporal Network Points . . . 125
A.4.1 Static Network Types . . . 125
A.4.1.1 Constructor Functions . . . 125
A.4.1.2 Accessor Functions . . . 125
A.4.1.3 Modification Functions . . . 126
A.4.1.4 Spatial Functions . . . 126
A.4.1.5 Comparison Operators . . . 126
A.4.2 Temporal Network Points . . . 126
A.4.2.1 Constructors . . . 126
A.4.2.2 Casting . . . 126
A.4.2.3 Functions and Operators . . . 126
A.4.2.4 Aggregate Functions . . . 127

B Synthetic Data Generator **128**
B.1 Generator for PostgreSQL Types . . . 128
B.2 Generator for PostGIS Types . . . 129
B.3 Generator for MobilityDB Time and Box Types . . . 130
B.4 Generator for MobilityDB Temporal Types . . . 130
B.5 Generation of Tables with Random Values . . . 131
B.6 Generator for Temporal Network Point Types . . . 137

7 Index **138**

List of Figures

5.1 Visualizing the speed of a moving object using a color ramp in QGIS. 77
5.2 Multidimensional tiling for temporal floats. 85

List of Tables

1.1 Variables for the user's and the developer's documentation 6

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).

This is the manual for MobilityDB v1.0. The MobilityDB Manual is licensed under a Creative Commons Attribution-Share Alike 3.0 License 3. Feel free to use this material any way you like, but we ask that you attribute credit to the MobilityDB Project and wherever possible, a link back to MobilityDB.

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, co-chair of the OGC Moving Feature Standard Working Group (MF-SWG)
- Vicky Vergara: integration with pgRouting, liason with OSGeo
- 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 Sponsors

1.3.1 Research Sponsors

These are research funding organizations (in alphabetical order) that have contributed with monetary funding to the MobilityDB project.

- European Commission
- Fonds de la Recherche Scientifique (FNRS), Belgium
- Innoviris, Belgium

1.3.2 Corporate Sponsors

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

- Adonmo, India
- Georepublic, Germany
- Université libre de Bruxelles, Belgium

1.4 Licenses

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

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

```
git clone https://github.com/MobilityDB/MobilityDB
mkdir MobilityDB/build
cd MobilityDB/build
cmake  ..
make
```

```
sudo make install
```

The above commands install the master branch. If you want to install another branch, for example, develop, you can replace the first command above as follows

```
git clone --branch develop https://github.com/MobilityDB/MobilityDB
```

You should also set the following in the postgresql.conf file depending on the version of PostGIS you have installed (below we use PostGIS 3):

```
shared_preload_libraries = 'postgis-3'
max_locks_per_transaction = 128
```

You can replace postgis-2.5 above if you want to use PostGIS 2.5.

If you do not preload the PostGIS library you will not be able to load the MobilityDB library and will get an error message such as the following one:

```
ERROR:  could not load library "/usr/local/pgsql/lib/libMobilityDB-1.0.so":
  undefined symbol: ST_Distance
```

Notice that you can find the location of the postgresql.conf file as given next.

```
$ which postgres
/usr/local/pgsql/bin/postgres
$ ls /usr/local/pgsql/data/postgresql.conf
/usr/local/pgsql/data/postgresql.conf
```

As can be seen, the PostgreSQL binaries are in the bin subdirectory while the postgresql.conf file is in the data subdirectory.

Once MobilityDB is installed, it needs to be enabled in each database you want to use it in. In the example below we use a database named mobility.

```
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 11 or higher. PostgreSQL is available from http://www.postgresql.org.
- PostGIS version 2.5 or higher. PostGIS is available from https://postgis.net/.
- 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 the user's documentation

- The DocBook DTD and XSL files are required for building the documentation. For Ubuntu, they are provided by the packages `docbook` and `docbook-xsl`.
- The XML validator `xmllint` is required for validating the XML files of the documentation. For Ubuntu, it is provided by the package `libxml2`.
- The XSLT processor `xsltproc` is required for building the documentation in HTML format. For Ubuntu, it is provided by the package `libxslt`.

- The program dblatex is required for building the documentation in PDF format. For Ubuntu, it is provided by the package dblatex.
- The program dbtoepub is required for building the documentation in EPUB format. For Ubuntu, it is provided by the package dbtoepub.

For the developers's documentation

- The program doxygen is required for building the documentation. For Ubuntu, it is provided by the package doxygen.

Example: Installing dependencies on Linux

Database dependencies

```
sudo apt-get install postgresql-13 postgresql-server-dev-13 postgresql-13-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_*'
```

1.5.8 Documentation

MobilityDB user's documentation can be generated in HTML, PDF, and EPUB format. Furthermore, the documentation is available in English and in other languages (currently, only in Spanish). The user's documentation can be generated in all formats and in all languages, or specific formats and/or languages can be specified. MobilityDB developer's documentation can only be generated in HTML format and in English.

The variables used for generating user's and the developer's documentation are as follows:

Variable	Default value	Comment
DOC_ALL	BOOL=OFF	The user's documentation is generated in HTML, PDF, and EPUB formats.
DOC_HTML	BOOL=OFF	The user's documentation is generated in HTML format.
DOC_PDF	BOOL=OFF	The user's documentation is generated in PDF format.
DOC_EPUB	BOOL=OFF	The user's documentation is generated in EPUB format.
LANG_ALL	BOOL=OFF	The user's documentation is generated in English and in all available translations.
ES	BOOL=OFF	The user's documentation is generated in English and in Spanish.
DOC_DEV	BOOL=OFF	The English developer's documentation is generated in HTML format.

Table 1.1: Variables for the user's and the developer's documentation

Generate the user's and the developer's documentation in all formats and in all languages.

```
cmake -D DOC_ALL=ON -D LANG_ALL=ON -D DOC_DEV=ON ..
make doc
make doc_dev
```

Generate the user's documentation in HTML format and in all languages.

```
cmake -D DOC_HTML=ON -D LANG_ALL=ON ..
make doc
```

Generate the English user's documentation in all formats.

```
cmake -D DOC_ALL=ON ..
make doc
```

Generate the user's documentation in PDF format in English and in Spanish.

```
cmake -D DOC_PDF=ON -D ES=ON ..
make doc
```

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 and range 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, `integer` 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: boolean
```

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}): integer`

```
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},integer): 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): integer`

```
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,integer): 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)"}"
```

2.1.4 Modification Functions

- 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]}"
```

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

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

```
SELECT tscale(timestampset '{2001-01-01}', '1 day');
-- {2001-01-01}
SELECT tscale(timestampset '{2001-01-01, 2001-01-03, 2001-01-05}', '2 days');
-- {2001-01-01, 2001-01-02, 2001-01-03}
SELECT tscale(period '[2001-01-01, 2001-01-03]', '1 day');
-- [2001-01-01, 2001-01-02]
SELECT tscale(periodset '{[2001-01-01, 2001-01-03], [2001-01-04, 2001-01-05]}', '1 day');
-- {[2001-01-01 00:00:00, 2001-01-01 12:00:00],
  [2001-01-01 18:00:00, 2001-01-02 00:00:00]}
SELECT tscale(timestampset '{2001-01-01}', '-1 day');
-- ERROR:  The duration must be a positive interval: -1 days
```

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

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

```
SELECT shiftTscale(timestampset '{2001-01-01}', '1 day', '1 day');
-- {2001-01-02}
SELECT shiftTscale(timestampset '{2001-01-01, 2001-01-03, 2001-01-05}', '1 day','2 days');
-- {2001-01-02, 2001-01-03, 2001-01-04}
SELECT shiftTscale(period '[2001-01-01, 2001-01-03]', '1 day', '1 day');
-- [2001-01-02, 2001-01-03]
SELECT shiftTscale(periodset '{[2001-01-01, 2001-01-03], [2001-01-04, 2001-01-05]}',
  '1 day', '1 day');
-- {[2001-01-02 00:00:00, 2001-01-02 12:00:00],
  [2001-01-02 18:00:00, 2001-01-03 00:00:00]}
```

- Round the bounds of a float range to a number of decimal places

  `round(floatrange,integer): floatrange`

```
SELECT round(floatrange '[1.123456789,2.123456789)', 3);
-- "[1.123,2.123)"
SELECT round(floatrange '(,2.123456789]', 3);
-- "(,2.123]"
SELECT round(floatrange '[1.123456789, inf)', 3);
-- "[1.123,Infinity)"
```

2.1.5 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: boolean`

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

- Are the time values different?

  `time <> time: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

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

2.1.6 Set Operators

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

- Union of the time values

  `time + 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: 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: 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]}"
```

2.1.7 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}: boolean`

```
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: boolean`

```
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}: boolean`

```
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: boolean`

```
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
```

2.1.8 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}: boolean`

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

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

  `{number,range} >> {number,range}: boolean`

```
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}: boolean`

```
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}: boolean`

```
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}: boolean`

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

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

  `time <<# time: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

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

2.1.9 Distance Operators

There are two distance operators for time types. These operators work with bouding periods and compute both the smallest distance between the two time values but they differ in the result type: the `<->` operator returns a standard SQL type `interval`, while the `|=|` operator returns a `float` which is the number of seconds between the two time values. The latter operator can also be used for nearest neighbor searches using a GiST or an SP-GiST index (see Section 2.2).

- Get the smallest distance ever in an interval

  `time <-> time: interval`

```
SELECT period '[2012-01-02, 2012-01-06)' <-> timestamptz '2012-01-07';
-- 1 day
SELECT timestampset '{2012-01-01, 2012-01-03, 2012-01-05}' <->
  timestampset '{2012-01-02, 2012-01-04}';
--  00:00:00
```

- Get the smallest distance ever in number of seconds

  `time |=| time: float`

```
SELECT period '[2012-01-02, 2012-01-06)' |=| timestamptz '2012-01-07';
-- 86400
SELECT timestampset '{2012-01-01, 2012-01-03, 2012-01-05}' |=|
  timestampset '{2012-01-02, 2012-01-04}';
--  0
```

2.1.10 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.

Similarly, there is an aggregate function for range types:

- Function `extent` returns a bounding range that encloses a set of integer or float range values.

Union is a very useful operation for time types. As we have seen in Section 2.1.6, 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}): {tint_instset,tint_seqset}`

```
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]"
```

- Bounding range

  `extent(range): range`

```
WITH ranges(r) AS (
  SELECT floatrange '[1, 4)' UNION
  SELECT floatrange '(5, 8)' UNION
  SELECT floatrange '(7, 9)'
)
SELECT extent(r) FROM ranges;
-- "[1,9)"
```

- 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. The GiST index implements an R-tree while the SP-GiST index implements a quad-tree. 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 an 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

We present next the functions and operators for bounding box types. These functions and operators are polimorhic, that is, their argumentos can be of various types and their result type may depend on the type of the arguments. To express this in the signature of the operators, we use the following notation:

- `box` represents any bounding box type, that is, `tbox` or `stbox`.

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.

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): stbox`

  `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`

  `{integer,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,geometry}`

```
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)"
SELECT ST_AsEWKT(stbox 'SRID=4326;STBOX T((1,1,2000-01-01),(5,5,2000-01-05))'::geometry);
-- SRID=4326;POLYGON((1 1,1 5,5 5,5 1,1 1))
SELECT ST_AsEWKT(stbox 'SRID=4326;STBOX T((1,1,2000-01-01),(1,5,2000-01-05))'::geometry);
-- SRID=4326;LINESTRING(1 1,1 5)
SELECT ST_AsEWKT(stbox 'SRID=4326;STBOX T((1,1,2000-01-01),(1,1,2000-01-05))'::geometry);
-- SRID=4326;POINT(1 1)
SELECT ST_AsEWKT(stbox 'SRID=4326;STBOX ZT((1,1,1,2000-01-01),(5,5,5,2000-01-05))'::
  geometry);
-- SRID=4326;POLYHEDRALSURFACE(((1 1 1,1 5 1,5 5 1,5 1 1,1 1 1)),
  ((1 1 5,5 1 5,5 5 5,1 5 5,1 1 5)),((1 1 1,1 1 5,1 5 5,1 5 1,1 1 1)),
```

```
  ((5 1 1,5 5 1,5 5 5,5 1 5,5 1 1)),((1 1 1,5 1 1,5 1 5,1 1 5,1 1 1)),
  ((1 5 1,1 5 5,5 5 5,5 5 1,1 5 1)))
```

- 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(box): boolean`

```
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): boolean`

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

- Has T dimension?

  `hasT(box): boolean`

```
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(box): 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(box): 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(box): timestamptz`

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

- Get the maximum T value

  `Tmax(box): 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(box,interval): box`

```
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

  `round(box,integer): box`

```
SELECT round(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 round(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): integer`

```
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 Aggregate Functions

- Bounding box extent

  `extent(box): box`

```
WITH boxes(b) AS (
  SELECT tbox 'TBOX((1, 2000-01-01),(3, 2000-01-03))' UNION
  SELECT tbox 'TBOX((5, 2000-01-05),(7, 2000-01-07))' UNION
  SELECT tbox 'TBOX((6, 2000-01-06),(8, 2000-01-08))'
)
SELECT extent(b) FROM boxes;
-- TBOX((1,2000-01-01 00:00:00+01),(8,2000-01-08 00:00:00+01))
WITH boxes(b) AS (
  SELECT stbox 'STBOX Z((1, 1, 1),(3, 3, 3))' UNION
  SELECT stbox 'STBOX Z((5, 5, 5),(7, 7, 7))' UNION
  SELECT stbox 'STBOX Z((6, 6, 6),(8, 8, 8))'
)
SELECT extent(b) FROM boxes;
-- STBOX Z((1,1,1),(8,8,8))
```

4.8 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?

  `box = box: 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?

  `box <> box: 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?

  `box < box: 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?

  `box > box: 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?

  `box <= box: 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?

  `box >= box: boolean`

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

4.9 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 raised. 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

  `box + box: box`

```
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

  `box * box:  box`

```
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))"
```

4.10 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?

  `box && box:  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?

  `box @> box:  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?

  `box <@ box:  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?

  `box ~= box: 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?

  `box -|- box: 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
```

4.11 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?

  `box <<# box: 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?

  `box #>> box: 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?

  `box &<# box: 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?

  `box #&> box: 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
```

4.12 Indexing of Box Types

GiST and SP-GiST indexes can be created for table columns of the `tbox` and `stbox` types. The GiST index implements an R-tree and the SP-GiST index implements an n-dimensional quad-tree. 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`,
- `ttype_inst` represents any temporal type with instant subtype,
- `ttype_instset` represents any temporal type with instant set subtype,
- `ttype_seq` represents any temporal type with sequence subtype,
- `tdisc_seq` represents any temporal type with sequence subtype and a discrete base type,
- `tcont_seq` represents any temporal type with sequence subtype and a continuous base type,
- `ttype_seqset` represents any temporal type with sequence set subtype,
- `base` represents any base type of a temporal type, that is, `boolean`, `integer`, `float`, `text`, `geometry`, or `geography`,
- `number` represents any number base type, that is, `integer` or `float`,
- `numrange` represents any number range type, that is, either `intrange` or `floatrange`,
- `geo` represents the types `geometry` or `geography`,
- `geompoint` represents the type `geometry` restricted to a point.
- `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, 0.745184033794557@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 a constructor function with the same name as the type and a suffix for the subtype, where the suffixes '`_inst`', '`_instset`', '`_seq`', and '`_seqset`' correspond, respectively, to the subtypes instant, instant set, sequence, and sequence set. Examples are `tint_seq` or `tgeompoint_seqset`. 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.
- Finally, another set of functions for sequence set values have as first argument an array of values of the corresponding instant values, and two arguments stating a maximum distance and a maximum time interval such that a gap is introduced between composing sequences of the result whenever two consecutive input instants have a distance or a time gap greater than these values. For temporal points the distance is specified in units of the underlying SRID. These two gaps arguments are optional and if they are not given, a zero value is assumed, which is not taken into account for determining the gaps. 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.

We give next the constructor functions for the various subtypes.

- Constructor for temporal types of instant subtype

  `ttype_inst(base,timestamptz): ttype_inst`

```
SELECT tbool_inst(true, '2001-01-01');
SELECT tint_inst(1, '2001-01-01');
```

- Constructor for temporal types of instant set subtype

  `ttype_instset(base,timestampset): ttype_instset`

  `ttype_instset(ttype_inst[]): ttype_instset`

```
SELECT tfloat_instset(1.5, '{2001-01-01, 2001-01-02}');
SELECT ttext_instset('AAA', '{2001-01-01, 2001-01-02}');
SELECT tbool_instset(ARRAY[tbool 'true@2001-01-01 08:00:00','false@2001-01-01 08:05:00']);
SELECT tint_instset(ARRAY[tint '1@2001-01-01 08:00:00', '2@2001-01-01 08:05:00']);
SELECT tfloat_instset(ARRAY[tfloat '1.0@2001-01-01 08:00:00', '2.0@2001-01-01 08:05:00']);
SELECT ttext_instset(ARRAY[ttext 'AAA@2001-01-01 08:00:00', 'BBB@2001-01-01 08:05:00']);
SELECT tgeompoint_instset(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 tgeogpoint_instset(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

  `tdisc_seq(base,period): tdisc_seq`

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

  `tdisc_seq(ttype_inst[],left_inc=true,right_inc=true}): tdisc_seq`

  `tcont_seq(ttype_inst[],left_inc=true,right_inc=true,linear=true}): tcont_seq`

```
SELECT tfloat_seq(1.5, '[2001-01-01, 2001-01-02]');
SELECT tgeompoint_seq('Point(0 0)', '[2001-01-01, 2001-01-02]', false);
SELECT tbool_seq(ARRAY[tbool 'true@2001-01-01 08:00:00', 'true@2001-01-03 08:05:00'],
  true, true);
SELECT tint_seq(ARRAY[tint(2,'2001-01-01 08:00:00'), tint(2,'2001-01-01 08:10:00')],
  true, false);
SELECT tfloat_seq(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 tfloat_seq(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 ttext_seq(ARRAY[ttext('AAA', '2001-01-01 08:00:00'),
  ttext('BBB', '2001-01-03 08:05:00'), ttext('BBB', '2001-01-03 08:10:00')]);
SELECT tgeompoint_seq(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 tgeogpoint_seq(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

  `tdisc_seqset(base,periodset): tdisc_seqset`

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

  `ttype_seqset(ttype_seq[]): ttype_seqset`

  `tdisc_seqset_gaps(ttype_inst[],maxdist=0.0,maxt='0 minutes'): tdisc_seqset`

  `tcont_seqset_gaps(ttype_inst[],linear=true,maxdist=0.0,maxt='0 minutes'): tcont_seqset`

```
SELECT ttext_seqset('AAA', '{[2001-01-01, 2001-01-02], [2001-01-03, 2001-01-04]}');
SELECT tgeogpoint_seqset('Point(0 0)', '{[2001-01-01, 2001-01-02],
  [2001-01-03, 2001-01-04]}', false);
SELECT tbool_seqset(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 tint_seqset(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 tfloat_seqset(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 tfloat_seqset(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 ttext_seqset(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 tgeompoint_seqset(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 tgeogpoint_seqset(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 tfloat_seqset(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
SELECT tint_seqset_gaps(ARRAY[tint '1@2000-01-01', '3@2000-01-02', '4@2000-01-03',
  '5@2000-01-05'], 1, '1 day');
-- {[1@2000-01-01], [3@2000-01-02, 4@2000-01-03], [5@2000-01-05]}
SELECT asText(tgeompoint_seqset_gaps(ARRAY[tgeompoint 'Point(1 1)@2000-01-01',
  'Point(2 2)@2000-01-02', 'Point(3 2)@2000-01-03', 'Point(3 2)@2000-01-05'],
  true, 1, '1 day'));
-- {[POINT(1 1)@2000-01-01], [POINT(2 2)@2000-01-02, POINT(3 2)@2000-01-03],
  [POINT(3 2)@2000-01-05]}
```

5.3 Casting

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

- Cast a temporal value to a period

  `ttype::period`

```
SELECT tint '[1@2001-01-01, 2@2001-01-03]'::period;
-- "[2001-01-01, 2001-01-03]"
SELECT ttext '(A@2000-01-01, B@2000-01-03, C@2000-01-05]'::period;
-- "(2000-01-01, 2000-01-05]"
```

- Cast a temporal number to a range

  `tnumber::range`

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

- Cast a temporal number to a `tbox`

  `tnumber::tbox`

```
SELECT tint '[1@2001-01-01, 2@2001-01-03]'::tbox;
-- "TBOX((1,2001-01-01 00:00:00+01),(2,2001-01-03 00:00:00+01))"
SELECT tfloat '(1@2000-01-01, 3@2000-01-03, 2@2000-01-05]'::tbox;
-- "TBOX((1,2000-01-01 00:00:00+01),(3,2000-01-05 00:00:00+01))"
```

- Cast a temporal point to an `stbox`

  `tpoint::stbox`

```
SELECT tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03]'::stbox;
-- STBOX T((1,1,2001-01-01),(3,3,2001-01-03))
SELECT SELECT tgeogpoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03]'::stbox;;
-- "SRID=4326;GEODSTBOX T
  ((0.9972609281539917,0.017449747771024704,0.01745240643728351,2001-01-01),
  (0.9996954202651978,0.05226423218846321,0.05233595624294383,2001-01-03))"
```

- 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, 2@2001-01-03]"
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, 2@2001-01-03]"
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 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(ttype_inst): 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 instant with the minimum value

  `minInstant(torder): base`

  The function does not take into account whether the bounds are inclusive or not. If several instants have the minimum value, the first one is returned.

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

- Get the instant with the maximum value

  `maxInstant(torder): base`

  The function does not take into account whether the bounds are inclusive or not. If several instants have the maximum value, the first one is returned.

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

- 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(ttype_inst): 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(tfloat '[1@2012-01-01, 2@2012-01-03, 2@2012-01-05)');
-- "4 days"
SELECT duration(tfloat '{[1@2012-01-01, 2@2012-01-03), [2@2012-01-04, 2@2012-01-05)}');
-- "3 days"
```

- Get the timespan ignoring the potential time gaps

  `timespan(ttype): interval`

```
SELECT timespan(tfloat '{1@2012-01-01, 2@2012-01-03, 2@2012-01-05}');
-- "4 days"
SELECT timespan(tfloat '{[1@2012-01-01, 2@2012-01-03), [2@2012-01-04, 2@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): integer`

```
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): ttype_inst`

  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): ttype_inst`

  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,integer): ttype_inst`

```
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): ttype_inst[]`

```
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): integer`

```
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,integer): 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({ttype_seq,ttype_seqset}): integer`

```
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({ttype_seq,ttype_seqset}): ttype_seq`

```
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({ttype_seq,ttype_seqset}): ttype_seq`

```
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({ttype_seq,ttype_seqset},integer): ttype_seq`

```
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({ttype_seq,ttype_seqset}): ttype_seq[]`

```
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({ttype_seq,ttype_seqset}): ttype_seq[]`

```
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]"}
```

- 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
```

5.5 Modification 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

  `ttype_inst(ttype): ttype_inst`

  `ttype_instset(ttype): ttype_instset`

  `ttype_seq(ttype): ttype_seq`

  `ttype_seqset(ttype): ttype_seqset`

```
SELECT tbool_inst(tbool '{[true@2001-01-01]}');
-- "t@2001-01-01 00:00:00+00"
SELECT tbool_inst(tbool '{[true@2001-01-01, true@2001-01-02]}');
-- ERROR: Cannot transform input to a temporal instant
SELECT tbool_instset(tbool 'true@2001-01-01');
-- "{t@2001-01-01}"
SELECT tint_seq(tint '1@2001-01-01');
-- "[1@2001-01-01]"
SELECT tfloat_seqset(tfloat '2.5@2001-01-01');
-- "{[2.5@2001-01-01]}"
SELECT tfloat_seqset(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,ttype_inst): ttype`

```
SELECT appendInstant(tint '1@2000-01-01', tint '1@2000-01-02');
-- "{1@2000-01-01, 1@2000-01-02}"
SELECT appendInstant(tint_seq(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]"
```

- 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]}"
```

5.6 Restriction Functions

5.6.1 Selection 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): ttype_inst`

```
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): {ttype_inst,ttype_instset}`

```
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]}"
```

5.6.2 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)}"
```

5.7 Comparison Operators

5.7.1 Traditional 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 2.1.5), then the bounding boxes (see Section 4.8) 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
```

5.7.2 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: boolean`

  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: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

  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: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

```
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: boolean`

```
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)}"
```

5.7.3 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)}"
```

5.8 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 4.10 and Section 4.11 for the bounding box operators.

5.9 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, 0.666666666666667@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 '[0.785398163397448@2000-01-01, 2.35619449019234@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 '[0.785398163397448@2000-01-01, 2.35619449019234@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
```

5.10 Boolean Operators

- Temporal and

  `{boolean,tbool} & {boolean,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

  `{boolean,tbool} | {boolean,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)"
```

5.11 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]"
```

5.12 Spatial Functions and Operators

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.

5.12.1 Input/Output Functions

- 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 integer=15,options integer=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,endian 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 'Point(1 2 3)@2012-01-01');
-- "\x0191000000000000f03f0000000000000040000000000000084000fce0136a580100"
```

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

  `asEWKB(tpoint): bytea`

  `asEWKB(tpoint,endian 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(tgeogpoint 'SRID=7844;Point(1 2 3)@2012-01-01');
-- "\x01f1a41e0000000000000000f03f0000000000000040000000000000084000fce0136a580100"
```

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

  `asHexEWKB(tpoint): text`

  `asHexEWKB(tpoint,endian 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=3812;Point(1 2 3)@2012-01-01');
-- "01D1E40E0000000000000000F03F0000000000000040000000000000084000FCE0136A580100"
```

- Input a temporal geometry point from a Well-Known Text (WKT) representation

  `tgeompointFromText(text): tgeompoint`

```
SELECT asEWKT(tgeompointFromText(text '[POINT(1 2)@2000-01-01, POINT(3 4)@2000-01-02]'));
-- "[POINT(1 2)@2000-01-01, POINT(3 4)@2000-01-02]"
```

- Input a temporal geography point from a Well-Known Text (WKT) representation

  `tgeogpointFromText(text): tgeogpoint`

```
SELECT asEWKT(tgeogpointFromText(text '[POINT(1 2)@2000-01-01, POINT(3 4)@2000-01-02]'));
-- "SRID=4326;[POINT(1 2)@2000-01-01, POINT(3 4)@2000-01-02]"
```

- Input a temporal geometry point from an Extended Well-Known Text (EWKT) representation

  `tgeompointFromEWKT(text): tgeompoint`

```
SELECT asEWKT(tgeompointFromEWKT(text 'SRID=3812;[POINT(1 2)@2000-01-01,
  POINT(3 4)@2000-01-02]'));
-- "SRID=3812;[POINT(1 2)@2000-01-01 00:00:00+01, POINT(3 4)@2000-01-02 00:00:00+01]"
```

- Input a temporal geography point from an Extended Well-Known Text (EWKT) representation

  `tgeogpointFromEWKT(text): tgeogpoint`

```
SELECT asEWKT(tgeogpointFromEWKT(text 'SRID=7844;[POINT(1 2)@2000-01-01,
  POINT(3 4)@2000-01-02]'));
-- "SRID=7844;[POINT(1 2)@2000-01-01, POINT(3 4)@2000-01-02]"
```

- Input a temporal geometry point from a Moving Features JSON representation

  `tgeompointFromMFJSON(text): tgeompoint`

```
SELECT asEWKT(tgeompointFromMFJSON(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 geography point from a Moving Features JSON representation

  `tgeogpointFromMFJSON(text): tgeogpoint`

```
SELECT asEWKT(tgeogpointFromMFJSON(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 geometry point from a Well-Known Binary (WKB) representation

  `tgeompointFromBinary(bytea): tgeompoint`

```
SELECT asEWKT(tgeompointFromBinary(
  '\x0181000000000000f03f0000000000000040005c6c29ffffffff'));
-- "POINT(1 2)@2000-01-01"
```

- Input a temporal geography point from a Well-Known Binary (WKB) representation

  `tgeogpointFromBinary(bytea): tgeogpoint`

```
SELECT asEWKT(tgeompointFromBinary(
  '\x01b1000000000000f03f000000000000f03f000000000000f03f005c6c29ffffffff'));
-- "SRID=4326;POINT Z (1 1 1)@2000-01-01"
```

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

  `tgeompointFromEWKB(bytea): tgeompoint`

```
SELECT asEWKT(tgeompointFromEWKB(
  '\x01c1e40e0000000000000000f03f0000000000000040005c6c29ffffffff'));
-- "SRID=3812;POINT(1 2)@2000-01-01"
```

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

  `tgeogpointFromEWKB(bytea): tgeogpoint`

```
SELECT asEWKT(tgeogpointFromEWKB(
  '\x01f1a41e0000000000000000f03f000000000000f03f000000000000f03f005c6c29ffffffff'));
-- "SRID=7844;POINT Z (1 1 1)@2000-01-01"
```

- Input a temporal geometry point from an Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `tgeompointFromHexEWKB(text): tgeompoint`

```
SELECT asEWKT(tgeompointFromHexEWKB(
  '01C1E40E0000000000000000F03F0000000000000040005C6C29FFFFFFFF'));
-- "SRID=3812;POINT(1 2)@2000-01-01"
```

- Input a temporal geography point from an Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `tgeogpointFromHexEWKB(text): tgeogpoint`

```
SELECT asEWKT(tgeogpointFromHexEWKB(
  '01F1A41E0000000000000000F03F000000000000F03F000000000000F03F005C6C29FFFFFFFF'));
-- "SRID=7844;POINT Z (1 1 1)@2000-01-01"
```

5.12.2 Spatial Reference System Functions

- 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"
```

5.12.3 Accessor Functions

- 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): boolean`

  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
```

- 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): tfloat_seq`

```
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): tfloat_seqset`

  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 temporal bearing

  `bearing({tpoint, point}, {tpoint, point}): tfloat`

```
SELECT degrees(bearing(tgeompoint '[Point(1 1)@2012-01-01, Point(3 3)@2012-01-03]',
  geometry 'Point(2 2)'));
-- [45@2012-01-01, 0@2012-01-02, 225@2012-01-03]
SELECT round(degrees(bearing(tgeompoint '[Point(0 0)@2012-01-01, Point(2 0)@2012-01-03]',
  tgeompoint '[Point(2 1)@2012-01-01, Point(0 1)@2012-01-03]')), 3);
--  [63.435@2012-01-01, 0@2012-01-02, 296.565@2012-01-03]
SELECT round(degrees(bearing(tgeompoint '[Point(2 1)@2012-01-01, Point(0 1)@2012-01-03]',
  tgeompoint '[Point(0 0)@2012-01-01, Point(2 0)@2012-01-03]')), 3);
-- [243.435@2012-01-01, 116.565@2012-01-03]
```

Please notice that this function currently does not accept two temporal geographic points.

5.12.4 Manipulation Functions

- Round the coordinate values to a number of decimal places

  `round(tpoint,integer): tpoint`

```
SELECT asText(round(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(round(tgeogpoint 'Point(1.12345 1.12345)@2000-01-01', 2));
-- "POINT(1.12 1.12)@2000-01-01"
```

- 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]}"}
```

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

  `simplify(tpoint,distance float): tpoint`

  `simplify(tpoint,distance float,speed 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)"

-- Only speed difference specified
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)
)
```

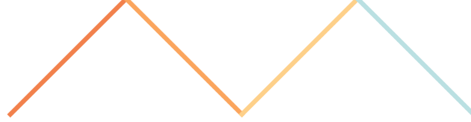

Figure 5.1: Visualizing the speed of a moving object using a color ramp in QGIS.

- Transform a temporal geometric point into the coordinate space of a Mapbox Vector Tile. The result is a couple composed of a `geometry` value and an array of associated timestamp values encoded as Unix epoch

  `asMVTGeom(tpoint,bounds,extent=4096,buffer=256,clip=TRUE): geom_times`

  The parameters are as follows:

  – `tpoint` is the temporal point to transform
  – `bounds` is an `stbox` defining the geometric bounds of the tile contents without buffer
  – `extent` is the tile extent in tile coordinate space
  – `buffer` is the buffer distance in tile coordinate space
  – `clip` is a Boolean that determines if the resulting geometries and timestamps should be clipped or not

```
SELECT ST_AsText((mvt).geom), (mvt).times
FROM (SELECT asMVTGeom(tgeompoint '[Point(0 0)@2000-01-01, Point(100 100)@2000-01-02]',
  stbox 'STBOX((40,40),(60,60))') AS mvt ) AS t;
-- LINESTRING(-256 4352,4352 -256) | {946714680,946734120}
SELECT ST_AsText((mvt).geom), (mvt).times
FROM (SELECT asMVTGeom(tgeompoint '[Point(0 0)@2000-01-01, Point(100 100)@2000-01-02]',
  stbox 'STBOX((40,40),(60,60))', clip:=false) AS mvt ) AS t;
-- LINESTRING(-8192 12288,12288 -8192) | {946681200,946767600}
```

5.12.5 Distance Functions and Operators

- Get the smallest distance ever

  `{geo,tpoint} |=| {geo,tpoint}: 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
SELECT 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
SELECT 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
```

The operator `|=|` can be used for doing nearest neighbor searches using a GiST or an SP-GiST index (see Section 5.17). This operator corresponds to the PostGIS function `ST_DistanceCPA`, 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 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 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
```

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 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)"
```

5.12.6 Spatial Relationships

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

There are two versions of the spatial relationships:

- The *ever relationships* determine whether the topological or distance relationship is ever satisfied (see Section 5.7.2) and returns a `boolean`. Examples are the `intersects` and `dwithin` functions.
- The *temporal relationships* compute the topological or distance relationship at each instant and results in a `tbool`. Examples are the `tintersects` and `tdwithin` functions.

For example, the following query

```
SELECT intersects(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  tgeompoint '[Point(0 2)@2012-01-01, Point(4 2)@2012-01-05)');
-- t
```

tests whether the temporal point ever intersects the geometry. In this case, the query is equivalent to the following one

```
SELECT ST_Intersects(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  geometry 'Linestring(0 2,4 2)');
```

where the second geometry is obtained by applying the `trajectory` function to the temporal point. In contrast, the query

```
SELECT tintersects(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  tgeompoint '[Point(0 2)@2012-01-01, Point(4 2)@2012-01-05)');
-- {[f@2012-01-01, t@2012-01-02, t@2012-01-04], (f@2012-01-04, f@2012-01-05)}
```

computes at each instant whether the temporal point intersects the geometry. Similarly, the following query

```
SELECT dwithin(tgeompoint '[Point(3 1)@2012-01-01, Point(5 1)@2012-01-03)',
  tgeompoint '[Point(3 1)@2012-01-01, Point(1 1)@2012-01-03)', 2);
-- t
```

tests whether the distance between the temporal points was ever less than or equal to 2, while the following query

```
SELECT tdwithin(tgeompoint '[Point(3 1)@2012-01-01, Point(5 1)@2012-01-03)',
  tgeompoint '[Point(3 1)@2012-01-01, Point(1 1)@2012-01-03)', 2);
-- {[t@2012-01-01, t@2012-01-02], (f@2012-01-02, f@2012-01-03)}
```

computes at each instant whether the distance between the temporal points is less than or equal to 2.

The ever relationships are sometimes used in combination with a spatiotemporal 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 from a spatiotemporal 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.

Not all spatial relationships available in PostGIS have a meaningful generalization for temporal points. A generalized version of the following relationships are defined for temporal geometric points: `intersects`, `disjoint`, `dwithin`, `contains`, and `touches`, while for temporal geographic points only the three first ones are defined. Furthermore, not all combinations of parameters are meaningful for a given generalized function. For example, while `tcontains(geometry, tpoint)` is meaningful, `tcontains(tpoint, geometry)` is meaningful only when the geometry is a single point, and `tcontains(tpoint, tpoint)` is equivalent to `tintersects(tpoint, geometry)`. For this reason, only the first combination of parameters is defined for `contains` and `tcontains`.

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

5.12.7 Ever Spatial Relationships

All these functions automatically include a bounding box comparison that makes use of any spatial indexes that are available on the arguments.

- Ever contains

  `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
```

- Is ever disjoint

  `disjoint({geo,tpoint},{geo,tpoint}): 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
SELECT disjoint(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 0)@2012-01-03)');
-- true
```

- Is ever at distance within

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

```
SELECT dwithin(geometry 'Point(1 1 1)',
  tgeompoint '[Point(0 0 0)@2000-01-01, Point(1 1 0)@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
```

- Ever intersects

  `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
```

- Ever touches

  `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
```

5.12.8 Temporal Spatial Relationships

A common requirement regarding the temporal spatial relationships is to restrict the result of the relationship to the instants when the value of the result is true (alternatively, false). As an example, the following query computes for each trip the time spent traveling in the Brussels municipality.

```
SELECT TripId, duration(atValue(tintersects(T.trip, M.geom), True))
FROM Trips T, Municipality M
WHERE M.Name = "Brussels" AND atValue(tintersects(T.trip, M.geom), True) IS NOT NULL;
```

To simplify query writing, the temporal spatial relationships have an optional last parameter, which if given applies the `atValue` function (see Section 5.6) to the result of the relationship. In this way, the above query can be written as follows.

```
SELECT TripId, duration(tintersects(T.trip, M.geom, True))
FROM Trips T, Municipality M
WHERE M.Name = "Brussels" AND tintersects(T.trip, M.geom, True) IS NOT NULL;
```

- Temporal contains

  `tcontains(geometry, tgeompoint): tbool`

  `tcontains(geometry, tgeompoint,atValue boolean): 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 disjoint

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

  `tdisjoint({geo,tpoint},{geo,tpoint},atValue boolean): tbool`

  The function only supports 3D or geographies for two temporal points

```
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 distance within

  `tdwithin({geompoint,tgeompoint},{geompoint,tgeompoint},float): tbool`

  `tdwithin({geompoint,tgeompoint},{geompoint,tgeompoint},float,atValue boolean): tbool`

  The function only allows 3D for two temporal points

```
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 intersects

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

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

  The function only supports 3D or geographies for two temporal points

```
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`

  `ttouches({geo,tgeompoint},{geo,tgeompoint},atValue boolean): 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]}"
```

5.13 Similarity Functions

- Get the discrete Fréchet distance between two temporal values

  `frechetDistance({tnumber, tgeo}, {tnumber, tgeo}): float`

  This function has a linear space complexity since only two rows of the distance matrix are allocated in memory. Nevertheless, its time complexity is quadratic in the number of instants of the temporal values. Therefore, the function will require considerable time for temporal values with large number of instants.

```
SELECT frechetDistance(tfloat '[1@2012-01-01, 3@2012-01-03, 1@2012-01-06]',
 tfloat '[1@2012-01-01, 1.5@2012-01-02, 2.5@2012-01-03, 1.5@2012-01-04, 1.5@2012-01-05]');
-- 0.5
SELECT round(frechetDistance(tgeompoint '[Point(1 1)@2012-01-01, Point(3 3)@2012-01-03,
  Point(1 1)@2012-01-05]', tgeompoint '[Point(1.1 1.1)@2012-01-01,
  Point(2.5 2.5)@2012-01-02, Point(4 4)@2012-01-03, Point(3 3)@2012-01-04,
  Point(1.5 2)@2012-01-05]')::numeric, 6);
-- 1.414214
```

- Get the correspondences between two temporal values with respect to the discrete Fréchet distance {}

  `frechetDistancePath({tnumber, tgeo}, {tnumber, tgeo}): pairs`

  This function requires to allocate in memory a distance matrix whose size is quadratic in the number of instants of the temporal values. Therefore, the function will fail for temporal values with large number of instants depending on the available memory.

```
SELECT frechetDistancePath(tfloat '[1@2012-01-01, 3@2012-01-03, 1@2012-01-06]',
 tfloat '[1@2012-01-01, 1.5@2012-01-02, 2.5@2012-01-03, 1.5@2012-01-04, 1.5@2012-01-05]');
-- (0,0)
   (1,0)
   (2,1)
   (3,2)
   (4,2)
SELECT frechetDistancePath(tgeompoint '[Point(1 1)@2012-01-01, Point(3 3)@2012-01-03,
  Point(1 1)@2012-01-05]', tgeompoint '[Point(1.1 1.1)@2012-01-01,
  Point(2.5 2.5)@2012-01-02, Point(4 4)@2012-01-03, Point(3 3)@2012-01-04,
  Point(1.5 2)@2012-01-05]');
-- (0,0)
   (1,1)
   (2,1)
   (3,1)
   (4,2)
```

- Get the Dynamic Time Warp (DTW) distance between two temporal values

  `dynamicTimeWarp({tnumber, tgeo}, {tnumber, tgeo}): float`

  This function has a linear space complexity since only two rows of the distance matrix are allocated in memory. Nevertheless, its time complexity is quadratic in the number of instants of the temporal values. Therefore, the function will require considerable time for temporal values with large number of instants.

```
SELECT dynamicTimeWarp(tfloat '[1@2012-01-01, 3@2012-01-03, 1@2012-01-06]',
 tfloat '[1@2012-01-01, 1.5@2012-01-02, 2.5@2012-01-03, 1.5@2012-01-04, 1.5@2012-01-05]');
-- 2
SELECT round(dynamicTimeWarp(tgeompoint '[Point(1 1)@2012-01-01, Point(3 3)@2012-01-03,
  Point(1 1)@2012-01-05]', tgeompoint '[Point(1.1 1.1)@2012-01-01,
  Point(2.5 2.5)@2012-01-02, Point(4 4)@2012-01-03, Point(3 3)@2012-01-04,
  Point(1.5 2)@2012-01-05]')::numeric, 6);
-- 3.380776
```

- Get the correspondences between two temporal values with respect to the discrete Fréchet distance {}

  `dynamicTimeWarpPath({tnumber, tgeo}, {tnumber, tgeo}): pairs`

  This function requires to allocate a distance matrix which is quadratic in the size of the number of instants of the temporal values. Therefore, memory allocation will fail for temporal values with large number of instants.

```
SELECT dynamicTimeWarpPath(tfloat '[1@2012-01-01, 3@2012-01-03, 1@2012-01-06]',
 tfloat '[1@2012-01-01, 1.5@2012-01-02, 2.5@2012-01-03, 1.5@2012-01-04, 1.5@2012-01-05]');
-- (0,0)
   (1,0)
   (2,1)
   (3,2)
   (4,2)
SELECT dynamicTimeWarpPath(tgeompoint '[Point(1 1)@2012-01-01, Point(3 3)@2012-01-03,
  Point(1 1)@2012-01-05]', tgeompoint '[Point(1.1 1.1)@2012-01-01,
  Point(2.5 2.5)@2012-01-02, Point(4 4)@2012-01-03, Point(3 3)@2012-01-04,
  Point(1.5 2)@2012-01-05]');
-- (0,0)
   (1,1)
   (2,1)
   (3,1)
   (4,2)
```

5.14 Multidimensional Tiling

Multidimensional tiling is the mechanism used to partition the domain of temporal values in buckets or tiles of varying number of dimensions. In the case of a single dimension, the domain can be partitioned by value or by time using buckets of the same size or duration, respectively. For temporal numbers, the domain can be partitioned in two-dimensional tiles of the same size for the value dimension and the same duration for the time dimension. For temporal points, the domain can be partitioned in space in two- or three-dimensional tiles, depending on the number of dimensions of the spatial coordinates. Finally, for temporal points, the domain can be partitioned in space and time using three- or four-dimensional tiles. Furthermore, the temporal values can also be fragmented according to a multidimensional grid defined over the underlying domain.

Multidimensional tiling can be used for various purposes. For example, it can be used for computing multidimensional histograms, where the temporal values are aggregated according to the underlying partition of the domain. On the other hand, multidimensional tiling can be used for distributing a dataset across a cluster of servers, where each server contains a partition of the dataset. The advantage of this partition mechanism is that it preserves proximitity in space and time, unlike the traditional hash-based partition mechanisms used in big data environments.

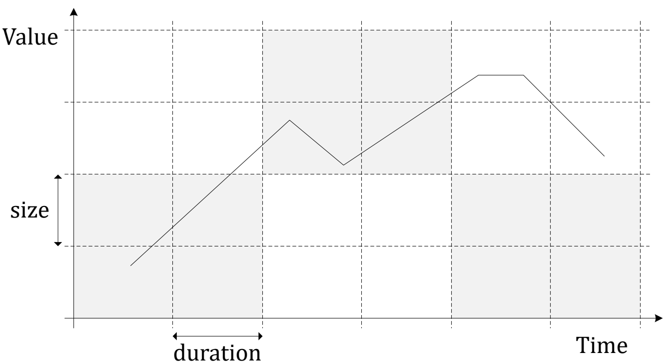

Figure 5.2: Multidimensional tiling for temporal floats.

Figure 5.2 illustrates multidimensional tiling for temporal floats. The two-dimensional domain is split into tiles having the same size for the value dimension and the same duration for the time dimension. Suppose that this tiling scheme is used for distribute a dataset across a cluster of six servers, as suggested by the gray pattern in the figure. In this case, the values are fragmented so each server will receive the data of contiguous tiles. This implies in particular that four nodes will receive one fragment of the temporal float depicted in the figure. One advantage of this distribution of data based on multidimensional tiling is that it reduces the data that needs to be exchanged between nodes when processing queries, a process typically referred to as *reshuffling*.

Many of the functions in this section are *set-returning functions* (also known as a *table functions*) since they typically return more than one value. In this case, the functions are marked with the {} symbol.

5.14.1 Bucket Functions

- Returns a set of couples (index, bucket) that cover the range or period with buckets of the same size or duration aligned with the origin. {}

  If the origin is not specified, it is set by default to 0 for ranges and Monday, January 3, 2000 for periods. The indices start by 1.

  `bucketList(bounds range,size number,origin number=0): setof index_range`

  `bucketList(bounds period,duration interval,origin timestamptz='2000-01-03'): setof index_period`

```
SELECT (bl).index, (bl).range
FROM (SELECT bucketList(tint '[15@2000-01-01, 25@2000-01-10]'::intrange, 2) AS bl) t;
-- 1 | [14,16)
   2 | [16,18)
   3 | [18,20)
   ...
SELECT bucketList(tfloat '[-1@2000-01-01, -10@2000-01-10]'::floatrange, 2, -7);
-- (1,"[-11,-9)")
   (2,"[-9,-7)")
   (3,"[-7,-5)")
   ...
SELECT (bl).index, (bl).period
FROM (SELECT bucketList(tfloat '[1@2000-01-15, 10@2000-01-25]'::period,'2 days') AS bl) t;
-- 1 | [2000-01-15, 2000-01-17)
   2 | [2000-01-17, 2000-01-19)
   3 | [2000-01-19, 2000-01-21)
   ...
SELECT bucketList(tfloat '[1@2000-01-15, 10@2000-01-25]'::period, '2 days', '2000-01-02');
-- (1,"[2000-01-14, 2000-01-16)")
   (2,"[2000-01-16, 2000-01-18)")
   (3,"[2000-01-18, 2000-01-20)")
   ...
```

- Returns the start value of the bucket that contains the input number.

  If the origin is not specified, it is set by default to 0.

  `valueBucket(value number,size number,origin number=0): number`

```
SELECT valueBucket(3, 2);
-- 2
SELECT valueBucket(3.5, 2.5, 1.5);
-- 1.5
```

- Returns the range in the bucket space that contains the input number.

  If the origin is not specified, it is set by default to 0.

  `rangeBucket(value number,size number,origin number=0): range`

```
SELECT rangeBucket(2, 2);
-- [2,4)
SELECT rangeBucket(2, 2, 1);
-- [1,3)
```

```
SELECT rangeBucket(2, 2.5);
-- [0,2.5)
SELECT rangeBucket(2, 2.5, 1.5);
-- [1.5,4)
```

- Returns the start timestamp of the bucket that contains the input timestamp.

  If the origin is not specified, it is set by default to Monday, January 3, 2000.

  `timeBucket(time timestamptz,duration interval,origin timestamptz='2000-01-03'): timestamptz`

```
SELECT timeBucket(timestamptz '2020-05-01', interval '2 days');
-- 2020-04-29 01:00:00+02
SELECT timeBucket(timestamptz '2020-05-01', interval '2 days', timestamptz '2020-01-01');
-- 2020-04-30 01:00:00+02
```

- Returns the period in the bucket space that contains the input timestamp.

  If the origin is not specified, it is set by default to Monday, January 3, 2000.

  `periodBucket(time timestamptz,duration interval,origin timestamptz='2000-01-03'): period`

```
SELECT periodBucket('2000-01-04', interval '1 week');
-- [2000-01-03, 2000-01-10)
SELECT periodBucket('2000-01-04', interval '1 week', '2000-01-07');
-- [1999-12-31, 2000-01-07)
```

5.14.2 Grid Functions

- Returns a set of couples (index, tile) that covers the box with multidimensional tiles of the same size and duration. {}

  If the origin of the value and/or time dimensions are not specified, they are set by default to 0 or `'Point(0 0 0)'` for the value dimension (depending on the box type) and to Monday, January 3, 2000 for the time dimension.

  `multidimGrid(bounds tbox,size float,duration interval,vorigin float=0, torigin timestamptz='2000-01-03'): setof index_box`

  `multidimGrid(bounds stbox,size float,sorigin geometry='Point(0 0 0)'): setof index_box`

  `multidimGrid(bounds stbox,size float,duration interval,sorigin geometry='Point(0 0 0)', torigin timestamptz='2000-01-03'): setof index_box`

  In the case of a spatiotemporal grid, the SRID of the tile coordinates is determined by the input box and the size is given in the units of the SRID. If the origin for the spatial coordinates is given, which must be a point, its dimensionality and SRID should be equal to the one of box, otherwise an error is raised.

```
SELECT (gr).index, (gr).box
FROM (SELECT multidimGrid(tfloat '[15@2000-01-15, 25@2000-01-25]'::tbox, 2.0, '2 days')
  AS gr) t;
-- 1 | TBOX((14,2000-01-15),(16,2000-01-17))
   2 | TBOX((16,2000-01-15),(18,2000-01-17))
   3 | TBOX((18,2000-01-15),(20,2000-01-17))
   ...
SELECT multidimGrid(tfloat '[15@2000-01-15, 25@2000-01-25]'::tbox, 2.0, '2 days', 11.5);
-- (1,"TBOX((13.5,2000-01-15),(15.5,2000-01-17))")
```

```
   (2,"TBOX((15.5,2000-01-15),(17.5,2000-01-17))")
   (3,"TBOX((17.5,2000-01-15),(19.5,2000-01-17))")
   ...
SELECT multidimGrid(tgeompoint '[Point(3 3)@2000-01-15,
  Point(15 15)@2000-01-25]'::stbox, 2.0);
-- (1,"STBOX((2,2),(4,4))")
   (2,"STBOX((4,2),(6,4))")
   (3,"STBOX((6,2),(8,4))")
   ...
SELECT multidimGrid(tgeompoint 'SRID=3812;[Point(3 3)@2000-01-15,
  Point(15 15)@2000-01-25]'::stbox, 2.0, geometry 'Point(3 3)');
-- (1,"SRID=3812;STBOX((3,3),(5,5))")
   (2,"SRID=3812;STBOX((5,3),(7,5))")
   (3,"SRID=3812;STBOX((7,3),(9,5))")
   ...
SELECT multidimGrid(tgeompoint '[Point(3 3 3)@2000-01-15,
  Point(15 15 15)@2000-01-25]'::stbox, 2.0, geometry 'Point(3 3 3)');
-- (1,"STBOX Z((3,3,3),(5,5,5))")
   (2,"STBOX Z((5,3,3),(7,5,5))")
   (3,"STBOX Z((7,3,3),(9,5,5))")
   ...
SELECT multidimGrid(tgeompoint '[Point(3 3)@2000-01-15,
  Point(15 15)@2000-01-25]'::stbox, 2.0, interval '2 days');
-- (1,"STBOX T((2,2,2000-01-15),(4,4,2000-01-17))")
   (2,"STBOX T((4,2,2000-01-15),(6,4,2000-01-17))")
   (3,"STBOX T((6,2,2000-01-15),(8,4,2000-01-17))")
   ...
SELECT multidimGrid(tgeompoint '[Point(3 3 3)@2000-01-15,
  Point(15 15 15)@2000-01-25]'::stbox, 2.0, '2 days', 'Point(3 3 3)', '2000-01-15');
-- (1,"STBOX ZT((3,3,3,2000-01-15),(5,5,5,2000-01-17))")
   (2,"STBOX ZT((5,3,3,2000-01-15),(7,5,5,2000-01-17))")
   (3,"STBOX ZT((7,3,3,2000-01-15),(9,5,5,2000-01-17))")
   ...
```

- Returns the tile of the multidimensional grid that contains the value and the timestamp.

  If the origin of the value and/or time dimensions are not specified, they are set by default to 0 or `Point(0 0 0)` for the value dimension and Monday, January 3, 2000 for the time dimension, respectively.

  `multidimTile(value float,time timestamptz,size float,duration interval, vorigin float=0.0,torigin timestamptz='2000-01-03'): tbox`

  `multidimTile(point geometry,size float,sorigin geometry='Point(0 0 0)'): stbox`

  `multidimTile(point geometry,time timestamptz,size float,duration interval,sorigin geometry='Point(0 0 0)',torigin timestamptz='2000-01-03'): stbox`

  In the case of a spatiotemporal grid, the SRID of the tile coordinates is determined by the input point and the size is given in the units of the SRID. If the origin for the spatial coordinates is given, which must be a point, its dimensionality and SRID should be equal to the one of box, otherwise an error is raised.

```
SELECT multidimTile(15, '2000-01-15', 2, interval '2 days');
-- TBOX((14,2000-01-15),(16,2000-01-17))
SELECT multidimTile(15, '2000-01-15', 2, interval '2 days', 1, '2000-01-02');
-- TBOX((15,2000-01-14),(17,2000-01-16))
SELECT multidimTile(geometry 'Point(1 1 1)', 2.0);
-- STBOX Z((0,0,0),(2,2,2))
SELECT multidimTile(geometry 'Point(1 1)', '2000-01-01', 2.0, interval '2 days');
-- STBOX T((0,0,2000-01-01),(2,2,2000-01-03))
SELECT multidimTile(geometry 'Point(1 1)', '2000-01-01', 2.0, '2 days', 'Point(1 1)',
  '2000-01-02');
-- STBOX T((1,1,1999-12-31),(3,3,2000-01-02))
```

5.14.3 Split Functions

These functions fragment a temporal value with respect to a sequence of buckets (see Section 5.14.1) or a multidimensional grid (see Section 5.14.2).

- Fragment the temporal number with respect to range buckets. {}

  `valueSplit(value tnumber,size number,origin number=0): setof number_tnumber`

  If the origin of values is not specified, it is set by default to 0.

```
SELECT (sp).number, (sp).tnumber
FROM (SELECT valueSplit(tint '[1@2012-01-01, 2@2012-01-02, 5@2012-01-05, 10@2012-01-10]',
  2) AS sp) t;
--  0 | {[1@2012-01-01 00:00:00+01, 1@2012-01-02 00:00:00+01)}
    2 | {[2@2012-01-02 00:00:00+01, 2@2012-01-05 00:00:00+01)}
    4 | {[5@2012-01-05 00:00:00+01, 5@2012-01-10 00:00:00+01)}
   10 | {[10@2012-01-10 00:00:00+01]}
SELECT valueSplit(tfloat '[1@2012-01-01, 10@2012-01-10)', 2.0, 1.0);
-- (1,"{[1@2012-01-01 00:00:00+01, 3@2012-01-03 00:00:00+01)}")
   (3,"{[3@2012-01-03 00:00:00+01, 5@2012-01-05 00:00:00+01)}")
   (5,"{[5@2012-01-05 00:00:00+01, 7@2012-01-07 00:00:00+01)}")
   (7,"{[7@2012-01-07 00:00:00+01, 9@2012-01-09 00:00:00+01)}")
   (9,"{[9@2012-01-09 00:00:00+01, 10@2012-01-10 00:00:00+01)}")
```

- Fragment the temporal value with respect to time buckets. {}

  `timeSplit(value ttype,duration interval,origin timestamptz='2000-01-03'): setof time_temp`

  If the origin of time is not specified, it is set by default to Monday, January 3, 2000.

```
SELECT (ts).time, (ts).temp
FROM (SELECT timeSplit(tfloat '[1@2012-01-01, 10@2012-01-10)', '2 days') AS ts) t;
-- 2011-12-31 | [1@2012-01-01, 2@2012-01-02)
   2012-01-02 | [2@2012-01-02, 4@2012-01-04)
   2012-01-04 | [4@2012-01-04, 6@2012-01-06)
   ...
SELECT (ts).time, astext((ts).temp) AS temp
FROM (SELECT timeSplit(tgeompoint '[Point(1 1)@2012-01-01, Point(10 10)@2012-01-10]',
  '2 days', '2012-01-01') AS ts) AS t;
-- 2012-01-01 | [POINT Z (1 1 1)@2012-01-01, POINT Z (3 3 3)@2012-01-03)
   2012-01-03 | [POINT Z (3 3 3)@2012-01-03, POINT Z (5 5 5)@2012-01-05)
   2012-01-05 | [POINT Z (5 5 5)@2012-01-05, POINT Z (7 7 7)@2012-01-07)
   ...
```

  Notice that we can fragment a temporal value in cyclic (instead of linear) time buckets. The following two examples show how to fragment a temporal value by hour and by day of the week.

```
SELECT (ts).time::time as hour, merge((ts).temp) as temp
FROM (SELECT timeSplit(tfloat '[1@2012-01-01, 10@2012-01-03]', '1 hour') AS ts) t
GROUP BY hour ORDER BY hour;

-- 00:00:00 | {[1@2012-01-01 00:00:00+01, 1.1875@2012-01-01 01:00:00+01),
               [5.5@2012-01-02 00:00:00+01, 5.6875@2012-01-02 01:00:00+01)}
   01:00:00 | {[1.1875@2012-01-01 01:00:00+01, 1.375@2012-01-01 02:00:00+01),
               [5.6875@2012-01-02 01:00:00+01, 5.875@2012-01-02 02:00:00+01)}
   02:00:00 | {[1.375@2012-01-01 02:00:00+01, 1.5625@2012-01-01 03:00:00+01),
               [5.875@2012-01-02 02:00:00+01, 6.0625@2012-01-02 03:00:00+01)}
```

```
   03:00:00 | {[1.5625@2012-01-01 03:00:00+01, 1.75@2012-01-01 04:00:00+01),
               [6.0625@2012-01-02 03:00:00+01, 6.25@2012-01-02 04:00:00+01)}
   ...
SELECT EXTRACT(DOW FROM (ts).time) as dow_no, TO_CHAR((ts).time, 'Dy') as dow,
  asText(round(merge((ts).temp), 2)) as temp
FROM (SELECT timeSplit(tgeompoint '[Point(1 1)@2012-01-01, Point(10 10)@2012-01-14)',
  '1 hour') AS ts) t
GROUP BY dow, dow_no ORDER BY dow_no;
-- 0 | Sun | {[POINT(1 1)@2012-01-01, POINT(1.69 1.69)@2012-01-02),
              [POINT(5.85 5.85)@2012-01-08, POINT(6.54 6.54)@2012-01-09)}
   1 | Mon | {[POINT(1.69 1.69)@2012-01-02, POINT(2.38 2.38)@2012-01-03),
              [POINT(6.54 6.54)@2012-01-09, POINT(7.23 7.23)@2012-01-10)}
   2 | Tue | {[POINT(2.38 2.38)@2012-01-03, POINT(3.08 3.08)@2012-01-04),
              [POINT(7.23 7.23)@2012-01-10, POINT(7.92 7.92)@2012-01-11)}
   ...
```

- Fragment the temporal number with respect to the tiles in a value-time grid. {}

  `valueTimeSplit(value tumber,size number,duration interval,vorigin number=0, torigin timestamptz='2000-01-03'): setof number_time_tnumber`

  If the origin of values and/or time are not specified, they are set by default to 0 and to Monday, January 3, 2000, respectively.

```
SELECT (sp).number, (sp).time, (sp).tnumber
FROM (SELECT valueTimeSplit(tint '[1@2012-01-01, 2@2012-01-02, 5@2012-01-05,
  10@2012-01-10]', 5, '5 days') AS sp) t;
-- 0 | 2011-12-31 | [1@2012-01-01, 2@2012-01-02, 2@2012-01-05)
   5 | 2012-01-05 | [5@2012-01-05, 5@2012-01-10)
  10 | 2012-01-10 | [10@2012-01-10]
SELECT (sp).number, (sp).time, (sp).tnumber
FROM (SELECT valueTimeSplit(tfloat '[1@2012-01-01, 10@2012-01-10)', 5.0, '5 days', 1.0,
  '2012-01-01') AS sp) t;
-- 1 | 2012-01-01 | [1@2012-01-01, 6@2012-01-06)
   6 | 2012-01-06 | [6@2012-01-06, 10@2012-01-10)
```

- Fragment the temporal point with respect to the tiles in a spatial grid. {}

  `spaceSplit(value tgeompoint,size float,origin geometry='Point(0 0 0)', bitmatrix=TRUE): setof point_tpoint`

  If the origin of the space dimension is not specified, it is set by default to `'Point(0 0 0)'`. If the argument `bitmatrix` is not specified, then the computation will use a bit matrix to speed up the process.

```
SELECT ST_AsText((sp).point) AS point, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceSplit(tgeompoint '[Point(1 1)@2020-03-01, Point(10 10)@2020-03-10]',
  2.0) AS sp) t;
-- POINT(0 0) | {[POINT(1 1)@2020-03-01, POINT(2 2)@2020-03-02)}
   POINT(2 2) | {[POINT(2 2)@2020-03-02, POINT(4 4)@2020-03-04)}
   POINT(4 4) | {[POINT(4 4)@2020-03-04, POINT(6 6)@2020-03-06)}
   ...
SELECT ST_AsText((sp).point) AS point, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceSplit(tgeompoint '[Point(1 1 1)@2020-03-01,
  Point(10 10 10)@2020-03-10]', 2.0, 'Point(1 1 1)') AS sp) t;
-- POINT Z (1 1 1) | {[POINT Z (1 1 1)@2020-03-01, POINT Z (3 3 3)@2020-03-03)}
   POINT Z (3 3 3) | {[POINT Z (3 3 3)@2020-03-03, POINT Z (5 5 5)@2020-03-05)}
   POINT Z (5 5 5) | {[POINT Z (5 5 5)@2020-03-05, POINT Z (7 7 7)@2020-03-07)}
   ...
```

- Fragment the temporal point with respect to the tiles in a spatiotemporal grid. {}

  `spaceTimeSplit(value tgeompoint,size float,duration interval,sorigin geometry='Point(0 0 0)',torigin timestamptz='2000-01-03', bitmatrix=TRUE): setof point_time_tpoint`

  If the origin of the space and time dimensions are not specified, they are set by default to `'Point(0 0 0)'` and Monday, January 3, 2000, respectively. If the argument `bitmatrix` is not specified, then the computation will use a bit matrix to speed up the process.

```
SELECT ST_AsText((sp).point) AS point, (sp).time, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceTimeSplit(tgeompoint '[Point(1 1)@2020-03-01, Point(10 10)@2020-03-10]',
  2.0, '2 days') AS sp) t;
-- POINT(0 0) | 2020-03-01 | {[POINT(1 1)@2020-03-01, POINT(2 2)@2020-03-02)}
   POINT(2 2) | 2020-03-01 | {[POINT(2 2)@2020-03-02, POINT(3 3)@2020-03-03]}
   POINT(2 2) | 2020-03-03 | {[POINT(3 3)@2020-03-03, POINT(4 4)@2020-03-04)}
   ...
SELECT ST_AsText((sp).point) AS point, (sp).time, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceTimeSplit(tgeompoint '[Point(1 1 1)@2020-03-01,
  Point(10 10 10)@2020-03-10]', 2.0, '2 days', 'Point(1 1 1)', '2020-03-01') AS sp) t;
-- POINT Z(1 1 1) | 2020-03-01 | {[POINT Z(1 1 1)@2020-03-01, POINT Z(3 3 3)@2020-03-03)}
   POINT Z(3 3 3) | 2020-03-03 | {[POINT Z(3 3 3)@2020-03-03, POINT Z(5 5 5)@2020-03-05)}
   POINT Z(5 5 5) | 2020-03-05 | {[POINT Z(5 5 5)@2020-03-05, POINT Z(7 7 7)@2020-03-07)}
   ...
```

5.15 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): {tint_instset,tint_seqset}`

```
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): {ttype_instset,ttype_seqset}`

```
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): {ttype_instset,ttype_seqset}`

```
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): {tnumberinstset,tnumberseqset}`

```
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): {tfloat_instset,tfloat_seqset}`

```
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): {tnumberinstset,tnumberseqset}`

```
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): {tnumberinstset,tnumberseqset}`

```
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): {tint_instset,tint_seqset}`

```
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): {tint_instset,tint_seqset}`

```
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): {tfloat_instset,tfloat_seqset}`

```
SELECT wavg(NoEmps, interval '2 days') FROM Department;
-- "{[10@2012-01-01, 10@2012-02-01), [7@2012-02-01, 7@2012-04-01),
  [8.66666666666667@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)}"
```

5.16 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"
```

5.17 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 and the SP-GiST index implements an n-dimensional quad-tree. 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 5.7 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.

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 5.12.6) 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);
```

5.18 Statistics and Selectivity for Temporal Types

5.18.1 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.

5.18.2 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 `integer`, `timestamptz`, etc.) or a MobilityDB type (such as `period`, `tint`, 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.

MobilityDB estimates restriction and join selectivity for temporal types, altough join selectivity for temporal numbers uses a default selectivity value for the value dimension since PostgreSQL currently does not provide join selectivity for range types.

Chapter 6

Temporal Network Points

The temporal points that we have considered so far represent the movement of objects that can move freely on space since it is assumed that they can change their position from one location to the next one without any motion restriction. This is the case for animals and for flying objects such as planes or drones. However, in many cases, objects do not move freely in space but rather within spatially embedded networks such as routes or railways. In this case, it is necessary to take the embedded networks into account while describing the movements of these moving objects. Temporal network points account for these requirements.

Compared with the free-space temporal points, network-based points have the following advantages:

- Network points provide road constraints that reflect the real movements of moving objects.
- The geometric information is not stored with the moving point, but once and for all in the fixed networks. In this way, the location representations and interpolations are more precise.
- Network points are more efficient in terms of data storage, location update, formulation of query, as well as indexing. These are discussed later in this document.

Temporal network points are based on pgRouting, a PostgreSQL extension for developing network routing applications and doing graph analysis. Therefore, temporal network points asume that the underlying network is defined in a table named `ways`, which has at least three columns: `gid` containing the unique route identifier, `length` containing the route length, and `the_geom` containing the route geometry.

There are two static network types, `npoint` (short for network point) and `nsegment` (short for network segment), which represent, respectively, a point and a segment of a route. An `npoint` value is composed of a route identifier and a float number in the range [0,1] determining a relative position of the route, where 0 corresponds to the begining of the route and 1 to the end of the route. An `nsegment` value is composed of a route identifier and two float numbers in the range [0,1] determining the start and end relative positions. A `nsegment` value whose start and end positions are equal corresponds to an `npoint` value.

The `npoint` type serves as base type for defining the temporal network point type `tnpoint`. The `tnpoint` type has similar functionality as the temporal point type `tgeompoint` with the exception that it only considers two dimensions. Thus, all functions and operators described before for the `tgeompoint` type are also applicable for the `tnpoint` type. In addition, there are specific functions defined for the `tnpoint` type.

6.1 Static Network Types

An `npoint` value is a couple of the form `(rid,position)` where `rid` is a `bigint` value representing a route identifier and `position` is a `float` value in the range [0,1] indicating its relative position. The values 0 and 1 of `position` denote, respectively, the starting and the ending position of the route. The road distance between an `npoint` value and the starting position of route with identifier `rid` is computed by multiplying `position` by length, where `length` is the route length. Examples of input of network point values are as follows:

```
SELECT npoint 'Npoint(76, 0.3)';
SELECT npoint 'Npoint(64, 1.0)';
```

The constructor function for network points has one argument for the route identifier and one argument for the relative position. An example of a network point value defined with the constructor function is as follows:

```
SELECT npoint(76, 0.3);
```

An `nsegment` value is a triple of the form `(rid,startPosition,endPosition)` where `rid` is a `bigint` value representing a route identifier and `startPosition` and `endPosition` are `float` values in the range [0,1] such that `startPosition` ≤ `endPosition`. Semantically, a network segment represents a set of network points `(rid,position)` with `startPosition` ≤ `position` ≤ `endPosition`. If `startPosition=0` and `endPosition=1`, the network segment is equivalent to the entire route. If `startPosition=endPosition`, the network segment represents into a single network point. Examples of input of network point values are as follows:

```
SELECT nsegment 'Nsegment(76, 0.3, 0.5)';
SELECT nsegment 'Nsegment(64, 0.5, 0.5)';
SELECT nsegment 'Nsegment(64, 0.0, 1.0)';
SELECT nsegment 'Nsegment(64, 1.0, 0.0)';
-- converted to nsegment 'Nsegment(64, 0.0, 1.0)';
```

As can be seen in the last example, the `startPosition` and `endPosition` values will be inverted to ensure that the condition `startPosition` ≤ `endPosition` is always satisfied. The constructor function for network segments has one argument for the route identifier and two optional arguments for the start and end positions. Examples of network segment values defined with the constructor function are as follows:

```
SELECT nsegment(76, 0.3, 0.3);
SELECT nsegment(76); -- start and end position assumed to be 0 and 1 respectively
SELECT nsegment(76, 0.5); -- end position assumed to be 1
```

Values of the `npoint` type can be converted to the `nsegment` type using an explicit `CAST` or using the `::` notation as shown next.

```
SELECT npoint(76, 0.33)::nsegment;
```

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

- The route identifier `rid` must be found in column `gid` of table `ways`.
- The `position`, `startPosition`, and `endPosition` values must be in the range [0,1]. An error is raised whenever one of these constraints are not satisfied.

Examples of incorrect static network type values are as follows.

```
-- incorrect rid value
SELECT npoint 'Npoint(87.5, 1.0)';
-- incorrect position value
SELECT npoint 'Npoint(87, 2.0)';
-- rid value not found in the ways table
SELECT npoint 'Npoint(99999999, 1.0)';
```

We give next the functions and operators for the static network types.

6.1.1 Constructor Functions

- Constructor for network points

  `npoint(bigint,double precision): npoint`

```
SELECT npoint(76, 0.3);
```

- Constructor for network segments

  `nsegment(bigint,double precision,double precision): nsegment`

```
SELECT nsegment(76, 0.3, 0.5);
```

6.1.2 Modification Functions

- Round the position(s) of the network point or the network segment to the number of decimal places

  `round({npoint,nsegment},integer): {npoint,nsegment}`

```
SELECT round(npoint(76, 0.123456789), 6);
--  NPoint(76,0.123457)
SELECT round(nsegment(76, 0.123456789, 0.223456789), 6);
--  NSegment(76,0.123457,0.223457)
```

6.1.3 Accessor Functions

- Get the route identifier

  `route({npoint,nsegment}): bigint`

```
SELECT route(npoint 'Npoint(63, 0.3)');
-- 63
SELECT route(nsegment 'Nsegment(76, 0.3, 0.3)');
-- 76
```

- Get the position

  `getPosition(npoint): float`

```
SELECT getPosition(npoint 'Npoint(63, 0.3)');
-- 0.3
```

- Get the start position

  `startPosition(npoint): float`

```
SELECT startPosition(nsegment 'Nsegment(76, 0.3, 0.5)');
-- 0.3
```

- Get the end position

  `endPosition(npoint): float`

```
SELECT endPosition(nsegment 'Nsegment(76, 0.3, 0.5)');
-- 0.5
```

6.1.4 Spatial Functions

- Get the spatial reference identifier

  `srid({npoint,nsegment}): int`

```
SELECT SRID(nspoint 'Npoint(76, 0.3)');
-- 5676
SELECT SRID(nsegment 'Nsegment(76, 0.3, 0.5)');
-- 5676
```

Values of the `npoint` and `nsegment` types can be converted to the `geometry` type using an explicit `CAST` or using the `::` notation as shown next.

- Cast a network point to a geometry

  `{npoint,nsegment}::geometry`

```
SELECT ST_AsText(npoint(76, 0.33)::geometry);
-- POINT(21.6338731332283 50.0545869554067)
SELECT ST_AsText(nsegment(76, 0.33, 0.66)::geometry);
-- LINESTRING(21.6338731332283 50.0545869554067,30.7475989651999 53.9185062927473)
SELECT ST_AsText(nsegment(76, 0.33, 0.33)::geometry);
-- POINT(21.6338731332283 50.0545869554067)
```

Similarly, `geometry` values of subtype `point` or `linestring` (restricted to two points) can be converted, respectively, to `npoint` and `nsegment` values using an explicit `CAST` or using the `::` notation. For this, the route that intersects the given points must be found, where a tolerance of 0.00001 units (depending on the coordinate system) is assumed so a point and a route that are close are considered to intersect. If no such route is found, a null value is returned.

- Cast a geometry to a network point

  `geometry::{npoint,nsegment}`

```
SELECT geometry 'Point(279.269156511873 811.497076880187)'::npoint;
-- NPoint(3,0.781413)
SELECT geometry 'LINESTRING(406.729536784738 702.58583437902,
  383.570801314823 845.137059419277)'::nsegment;
-- NSegment(3,0.6,0.9)
SELECT geometry 'Point(279.3 811.5)'::npoint;
-- NULL
SELECT geometry 'LINESTRING(406.7 702.6,383.6 845.1)'::nsegment;
-- NULL
```

Two `npoint` values may be have different route identifiers but may represent the same spatial point at the intersection of the two routes. Function `equals` is used for testing spatial equality of network points.

- Spatial equality for network points

  `equals(npoint, npoint)::Boolean`

```
WITH inter(geom) AS (
  SELECT st_intersection(t1.the_geom, t2.the_geom)
  FROM ways t1, ways t2 WHERE t1.gid = 1 AND t2.gid = 2),
fractions(f1, f2) AS (
  SELECT ST_LineLocatePoint(t1.the_geom, i.geom), ST_LineLocatePoint(t2.the_geom, i.geom)
  FROM ways t1, ways t2, inter i WHERE t1.gid = 1 AND t2.gid = 2)
SELECT equals(npoint(1, f1), npoint(2, f2)) FROM fractions;
-- true
```

6.1.5 Comparison Operators

The comparison operators (=, <, and so on) for static network types require that the left and right arguments be of the same type. Excepted the equality and inequality, the other comparison operators are not useful in the real world but allow B-tree indexes to be constructed on static network types.

- Are the values equal?

  `{npoint,nsegment} = {npoint,nsegment}`

```
SELECT npoint 'Npoint(3, 0.5)' = npoint 'Npoint(3, 0.5)';
-- true
SELECT nsegment 'Nsegment(3, 0.5, 0.5)' = nsegment 'Nsegment(3, 0.5, 0.6)';
-- false
```

- Are the values different?

  `{npoint,nsegment} <> {npoint,nsegment}`

```
SELECT npoint 'Npoint(3, 0.5)' <> npoint 'Npoint(3, 0.6);
-- true
SELECT nsegment 'Nsegment(3, 0.5, 0.5)' <> nsegment 'Nsegment(3, 0.5, 0.5)';
-- false
```

- Is the first value less than the second one?

  `{npoint,nsegment} < {npoint,nsegment}`

```
SELECT nsegment 'Nsegment(3, 0.5, 0.5)' < nsegment 'Nsegment(3, 0.5, 0.6)';
-- true
```

- Is the first value greater than the second one?

  `{npoint,nsegment} > {npoint,nsegment}`

```
SELECT nsegment 'Nsegment(3, 0.5, 0.5)' > nsegment 'Nsegment(2, 0.5, 0.5)';
-- true
```

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

  `{npoint,nsegment} <= {npoint,nsegment}`

```
SELECT npoint 'Npoint(1, 0.5)' <= npoint 'Npoint(2, 0.5)';
-- true
```

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

  `{npoint,nsegment} >= {npoint,nsegment}`

```
SELECT npoint 'Npoint(1, 0.6)' >= npoint 'Npoint(1, 0.5)';
-- true
```

6.2 Temporal Network Points

The temporal network point type `tnpoint` allows to represent the movement of objects over a network. It corresponds to the temporal point type `tgeompoint` restricted to two-dimensional coordinates. As all the other temporal types it comes in four subtypes, namely, instant, instant set, sequence, and sequence set. Examples of `tnpoint` values in these subtypes are given next.

```
SELECT tnpoint 'Npoint(1, 0.5)@2000-01-01';
SELECT tnpoint '{Npoint(1, 0.3)@2000-01-01, Npoint(1, 0.5)@2000-01-02,
  Npoint(1, 0.5)@2000-01-03}';
SELECT tnpoint '[Npoint(1, 0.2)@2000-01-01, Npoint(1, 0.4)@2000-01-02,
  Npoint(1, 0.5)@2000-01-03]';
SELECT tnpoint '{[Npoint(1, 0.2)@2000-01-01, Npoint(1, 0.4)@2000-01-02,
  Npoint(1, 0.5)@2000-01-03], [Npoint(2, 0.6)@2000-01-04, Npoint(2, 0.6)@2000-01-05]}';
```

The temporal network point type accepts type modifiers (or `typmod` in PostgreSQL terminology). 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.

```
SELECT tnpoint(Sequence) '[Npoint(1, 0.2)@2000-01-01, Npoint(1, 0.4)@2000-01-02,
  Npoint(1, 0.5)@2000-01-03]';
SELECT tnpoint(Sequence) 'Npoint(1, 0.2)@2000-01-01';
-- ERROR: Temporal type (Instant) does not match column type (Sequence)
```

Temporal network point values of sequence subtype must be defined on a single route. Therefore, a value of sequence set subtype is needed for representing the movement of an object that traverses several routes, even if there is no temporal gap. For example, in the following value

```
SELECT tnpoint '{[NPoint(1, 0.2)@2001-01-01, NPoint(1, 0.5)@2001-01-03),
  [NPoint(2, 0.4)@2001-01-03, NPoint(2, 0.6)@2001-01-04)}';
```

the network point changes its route at 2001-01-03.

Temporal network point values of sequence or sequence set subtype are converted into a normal form so that equivalent values have identical representations. For this, consecutive instant values are merged when possible. 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 a normal form are as follows.

```
SELECT tnpoint '[NPoint(1, 0.2)@2001-01-01, NPoint(1, 0.4)@2001-01-02,
  NPoint(1, 0.6)@2001-01-03)';
-- [NPoint(1,0.2)@2001-01-01, NPoint(1,0.6)@2001-01-03)
SELECT tnpoint '{[NPoint(1, 0.2)@2001-01-01, NPoint(1, 0.3)@2001-01-02,
  NPoint(1, 0.5)@2001-01-03), [NPoint(1, 0.5)@2001-01-03, NPoint(1, 0.7)@2001-01-04)}';
-- {[NPoint(1,0.2)@2001-01-01, NPoint(1,0.3)@2001-01-02, NPoint(1,0.7)@2001-01-04)}
```

6.3 Validity of Temporal Network Points

Temporal network point values must satisfy the constraints specified in Section 3.2 so that they are well defined. An error is raised whenever one of these constraints are not satisfied. Examples of incorrect values are as follows.

```
-- null values are not allowed
SELECT tnpoint 'NULL@2001-01-01 08:05:00';
SELECT tnpoint 'Point(0 0)@NULL';
-- base type is not a network point
SELECT tnpoint 'Point(0 0)@2001-01-01 08:05:00';
-- multiple routes in a sequence
SELECT tnpoint '[Npoint(1, 0.2)@2001-01-01 09:00:00, Npoint(2, 0.2)@2001-01-01 09:05:00)';
```

6.4 Constructors for Temporal Network Points

- Constructor for temporal network points of instant subtype

  `tnpoint_inst(val npoint,t timestamptz):tnpoint_inst`

  ```
  SELECT tnpoint_inst('Npoint(1, 0.5)', '2000-01-01');
  -- NPoint(1,0.5)@2000-01-01
  ```

- Constructors for temporal network points of instant set subtype

  `tnpoint_instset(tnpoint[]):tnpoint_instset`

  `tnpoint_instset(npoint,timestampset):tnpoint_instset`

  ```
  SELECT tnpoint_instset(ARRAY[tnpoint 'Npoint(1, 0.3)@2000-01-01',
    'Npoint(1, 0.5)@2000-01-02', 'Npoint(1, 0.5)@2000-01-03']);
  -- {NPoint(1,0.3)@2000-01-01, NPoint(1,0.5)@2000-01-02, NPoint(1,0.5)@2000-01-03}
  SELECT tnpoint_instset('Npoint(1, 0.3)', '{2000-01-01, 2000-01-03, 2000-01-05}');
  -- {NPoint(1,0.3)@2000-01-01, NPoint(1,0.3)@2000-01-03, NPoint(1,0.3)@2000-01-05}
  ```

- Constructor for temporal network points of sequence subtype

  `tnpoint_seq(tnpoint[],lower_inc boolean=true,upper_inc boolean=true,`
  ` linear boolean=true):tnpoint_seq`

  `tnpoint_seq(npoint,period,linear boolean=true):tnpoint_seq`

  ```
  SELECT tnpoint_seq(ARRAY[tnpoint 'Npoint(1, 0.2)@2000-01-01', 'Npoint(1, 0.4)@2000-01-02',
    'Npoint(1, 0.5)@2000-01-03']);
  -- [NPoint(1,0.2)@2000-01-01, NPoint(1,0.4)@2000-01-02, NPoint(1,0.5)@2000-01-03]
  SELECT tnpoint_seq(npoint 'Npoint(1, 0.2)', '[2000-01-01, 2000-01-03]', false);
  -- Interp=Stepwise;[NPoint(1,0.2)@2000-01-01, NPoint(1,0.2)@2000-01-03]
  ```

- Constructor for temporal network points of sequence set subtype

  `tnpoint_seqset(tnpoint[]):tnpoint_seqset`

  `tnpoint_seqset(npoint,periodset,boolean=true):tnpoint_seqset`

```
SELECT tnpoint_seqset(ARRAY[tnpoint '[Npoint(1, 0.2)@2000-01-01, Npoint(1, 0.4)@2000 ↩
    -01-02,
  Npoint(1, 0.5)@2000-01-03]', '[Npoint(2, 0.6)@2000-01-04, Npoint(2, 0.6)@2000-01-05]']);
-- {[NPoint(1,0.2)@2000-01-01, NPoint(1,0.4)@2000-01-02, NPoint(1,0.5)@2000-01-03],
  [NPoint(2,0.6)@2000-01-04, NPoint(2,0.6)@2000-01-05]}
```

6.5 Casting for Temporal Network Points

A temporal network point value can be converted to and from a temporal geometry point. This can be done using an explicit `CAST` or using the `::` notation. A null value is returned if any of the composing geometry point values cannot be converted into a `npoint` value.

- Cast a temporal network point to a temporal geometry point

  `tnpoint::tgeompoint`

```
SELECT astext((tnpoint '[NPoint(1, 0.2)@2001-01-01,
  NPoint(1, 0.3)@2001-01-02)')::tgeompoint);
-- [POINT(23.057077727326 28.7666335767956)@2001-01-01,
  POINT(48.7117553116406 20.9256801894708)@2001-01-02)
```

- Cast a temporal geometry point to a temporal network point

  `tgeompoint::tnpoint`

```
SELECT tgeompoint '[POINT(23.057077727326 28.7666335767956)@2001-01-01,
  POINT(48.7117553116406 20.9256801894708)@2001-01-02)'::tnpoint
-- [NPoint(1,0.2)@2001-01-01, NPoint(1,0.3)@2001-01-02)
SELECT tgeompoint '[POINT(23.057077727326 28.7666335767956)@2001-01-01,
  POINT(48.7117553116406 20.9)@2001-01-02)'::tnpoint
-- NULL
```

We give next the functions and operators for network point types.

6.6 Functions and Operators for Temporal Network Points

All functions for temporal types described in Chapter 5 can be applied for temporal network point types. Therefore, in the signatures of the functions, the notation `base` also represents an `npoint` and the notations `ttype`, `tpoint`, and `tgeompoint` also represent a `tnpoint`. Furthermore, the functions that have an argument of type `geometry` accept in addition an argument of type `npoint`. To avoid redundancy, we only present next some examples of these functions and operators for temporal network points.

- Transform a temporal network point to another subtype

  `tnpoint_inst(tnpoint): tnpoint_inst`

  `tnpoint_instset(tnpoint): tnpoint_instset`

  `tnpoint_seq(tnpoint): tnpoint_seq`

  `tnpoint_seqset(tnpoint): tnpoint_seqset`

```
SELECT tnpoint_seqset(tnpoint 'NPoint(1, 0.5)@2001-01-01');
-- {[NPoint(1,0.5)@2001-01-01]}
```

- Round the fraction of the temporal network point to the number of decimal places

  `round(tnpoint,integer): tnpoint`

```
SELECT round(tnpoint '{[NPoint(1, 0.123456789)@2012-01-01, NPoint(1, 0.5)@2012-01-02)}', ↩
    6);
-- {[NPoint(1,0.123457)@2012-01-01 00:00:00+01, NPoint(1,0.5)@2012-01-02 00:00:00+01)}
```

- Get the values

  `getValues(tnpoint): npoint[]`

```
SELECT getValues(tnpoint '{[NPoint(1, 0.3)@2012-01-01, NPoint(1, 0.5)@2012-01-02)}');
-- {"NPoint(1,0.3)","NPoint(1,0.5)"}
SELECT getValues(tnpoint '{[NPoint(1, 0.3)@2012-01-01, NPoint(1, 0.3)@2012-01-02)}');
-- {"NPoint(1,0.3)"}
```

- Get the value at a timestamp

  `valueAtTimestamp(tnpoint,timestamptz): npoint`

```
SELECT valueAtTimestamp(tnpoint '[NPoint(1, 0.3)@2012-01-01, NPoint(1, 0.5)@2012-01-03)',
  '2012-01-02');
-- NPoint(1,0.4)
```

- Get the length traversed by the temporal network point

  `length(tnpoint): float`

```
SELECT length(tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]');
-- 54.3757408468784
```

- Get the cumulative length traversed by the temporal network point

  `cumulativeLength(tnpoint): tfloat`

```
SELECT cumulativeLength(tnpoint '{[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02,
  NPoint(1, 0.5)@2000-01-03], [NPoint(1, 0.6)@2000-01-04, NPoint(1, 0.7)@2000-01-05]}');
-- {[0@2000-01-01, 54.3757408468784@2000-01-02, 54.3757408468784@2000-01-03],
  [54.3757408468784@2000-01-04, 81.5636112703177@2000-01-05]}
```

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

  `speed({tnpoint_seq, tpoint_seqset}): tfloat_seqset`

```
SELECT speed(tnpoint '[NPoint(1, 0.1)@2000-01-01, NPoint(1, 0.4)@2000-01-02,
  NPoint(1, 0.6)@2000-01-03]') * 3600 * 24;
-- Interp=Stepwise;[21.4016800272077@2000-01-01, 14.2677866848051@2000-01-02,
  14.2677866848051@2000-01-03]
```

- Construct the bounding box from a npoint and, optionally, a timestamp or a period

  `stbox(npoint): stbox`

  `stbox(npoint,{timestamptz,period}): stbox`

```
SELECT stbox(npoint 'NPoint(1,0.3)');
-- STBOX((48.711754,20.92568),(48.711758,20.925682))
SELECT stbox(npoint 'NPoint(1,0.3)', timestamptz '2000-01-01');
-- STBOX T((62.786633,80.143555,2000-01-01),(62.786636,80.143562,2000-01-01))
SELECT stbox(npoint 'NPoint(1,0.3)', period '[2000-01-01,2000-01-02]');
-- STBOX T((62.786633,80.143555,2000-01-01),(62.786636,80.143562,2000-01-02))
```

- Get the time-weighted centroid

  `twCentroid(tnpoint): geometry(Point)`

```
SELECT st_astext(twCentroid(tnpoint '{[NPoint(1, 0.3)@2012-01-01,
  NPoint(1, 0.5)@2012-01-02, NPoint(1, 0.5)@2012-01-03, NPoint(1, 0.7)@2012-01-04)}'));
-- POINT(79.9787466444847 46.2385558051041)
```

- Get the temporal azimuth

  `azimuth(tnpoint): tfloat`

```
SELECT azimuth(tnpoint '[NPoint(2, 0.3)@2012-01-01, NPoint(2, 0.7)@2012-01-02]');
-- {[0.974681063778863@2012-01-01 00:00:00+01,
  0.974681063778863@2012-01-01 23:54:36.721091+01),
  [3.68970843029227@2012-01-01 23:54:36.721091+01,
  3.68970843029227@2012-01-02 00:00:00+01)}
```

  Since the underlying geometry associated to a route may have several vertices, the azimuth value may change between instants of the input temporal network point, as shown in the example above.

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

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

```
SELECT nearestApproachInstant(tnpoint '[NPoint(2, 0.3)@2012-01-01,
  NPoint(2, 0.7)@2012-01-02]', geometry 'Linestring(50 50,55 55)');
-- NPoint(2,0.349928)@2012-01-01 02:59:44.402905+01
SELECT nearestApproachInstant(tnpoint '[NPoint(2, 0.3)@2012-01-01,
  NPoint(2, 0.7)@2012-01-02]', npoint 'NPoint(1, 0.5)');
-- NPoint(2,0.592181)@2012-01-01 17:31:51.080405+01
```

- Get the smallest distance ever between the two arguments

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

```
SELECT nearestApproachDistance(tnpoint '[NPoint(2, 0.3)@2012-01-01,
  NPoint(2, 0.7)@2012-01-02]', geometry 'Linestring(50 50,55 55)');
-- 1.41793220500979
SELECT nearestApproachDistance(tnpoint '[NPoint(2, 0.3)@2012-01-01,
  NPoint(2, 0.7)@2012-01-02]', npoint 'NPoint(1, 0.5)');
-- NPoint(2,0.592181)@2012-01-01 17:31:51.080405+01
```

  Function `nearestApproachDistance` has an associated operator `|=|` that can be used for doing nearest neighbor searches using a GiST index (see Section 5.17).

- Get the line connecting the nearest approach point between the two arguments

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

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

```
SELECT st_astext(shortestLine(tnpoint '[NPoint(2, 0.3)@2012-01-01,
  NPoint(2, 0.7)@2012-01-02]', geometry 'Linestring(50 50,55 55)'));
-- LINESTRING(50.7960725266492 48.8266286733015,50 50)
SELECT st_astext(shortestLine(tnpoint '[NPoint(2, 0.3)@2012-01-01,
  NPoint(2, 0.7)@2012-01-02]', npoint 'NPoint(1, 0.5)'));
-- LINESTRING(77.0902838115125 66.6659083092593,90.8134936900394 46.4385792121146)
```

- Restrict to a value

  `atValue(tnpoint,base): tnpoint`

```
SELECT atValue(tnpoint '[NPoint(2, 0.3)@2012-01-01, NPoint(2, 0.7)@2012-01-03]',
  'NPoint(2, 0.5)');
-- {[NPoint(2,0.5)@2012-01-02]}
```

- Restrict to a geometry

  `atGeometry(tnpoint,geometry): tnpoint`

```
SELECT atGeometry(tnpoint '[NPoint(2, 0.3)@2012-01-01, NPoint(2, 0.7)@2012-01-03]',
  'Polygon((40 40,40 50,50 50,50 40,40 40))');
```

- Difference with a value

  `minusValue(tnpoint,base): tnpoint`

```
SELECT minusValue(tnpoint '[NPoint(2, 0.3)@2012-01-01, NPoint(2, 0.7)@2012-01-03]',
  'NPoint(2, 0.5)');
-- {[NPoint(2,0.3)@2012-01-01, NPoint(2,0.5)@2012-01-02),
  (NPoint(2,0.5)@2012-01-02, NPoint(2,0.7)@2012-01-03]}
```

- Difference with a geometry

  `minusGeometry(tnpoint,geometry): tnpoint`

```
SELECT minusGeometry(tnpoint '[NPoint(2, 0.3)@2012-01-01, NPoint(2, 0.7)@2012-01-03]',
  'Polygon((40 40,40 50,50 50,50 40,40 40))');
-- {(NPoint(2,0.342593)@2012-01-01 05:06:40.364673+01,
  NPoint(2,0.7)@2012-01-03 00:00:00+01]}
```

- Traditional comparison operators

  `tnpoint = tnpoint: boolean`

  `tnpoint <> tnpoint: boolean`

  `tnpoint < tnpoint: boolean`

  `tnpoint > tnpoint: boolean`

  `tnpoint <= tnpoint: boolean`

  `tnpoint >= tnpoint: boolean`

```
SELECT tnpoint '{[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-02),
  [NPoint(1, 0.3)@2001-01-02, NPoint(1, 0.5)@2001-01-03]}' =
  tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.5)@2001-01-03]';
-- true
SELECT tnpoint '{[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.5)@2001-01-03]}' <>
  tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.5)@2001-01-03]';
-- false
SELECT tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.5)@2001-01-03]' <
  tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.6)@2001-01-03]';
-- true
```

- Temporal comparison operators

  `tnpoint #= tnpoint: tbool`

  `tnpoint #<> tnpoint: tbool`

```
SELECT tnpoint '[NPoint(1, 0.2)@2012-01-01, NPoint(1, 0.4)@2012-01-03)' #=
  npoint 'NPoint(1, 0.3)';
-- {[f@2012-01-01, t@2012-01-02], (f@2012-01-02, f@2012-01-03)}
SELECT tnpoint '[NPoint(1, 0.2)@2012-01-01, NPoint(1, 0.8)@2012-01-03)' #<>
  tnpoint '[NPoint(1, 0.3)@2012-01-01, NPoint(1, 0.7)@2012-01-03)';
-- {[t@2012-01-01, f@2012-01-02], (t@2012-01-02, t@2012-01-03)}
```

- Ever and always equal operators

  `tnpoint ?= tnpoint: boolean`

  `tnpoint &= tnpoint: boolean`

```
SELECT tnpoint '[Npoint(1, 0.2)@2012-01-01, Npoint(1, 0.4)@2012-01-04)' ?= Npoint(1, 0.3);
-- true
SELECT tnpoint '[Npoint(1, 0.2)@2012-01-01, Npoint(1, 0.2)@2012-01-04)' &= Npoint(1, 0.2);
-- true
```

- Relative position operators

  `tnpoint << tnpoint: boolean`

  `tnpoint &< tnpoint: boolean`

  `tnpoint >> tnpoint: boolean`

  `tnpoint &> tnpoint: boolean`

  `tnpoint <<| tnpoint: boolean`

  `tnpoint &<| tnpoint: boolean`

  `tnpoint |>> tnpoint: boolean`

  `tnpoint |&> tnpoint: boolean`

  `tnpoint <<# tnpoint: boolean`

  `tnpoint &<# tnpoint: boolean`

  `tnpoint #>> tnpoint: boolean`

  `tnpoint |&> tnpoint: boolean`

```
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]' <<
  npoint 'NPoint(1, 0.2)'
-- false
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]' <<|
```

```
  stbox(npoint 'NPoint(1, 0.5)')
-- false
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]' &>
  npoint 'NPoint(1, 0.3)'::geometry
-- true
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]' >>#
  tnpoint '[NPoint(1, 0.3)@2000-01-03, NPoint(1, 0.5)@2000-01-05]'
-- true
```

- Topological operators

  `tnpoint && tnpoint: boolean`

  `tnpoint <@ tnpoint: boolean`

  `tnpoint @> tnpoint: boolean`

  `tnpoint ~= tnpoint: boolean`

  `tnpoint -|- tnpoint: boolean`

```
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]' &&
  npoint 'NPoint(1, 0.5)'
-- true
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]' @>
  stbox(npoint 'NPoint(1, 0.5)')
-- true
SELECT npoint 'NPoint(1, 0.5)'::geometry <@
  tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-02]'
-- true
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-03]' ~=
  tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.35)@2000-01-02,
  NPoint(1, 0.5)@2000-01-03]'
-- true
```

- Get the smallest distance ever between the two arguments

  `tgeompoint |=| tnpoint: float`

```
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-03]' |=|
  npoint 'NPoint(1, 0.2)';
-- 2.34988300875063
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-03]' |=|
  geometry 'Linestring(2 2,2 1,3 1)';
-- 82.2059262761477
```

- Get the temporal distance

  `tgeompoint <-> tnpoint: tfloat`

```
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-03]' <->
  npoint 'NPoint(1, 0.2)';
-- [2.34988300875063@2000-01-02 00:00:00+01, 2.34988300875063@2000-01-03 00:00:00+01]
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-03]' <->
  geometry 'Point(50 50)';
-- [25.0496666945044@2000-01-01 00:00:00+01, 26.4085688426232@2000-01-03 00:00:00+01]
SELECT tnpoint '[NPoint(1, 0.3)@2000-01-01, NPoint(1, 0.5)@2000-01-03]' <->
  tnpoint '[NPoint(1, 0.3)@2000-01-02, NPoint(1, 0.5)@2000-01-04]'
-- [2.34988300875063@2000-01-02 00:00:00+01, 2.34988300875063@2000-01-03 00:00:00+01]
```

- Possible spatial relationships

  `contains(geometry,tnpoint): boolean`

  `disjoint({geometry,npoint,tnpoint},{geometry,npoint,tnpoint}): boolean`

  `intersects({geometry,npoint,tnpoint},{geometry,npoint,tnpoint}): boolean`

  `touches({geometry,npoint,tnpoint},{geometry,npoint,tnpoint}): boolean`

  `dwithin({geometry,npoint,tnpoint},{geometry,npoint,tnpoint},float): boolean`

```
SELECT contains(geometry 'Polygon((0 0,0 50,50 50,50 0,0 0))',
  tnpoint '[NPoint(1, 0.1)@2012-01-01, NPoint(1, 0.3)@2012-01-03)');
-- false
SELECT disjoint(npoint 'NPoint(2, 0.0)',
  tnpoint '[NPoint(1, 0.1)@2012-01-01, NPoint(1, 0.3)@2012-01-03)');
-- true
SELECT intersects(tnpoint '[NPoint(1, 0.1)@2012-01-01, NPoint(1, 0.3)@2012-01-03)',
  tnpoint '[NPoint(2, 0.0)@2012-01-01, NPoint(2, 1)@2012-01-03)');
-- false
```

- Temporal spatial relationships

  `tcontains(geometry,tnpoint): boolean`

  `tdisjoint({geometry,npoint,tnpoint},{geometry,npoint,tnpoint}): boolean`

  `tintersects({geometry,npoint,tnpoint},{geometry,npoint,tnpoint}): boolean`

  `ttouches({geometry,npoint,tnpoint},{geometry,npoint,tnpoint}): boolean`

  `tdwithin({geometry,npoint,tnpoint},{geometry,npoint,tnpoint},float): boolean`

```
SELECT tdisjoint(geometry 'Polygon((0 0,0 50,50 50,50 0,0 0))',
  tnpoint '[NPoint(1, 0.1)@2012-01-01, NPoint(1, 0.3)@2012-01-03)');
-- {[t@2012-01-01 00:00:00+01, t@2012-01-03 00:00:00+01)}
SELECT tdwithin(tnpoint '[NPoint(1, 0.3)@2012-01-01, NPoint(1, 0.5)@2012-01-03)',
  tnpoint '[NPoint(1, 0.5)@2012-01-01, NPoint(1, 0.3)@2012-01-03)', 1);
-- {[t@2012-01-01 00:00:00+01, t@2012-01-01 22:35:55.379053+01],
  (f@2012-01-01 22:35:55.379053+01, t@2012-01-02 01:24:04.620946+01,
  t@2012-01-03 00:00:00+01)}
```

6.7 Aggregate Functions

The three aggregate functions for temporal network points are illustrated next.

- Temporal count

  `tcount(tnpoint): {tint_instset,tint_seqset}`

```
WITH Temp(temp) AS (
SELECT tnpoint '[NPoint(1, 0.1)@2012-01-01, NPoint(1, 0.3)@2012-01-03)' UNION
SELECT tnpoint '[NPoint(1, 0.2)@2012-01-02, NPoint(1, 0.4)@2012-01-04)' UNION
SELECT tnpoint '[NPoint(1, 0.3)@2012-01-03, NPoint(1, 0.5)@2012-01-05)' )
SELECT tcount(Temp)
FROM Temp
-- {[1@2012-01-01, 2@2012-01-02, 1@2012-01-04, 1@2012-01-05)}
```

- Window count

  `wcount(tnpoint): {tint_instset,tint_seqset}`

```
WITH Temp(temp) AS (
SELECT tnpoint '[NPoint(1, 0.1)@2012-01-01, NPoint(1, 0.3)@2012-01-03)' UNION
SELECT tnpoint '[NPoint(1, 0.2)@2012-01-02, NPoint(1, 0.4)@2012-01-04)' UNION
SELECT tnpoint '[NPoint(1, 0.3)@2012-01-03, NPoint(1, 0.5)@2012-01-05)' )
SELECT wcount(Temp, '1 day')
FROM Temp
-- {[1@2012-01-01, 2@2012-01-02, 3@2012-01-03, 2@2012-01-04, 1@2012-01-05,
  1@2012-01-06)}
```

- Temporal centroid

  `tcentroid(tnpoint): tgeompoint`

```
WITH Temp(temp) AS (
SELECT tnpoint '[NPoint(1, 0.1)@2012-01-01, NPoint(1, 0.3)@2012-01-03)' UNION
SELECT tnpoint '[NPoint(1, 0.2)@2012-01-01, NPoint(1, 0.4)@2012-01-03)' UNION
SELECT tnpoint '[NPoint(1, 0.3)@2012-01-01, NPoint(1, 0.5)@2012-01-03)' )
SELECT astext(tcentroid(Temp))
FROM Temp
-- {[POINT(72.451531682218 76.5231414472853)@2012-01-01,
  POINT(55.7001249027598 72.9552602410653)@2012-01-03)}
```

6.8 Indexing of Temporal Network Points

GiST and SP-GiST indexes can be created for table columns of temporal networks points. An example of index creation is follows:

```
CREATE INDEX Trips_Trip_SPGist_Idx ON Trips USING SPGist(Trip);
```

The GiST and SP-GiST indexes store the bounding box for the temporal network points, which is an stbox and thus stores the absolute coordinates of the underlying space.

A GiST or SP-GiST index can accelerate queries involving the following operators:

- `<<`, `&<`, `&>`, `>>`, `<<|`, `&<|`, `|&>`, `|>>`, which only consider the spatial dimension in temporal network points,
- `<<#`, `&<#`, `#&>`, `#>>`, which only consider the time dimension in temporal network points,
- `&&`, `@>`, `<@`, `~=`, `-|-`, and `|=|` , which consider as many dimensions as they are shared by the indexed column and the query argument.

These operators work on bounding boxes, not the entire values.

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

A.1.4 Modification Functions

- shift: Shift the time value by an interval
- tscale: Scale the time value to an interval
- shiftTscale: Shift and scale the time value with the intervals
- round: Round the bounds of a float range to a number of decimal places

A.1.5 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.1.6 Set Operators

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

A.1.7 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.1.8 Distance Operators

- <->: Get the smallest distance ever in an interval
- |=|: Get the smallest distance ever in number of seconds

A.1.9 Aggregate Functions

- tcount: Temporal count
- extent: Bounding period
- extent: Bounding range
- 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
- round: 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 Aggregate Functions

- extent: Bounding box extent

A.2.7 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.2.8 Set Operators

- +: Union of the bounding boxes
- *: Intersection of the bounding boxes

A.2.9 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.2.10 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

- ttype_inst: Constructor for temporal types of instant subtype
- ttype_instset: Constructor for temporal types of instant set subtype
- ttype_seq: Constructor for temporal types of sequence subtype
- ttype_seqset: Constructor for temporal types of sequence set subtype

A.3.2 Casting

- ttype::period: Cast a temporal value to a period
- tnumber::range: Cast a temporal number to a range
- tnumber::tbox: Cast a temporal number to a `tbox`
- ttype::period: Cast a temporal point to an `stbox`
- 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 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
- minInstant: Get the instant with the minimum value
- maxInstant: Get the instant with 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.3.4 Modification Functions

- ttype_inst, ttype_instset, ttype_seq, ttype_seqset: 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.3.5 Restriction Functions

A.3.5.1 Selection 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.3.5.2 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.3.6 Comparison Operators

A.3.6.1 Traditional 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.3.6.2 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.3.6.3 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.3.7 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.3.8 Boolean Operators

- &: Temporal and
- |: Temporal or
- ~: Temporal not

A.3.9 Text Functions and Operators

- ||: Temporal text concatenation
- upper: Transform to uppercase
- lower: Transform to lowercase

A.3.10 Spatial Functions and Operators

A.3.10.1 Input/Output 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
- tgeompointFromText: Input a temporal geometry point from a Well-Known Text (WKT) representation
- tgeogpointFromText: Input a temporal geography point from a Well-Known Text (WKT) representation
- tgeompointFromEWKT: Input a temporal geometry point from an Extended Well-Known Text (EWKT) representation
- tgeogpointFromEWKT: Input a temporal geography point from an Extended Well-Known Text (EWKT) representation
- tgeompointFromMFJSON: Input a temporal geometry point from a Moving Features JSON representation
- tgeogpointFromMFJSON: Input a temporal geography geometry point from a Moving Features JSON representation
- tgeompointFromBinary: Input a temporal geometry point from a Well-Known Binary (WKB) representation
- tgeogpointFromBinary: Input a temporal geography point from a Well-Known Binary (WKB) representation
- tgeompointFromEWKB: Input a temporal geometry point from an Extended Well-Known Binary (EWKB) representation
- tgeogpointFromEWKB: Input a temporal geography point from an Extended Well-Known Binary (EWKB) representation
- tgeompointFromHexEWKB: Input a temporal geometry point from an Hexadecimal Extended Well-Known Binary (EWKB) representation as text
- tgeogpointFromHexEWKB: Input a temporal geography point from an Hexadecimal Extended Well-Known Binary (EWKB) representation as text

A.3.10.2 Spatial Reference System Functions

- SRID: Get the spatial reference identifier
- setSRID: Set the spatial reference identifier
- transform: Transform to a different spatial reference

A.3.10.3 Accessor Functions

- 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
- isSimple: Returns true if the temporal point does not spatially self-intersect
- 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
- bearing: Get the temporal bearing

A.3.10.4 Manipulation Functions

- round: Round the coordinate values to a number of decimal places
- makeSimple: Returns an array of fragments of the temporal point which are simple
- 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
- asMVTGeom: Transform a temporal geometric point into the coordinate space of a Mapbox Vector Tile

A.3.10.5 Distance Functions and Operators

- |=|: Get the smallest distance ever
- nearestApproachInstant: Get the instant of the first temporal point at which the two arguments are at the nearest distance
- shortestLine: Get the line connecting the nearest approach point
- <->: Get the temporal distance

A.3.10.6 Possible Spatial Relationships

- contains: May contain
- disjoint: May be disjoint
- dwithin: May be at distance within
- intersects: May intersect
- touches: May touch

A.3.10.7 Temporal Spatial Relationships

- tcontains: Temporal contains
- tdisjoint: Temporal disjoint
- tdwithin: Temporal distance within
- tintersects: Temporal intersects
- ttouches: Temporal touches

A.3.11 Similarity Functions

- frechetDistance: Get the discrete Fréchet distance between two temporal values
- frechetDistancePath: Get the correspondence pairs between two temporal values with respect to the discrete Fréchet distance
- dynamicTimeWarp: Get the Dynamic Time Warp distance between two temporal values
- dynamicTimeWarpPath: Get the correspondence pairs between two temporal values with respect to the Dynamic Time Warp distance

A.3.12 Multidimensional Tiling

A.3.12.1 Bucket Functions

- bucketList: Returns a set of couples (index, bucket) that cover the range or period with buckets of the same width or duration aligned with the origin
- valueBucket: Returns the start value of the bucket that contains the input number.
- rangeBucket: Returns the range in the bucket space that contains the input number
- timeBucket: Returns the start timestamp of the bucket that contains the input timestamp.
- periodBucket: Returns a period in the bucket space that contains the input timestamp

A.3.12.2 Grid Functions

- multidimGrid: Returns a set of couples (index, tile) that covers the box with multidimensional tiles of the same size and duration.
- multidimTile: Returns the tile of the multidimensional grid that contains the value and the timestamp

A.3.12.3 Split Functions

- valueSplit: Fragment the temporal number with respect to value buckets
- timeSplit: Fragment the temporal value with respect to time buckets
- valueTimeSplit: Fragment the temporal number with respect to the tiles in a value-time grid
- spaceSplit: Fragment the temporal point with respect to tiles in a spatial grid
- spaceTimeSplit: Fragment the temporal point with respect to tiles in a spatiotemporal grid

A.3.13 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.3.14 Utility Functions

- mobilitydb_version: Get the version of the MobilityDB extension
- mobilitydb_full_version: Get the versions of the MobilityDB extension and its dependencies

A.4 Functions and Operators for Temporal Network Points

A.4.1 Static Network Types

A.4.1.1 Constructor Functions

- npoint: Constructor for network points
- nsegment: Constructor for network segments

A.4.1.2 Accessor Functions

- route: Get the route identifier
- getPosition: Get the position fraction
- startPosition: Get the start position
- endPosition: Get the end position
- SRID: Get the spatial reference identifier

A.4.1.3 Modification Functions

- round: Round the position(s) of the network point or the network segment to the number of decimal places

A.4.1.4 Spatial Functions

- ::: Cast a network point to a geometry
- ::: Cast a geometry to a network point
- equals: Spatial equality for network points

A.4.1.5 Comparison Operators

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

A.4.2 Temporal Network Points

A.4.2.1 Constructors

- tnpoint_inst: Constructor for temporal network points of instant subtype
- tnpoint_instset: Constructors for temporal network points of instant set subtype
- tnpoint_seq: Constructor for temporal network points of sequence subtype
- tnpoint_seqset: Constructor for temporal network points of sequence set subtype

A.4.2.2 Casting

- tnpoint::tgeompoint: Cast a temporal network point to a temporal geometry point
- tgeompoint::tnpoint: Cast a temporal geometry point to a temporal network point

A.4.2.3 Functions and Operators

- tnpoint_inst, tnpoint_instset, tnpoint_seq, tnpoint_seqset: Transform a temporal network to another subtype
- round: Round the fraction of the temporal network point to the number of decimal places
- getValues: Get the values
- valueAtTimestamp: Get the value at a timestamp
- length: Get the length traversed by the temporal network point
- cumulativeLength: Get the cumulative length traversed by the temporal network point
- speed: Get the speed of the temporal network point in units per second

- stbox: Construct the bounding box from a npoint and, optionally, a timestamp or a period
- twCentroid: Get the time-weighted centroid
- azimuth: Get the temporal azimuth
- nearestApproachInstant: Get the instant of the first temporal network point at which the two arguments are at the nearest distance
- nearestApproachDistance: Smallest distance ever between the two arguments
- shortestLine: Get the line connecting the nearest approach point between the two arguments
- atValue: Restrict to a value
- atGeometry: Restrict to a geometry
- minusValue: Difference with a value
- minusGeometry: Difference with a geometry
- =, <>, <, >, <=, >=: Traditional comparison operators
- #=, #<>: Temporal comparison operators
- ?=, &=: Ever and always equal operators
- <<, &<, >>, &>, <<|, &<|, |>>, |&>, <<#, &<#, #>>, |&>: Relative position operators
- &&, <@, @>, ~=, -|-: Topological operators
- |=|: Get the smallest distance ever between the two arguments
- <->: Get the temporal distance
- contains, disjoint, intersects, touches, dwithin: Possible spatial relationships
- tcontains, tdisjoint, tintersects, ttouches, tdwithin: Temporal spatial relationships

A.4.2.4 Aggregate Functions

- tcount: Temporal count
- tcount: Window count
- tcentroid: Temporal centroid

Appendix B

Synthetic Data Generator

In many circumstances, it is necessary to have a test dataset to evaluate alternative implementation approaches or to perform benchmarks. It is often required that such a data set have particular requirements in size or in the intrinsic characteristics of its data. Even if a real-world dataset could be available, it may be not ideal for such experiments for multiple reasons. Therefore, a synthetic data generator that could be customized to produce data according to the given requirements is often the best solution. Obviously, experiments with synthetic data should be complemented with experiments with real-world data to have a thorough understanding of the problem at hand.

MobilityDB provides a simple synthetic data generator that can be used for such purposes. In particular, such a data generator was used for generating the database used for the regression tests in MobilityDB. The data generator is programmed in PL/pgSQL so it can be easily customized. It is located in the directory `datagen` in the repository. In this appendix, we briefly introduce the basic functionality of the generator. We first list the functions generating random values for the various PostgreSQL, PostGIS, and MobilityDB data types, and then give examples how to use these functions for generating tables of such values. The parameters of the functions are not specified, please refer to the source files where detailed explanations about the various parameters can be found.

B.1 Generator for PostgreSQL Types

- `random_bool`: Generate a random boolean
- `random_int`: Generate a random integer
- `random_int_array`: Generate a random array of integers
- `random_intrange`: Generate a random integer range
- `random_float`: Generate a random float
- `random_float_array`: Generate a random array of floats
- `random_floatrange`: Generate a random float range
- `random_text`: Generate a random text
- `random_timestamptz`: Generate a random timestamp with time zone
- `random_timestamptz_array`: Generate a random array of timestamps with time zone
- `random_minutes`: Generate a random interval of minutes
- `random_tstzrange`: Generate a random timestamp with time zone range
- `random_tstzrange_array`: Generate a random array of timestamp with time zone ranges

B.2 Generator for PostGIS Types

- random_geom_point: Generate a random 2D geometric point
- random_geom_point3D: Generate a random 3D geometric point
- random_geog_point: Generate a random 2D geographic point
- random_geog_point3D: Generate a random 3D geographic point
- random_geom_point_array: Generate a random array of 2D geometric points
- random_geom_point3D_array: Generate a random array of 3D geometric points
- random_geog_point_array: Generate a random array of 2D geographic points
- random_geog_point3D_array: Generate a random array of 3D geographic points
- random_geom_linestring: Generate a random geometric 2D linestring
- random_geom_linestring3D: Generate a random geometric 3D linestring
- random_geog_linestring: Generate a random geographic 2D linestring
- random_geog_linestring3D: Generate a random geographic 3D linestring
- random_geom_polygon: Generate a random geometric 2D polygon without holes
- random_geom_polygon3D: Generate a random geometric 3D polygon without holes
- random_geog_polygon: Generate a random geographic 2D polygon without holes
- random_geog_polygon3D: Generate a random geographic 3D polygon without holes
- random_geom_multipoint: Generate a random geometric 2D multipoint
- random_geom_multipoint3D: Generate a random geometric 3D multipoint
- random_geog_multipoint: Generate a random geographic 2D multipoint
- random_geog_multipoint3D: Generate a random geographic 3D multipoint
- random_geom_multilinestring: Generate a random geometric 2D multilinestring
- random_geom_multilinestring3D: Generate a random geometric 3D multilinestring
- random_geog_multilinestring: Generate a random geographic 2D multilinestring
- random_geog_multilinestring3D: Generate a random geographic 3D multilinestring
- random_geom_multipolygon: Generate a random geometric 2D multipolygon without holes
- random_geom_multipolygon3D: Generate a random geometric 3D multipolygon without holes
- random_geog_multipolygon: Generate a random geographic 2D multipolygon without holes
- random_geog_multipolygon3D: Generate a random geographic 3D multipolygon without holes

B.3 Generator for MobilityDB Time and Box Types

- `random_timestampset`: Generate a random `timestampset`
- `random_period`: Generate a random `period`
- `random_period_array`: Generate a random array of `period` values
- `random_periodset`: Generate a random `periodset`
- `random_tbox`: Generate a random `tbox`
- `random_stbox`: Generate a random 2D `stbox`
- `random_stbox3D`: Generate a random 3D `stbox`
- `random_geodstbox`: Generate a random 2D geodetic `stbox`
- `random_geodstbox3D`: Generate a random 3D geodetic `stbox`

B.4 Generator for MobilityDB Temporal Types

- `random_tbool_inst`: Generate a random temporal Boolean of instant subtype
- `random_tint_inst`: Generate a random temporal integer of instant subtype
- `random_tfloat_inst`: Generate a random temporal float of instant subtype
- `random_ttext_inst`: Generate a random temporal text of instant subtype
- `random_tgeompoint_inst`: Generate a random temporal geometric 2D point of instant subtype
- `random_tgeompoint3D_inst`: Generate a random temporal geometric 3D point of instant subtype
- `random_tgeogpoint_inst`: Generate a random temporal geographic 2D point of instant subtype
- `random_tgeogpoint3D_inst`: Generate a random temporal geographic 3D point of instant subtype
- `random_tbool_instset`: Generate a random temporal Boolean of instant set subtype
- `random_tint_instset`: Generate a random temporal integer of instant set subtype
- `random_tfloat_instset`: Generate a random temporal float of instant set subtype
- `random_ttext_instset`: Generate a random temporal text of instant set subtype
- `random_tgeompoint_instset`: Generate a random temporal geometric 2D point of instant set subtype
- `random_tgeompoint3D_instset`: Generate a random temporal geometric 3D point of instant set subtype
- `random_tgeogpoint_instset`: Generate a random temporal geographic 2D point of instant set subtype
- `random_tgeogpoint3D_instset`: Generate a random temporal geographic 3D point of instant set subtype
- `random_tbool_seq`: Generate a random temporal Boolean of sequence subtype
- `random_tint_seq`: Generate a random temporal integer of sequence subtype
- `random_tfloat_seq`: Generate a random temporal float of sequence subtype
- `random_ttext_seq`: Generate a random temporal text of sequence subtype
- `random_tgeompoint_seq`: Generate a random temporal geometric 2D point of sequence subtype
- `random_tgeompoint3D_seq`: Generate a random temporal geometric 3D point of sequence subtype

- `random_tgeogpoint_seq`: Generate a random temporal geographic 2D point of sequence subtype
- `random_tgeogpoint3D_seq`: Generate a random temporal geographic 3D point of sequence subtype
- `random_tbool_seqset`: Generate a random temporal Boolean of sequence set subtype
- `random_tint_seqset`: Generate a random temporal integer of sequence set subtype
- `random_tfloat_seqset`: Generate a random temporal float of sequence set subtype
- `random_ttext_seqset`: Generate a random temporal text of sequence set subtype
- `random_tgeompoint_seqset`: Generate a random temporal geometric 2D point of sequence set subtype
- `random_tgeompoint3D_seqset`: Generate a random temporal geometric 3D point of sequence set subtype
- `random_tgeogpoint_seqset`: Generate a random temporal geographic 2D point of sequence set subtype
- `random_tgeogpoint3D_seqset`: Generate a random temporal geographic 3D point of sequence set subtype

B.5 Generation of Tables with Random Values

The files `create_test_tables_temporal.sql` and `create_test_tables_tpoint.sql` provide usage examples for the functions generating random values listed above. For example, the first file defines the following function.

```
CREATE OR REPLACE FUNCTION create_test_tables_temporal(size int DEFAULT 100)
RETURNS text AS $$
DECLARE
  perc int;
BEGIN
perc := size * 0.01;
IF perc < 1 THEN perc := 1; END IF;

-- ... Table generation ...

RETURN 'The End';
END;
$$ LANGUAGE 'plpgsql';
```

The function has a `size` parameter that defines the number of rows in the tables. If not provided, it creates by default tables of 100 rows. The function defines a variable `perc` that computes the 1% of the size of the tables. This parameter is used, for example, for generating tables having 1% of null values. We illustrate next some of the commands generating tables.

The creation of a table `tbl_float` containing random `float` values in the range [0,100] with 1% of null values is given next.

```
CREATE TABLE tbl_float AS
/* Add perc NULL values */
SELECT k, NULL AS f
FROM generate_series(1, perc) AS k UNION
SELECT k, random_float(0, 100)
FROM generate_series(perc+1, size) AS k;
```

The creation of a table `tbl_tbox` containing random `tbox` values where the bounds for values are in the range [0,100] and the bounds for timestamps are in the range [2001-01-01, 2001-12-31] is given next.

```
CREATE TABLE tbl_tbox AS
/* Add perc NULL values */
SELECT k, NULL AS b
FROM generate_series(1, perc) AS k UNION
SELECT k, random_tbox(0, 100, '2001-01-01', '2001-12-31', 10, 10)
FROM generate_series(perc+1, size) AS k;
```

The creation of a table `tbl_floatrange` containing random `floatrange` values where the bounds for values are in the range [0,100] and the maximum difference between the lower and the upper bounds is 10 is given next.

```
CREATE TABLE tbl_floatrange AS
/* Add perc NULL values */
SELECT k, NULL AS f
FROM generate_series(1, perc) AS k UNION
SELECT k, random_floatrange(0, 100, 10)
FROM generate_series(perc+1, size) AS k;
```

The creation of a table `tbl_timestampset` containing random `timestampset` values having between 5 and 10 timestamps where the timestamps are in the range [2001-01-01, 2001-12-31] and the maximum interval between consecutive timestamps is 10 minutes is given next.

```
CREATE TABLE tbl_timestampset AS
/* Add perc NULL values */
SELECT k, NULL AS ts
FROM generate_series(1, perc) AS k UNION
SELECT k, random_timestampset('2001-01-01', '2001-12-31', 10, 5, 10)
FROM generate_series(perc+1, size) AS k;
```

The creation of a table `tbl_period` containing random `period` values where the timestamps are in the range [2001-01-01, 2001-12-31] and the maximum difference between the lower and the upper bounds is 10 minutes is given next.

```
CREATE TABLE tbl_period AS
/* Add perc NULL values */
SELECT k, NULL AS p
FROM generate_series(1, perc) AS k UNION
SELECT k, random_period('2001-01-01', '2001-12-31', 10)
FROM generate_series(perc+1, size) AS k;
```

The creation of a table `tbl_geom_point` containing random `geometry` 2D point values, where the x and y coordinates are in the range [0, 100] and in SRID 3812 is given next.

```
CREATE TABLE tbl_geom_point AS
SELECT 1 AS k, geometry 'SRID=3812;point empty' AS g UNION
SELECT k, random_geom_point(0, 100, 0, 100, 3812)
FROM generate_series(2, size) k;
```

Notice that the table contains an empty point value. If the SRID is not given it is set by default to 0.

The creation of a table `tbl_geog_point3D` containing random `geography` 3D point values, where the x, y, and z coordinates are, respectively, in the ranges [-10, 32], [35, 72], and [0, 1000] and in SRID 7844 is given next.

```
CREATE TABLE tbl_geog_point3D AS
SELECT 1 AS k, geography 'SRID=7844;pointZ empty' AS g UNION
SELECT k, random_geog_point3D(-10, 32, 35, 72, 0, 1000, 7844)
FROM generate_series(2, size) k;
```

Notice that latitude and longitude values are chosen to approximately cover continental Europe. If the SRID is not given it is set by default to 4326.

The creation of a table `tbl_geom_linestring` containing random `geometry` 2D linestring values having between 5 and 10 vertices, where the x and y coordinates are in the range [0, 100] and in SRID 3812 and the maximum difference between consecutive coordinate values is 10 units in the underlying SRID is given next.

```
CREATE TABLE tbl_geom_linestring AS
SELECT 1 AS k, geometry 'linestring empty' AS g UNION
SELECT k, random_geom_linestring(0, 100, 0, 100, 10, 5, 10, 3812)
FROM generate_series(2, size) k;
```

The creation of a table `tbl_geom_linestring` containing random `geometry` 2D linestring values having between 5 and 10 vertices, where the x and y coordinates are in the range [0, 100] and the maximum difference between consecutive coordinate values is 10 units in the underlying SRID is given next.

```
CREATE TABLE tbl_geom_linestring AS
SELECT 1 AS k, geometry 'linestring empty' AS g UNION
SELECT k, random_geom_linestring(0, 100, 0, 100, 10, 5, 10)
FROM generate_series(2, size) k;
```

The creation of a table `tbl_geom_polygon3D` containing random `geometry` 3D polygon values without holes, having between 5 and 10 vertices, where the x, y, and z coordinates are in the range [0, 100] and the maximum difference between consecutive coordinate values is 10 units in the underlying SRID is given next.

```
CREATE TABLE tbl_geom_polygon3D AS
SELECT 1 AS k, geometry 'polygon Z empty' AS g UNION
SELECT k, random_geom_polygon3D(0, 100, 0, 100, 0, 100, 10, 5, 10)
FROM generate_series(2, size) k;
```

The creation of a table `tbl_geom_multipoint` containing random `geometry` 2D multipoint values having between 5 and 10 points, where the x and y coordinates are in the range [0, 100] and the maximum difference between consecutive coordinate values is 10 units in the underlying SRID is given next.

```
CREATE TABLE tbl_geom_multipoint AS
SELECT 1 AS k, geometry 'multipoint empty' AS g UNION
SELECT k, random_geom_multipoint(0, 100, 0, 100, 10, 5, 10)
FROM generate_series(2, size) k;
```

The creation of a table `tbl_geog_multilinestring` containing random `geography` 2D multilinestring values having between 5 and 10 linestrings, each one having between 5 and 10 vertices, where the x and y coordinates are, respectively, in the ranges [-10, 32] and [35, 72], and the maximum difference between consecutive coordinate values is 10 is given next.

```
CREATE TABLE tbl_geog_multilinestring AS
SELECT 1 AS k, geography 'multilinestring empty' AS g UNION
SELECT k, random_geog_multilinestring(-10, 32, 35, 72, 10, 5, 10, 5, 10)
FROM generate_series(2, size) k;
```

The creation of a table `tbl_geometry3D` containing random `geometry` 3D values of various types is given next. This function requires that the tables for the various geometry types have been created previously.

```
CREATE TABLE tbl_geometry3D (
  k serial PRIMARY KEY,
  g geometry);
INSERT INTO tbl_geometry3D(g)
(SELECT g FROM tbl_geom_point3D ORDER BY k LIMIT (size * 0.1)) UNION ALL
(SELECT g FROM tbl_geom_linestring3D ORDER BY k LIMIT (size * 0.1)) UNION ALL
(SELECT g FROM tbl_geom_polygon3D ORDER BY k LIMIT (size * 0.2)) UNION ALL
(SELECT g FROM tbl_geom_multipoint3D ORDER BY k LIMIT (size * 0.2)) UNION ALL
(SELECT g FROM tbl_geom_multilinestring3D ORDER BY k LIMIT (size * 0.2)) UNION ALL
(SELECT g FROM tbl_geom_multipolygon3D ORDER BY k LIMIT (size * 0.2));
```

The creation of a table `tbl_tbool_inst` containing random `tbool` values of instant subtype where the timestamps are in the range [2001-01-01, 2001-12-31] is given next.

```
CREATE TABLE tbl_tbool_inst AS
/* Add perc NULL values */
SELECT k, NULL AS inst
FROM generate_series(1, perc) AS k UNION
SELECT k, random_tbool_inst('2001-01-01', '2001-12-31')
FROM generate_series(perc+1, size) k;
/* Add perc duplicates */
UPDATE tbl_tbool_inst t1
SET inst = (SELECT inst FROM tbl_tbool_inst t2 WHERE t2.k = t1.k+perc)
WHERE k in (SELECT i FROM generate_series(1 + 2*perc, 3*perc) i);
/* Add perc rows with the same timestamp */
UPDATE tbl_tbool_inst t1
SET inst = (SELECT tboolinst(random_bool(), getTimestamp(inst))
  FROM tbl_tbool_inst t2 WHERE t2.k = t1.k+perc)
WHERE k in (SELECT i FROM generate_series(1 + 4*perc, 5*perc) i);
```

As can be seen above, the table has a percentage of null values, of duplicates, and of rows with the same timestamp.

The creation of a table `tbl_tint_instset` containing random `tint` values of instant set subtype having between 5 and 10 timestamps where the integer values are in the range [0, 100], the timestamps are in the range [2001-01-01, 2001-12-31], the maximum difference between two consecutive values is 10, and the maximum interval between two consecutive instants is 10 minutes is given next.

```
CREATE TABLE tbl_tint_instset AS
/* Add perc NULL values */
SELECT k, NULL AS ti
FROM generate_series(1, perc) AS k UNION
SELECT k, random_tint_instset(0, 100, '2001-01-01', '2001-12-31', 10, 10, 5, 10) AS ti
FROM generate_series(perc+1, size) k;
/* Add perc duplicates */
UPDATE tbl_tint_instset t1
SET ti = (SELECT ti FROM tbl_tint_instset t2 WHERE t2.k = t1.k+perc)
WHERE k in (SELECT i FROM generate_series(1 + 2*perc, 3*perc) i);
/* Add perc rows with the same timestamp */
UPDATE tbl_tint_instset t1
SET ti = (SELECT ti + random_int(1, 2) FROM tbl_tint_instset t2 WHERE t2.k = t1.k+perc)
WHERE k in (SELECT i FROM generate_series(1 + 4*perc, 5*perc) i);
/* Add perc rows that meet */
UPDATE tbl_tint_instset t1
SET ti = (SELECT shift(ti, endTimestamp(ti)-startTimestamp(ti))
```

```
  FROM tbl_tint_instset t2 WHERE t2.k = t1.k+perc)
WHERE t1.k in (SELECT i FROM generate_series(1 + 6*perc, 7*perc) i);
/* Add perc rows that overlap */
UPDATE tbl_tint_instset t1
SET ti = (SELECT shift(ti, date_trunc('minute',(endTimestamp(ti)-startTimestamp(ti))/2))
  FROM tbl_tint_instset t2 WHERE t2.k = t1.k+2)
WHERE t1.k in (SELECT i FROM generate_series(1 + 8*perc, 9*perc) i);
```

As can be seen above, the table has a percentage of null values, of duplicates, of rows with the same timestamp, of rows that meet, and of rows that overlap.

The creation of a table `tbl_tfloat_seq` containing random `tfloat` values of sequence subtype having between 5 and 10 timestamps where the float values are in the range [0, 100], the timestamps are in the range [2001-01-01, 2001-12-31], the maximum difference between two consecutive values is 10, and the maximum interval between two consecutive instants is 10 minutes is given next.

```
CREATE TABLE tbl_tfloat_seq AS
/* Add perc NULL values */
SELECT k, NULL AS seq
FROM generate_series(1, perc) AS k UNION
SELECT k, random_tfloat_seq(0, 100, '2001-01-01', '2001-12-31', 10, 10, 5, 10) AS seq
FROM generate_series(perc+1, size) k;
/* Add perc duplicates */
UPDATE tbl_tfloat_seq t1
SET seq = (SELECT seq FROM tbl_tfloat_seq t2 WHERE t2.k = t1.k+perc)
WHERE k in (SELECT i FROM generate_series(1 + 2*perc, 3*perc) i);
/* Add perc tuples with the same timestamp */
UPDATE tbl_tfloat_seq t1
SET seq = (SELECT seq + random_int(1, 2) FROM tbl_tfloat_seq t2 WHERE t2.k = t1.k+perc)
WHERE k in (SELECT i FROM generate_series(1 + 4*perc, 5*perc) i);
/* Add perc tuples that meet */
UPDATE tbl_tfloat_seq t1
SET seq = (SELECT shift(seq, timespan(seq)) FROM tbl_tfloat_seq t2 WHERE t2.k = t1.k+perc)
WHERE t1.k in (SELECT i FROM generate_series(1 + 6*perc, 7*perc) i);
/* Add perc tuples that overlap */
UPDATE tbl_tfloat_seq t1
SET seq = (SELECT shift(seq, date_trunc('minute',timespan(seq)/2))
  FROM tbl_tfloat_seq t2 WHERE t2.k = t1.k+perc)
WHERE t1.k in (SELECT i FROM generate_series(1 + 8*perc, 9*perc) i);
```

The creation of a table `tbl_ttext_seqset` containing random `ttext` values of sequence set subtype having between 5 and 10 sequences, each one having between 5 and 10 timestamps, where the text values have at most 10 characters, the timestamps are in the range [2001-01-01, 2001-12-31], and the maximum interval between two consecutive instants is 10 minutes is given next.

```
CREATE TABLE tbl_ttext_seqset AS
/* Add perc NULL values */
SELECT k, NULL AS ts
FROM generate_series(1, perc) AS k UNION
SELECT k, random_ttext_seqset('2001-01-01', '2001-12-31', 10, 10, 5, 10, 5, 10) AS ts
FROM generate_series(perc+1, size) AS k;
/* Add perc duplicates */
UPDATE tbl_ttext_seqset t1
SET ts = (SELECT ts FROM tbl_ttext_seqset t2 WHERE t2.k = t1.k+perc)
WHERE k in (SELECT i FROM generate_series(1 + 2*perc, 3*perc) i);
/* Add perc tuples with the same timestamp */
UPDATE tbl_ttext_seqset t1
SET ts = (SELECT ts || text 'A' FROM tbl_ttext_seqset t2 WHERE t2.k = t1.k+perc)
```

```
WHERE k in (SELECT i FROM generate_series(1 + 4*perc, 5*perc) i);
/* Add perc tuples that meet */
UPDATE tbl_ttext_seqset t1
SET ts = (SELECT shift(ts, timespan(ts)) FROM tbl_ttext_seqset t2 WHERE t2.k = t1.k+perc)
WHERE t1.k in (SELECT i FROM generate_series(1 + 6*perc, 7*perc) i);
/* Add perc tuples that overlap */
UPDATE tbl_ttext_seqset t1
SET ts = (SELECT shift(ts, date_trunc('minute', timespan(ts)/2))
  FROM tbl_ttext_seqset t2 WHERE t2.k = t1.k+perc)
WHERE t1.k in (SELECT i FROM generate_series(1 + 8*perc, 9*perc) i);
```

The creation of a table `tbl_tgeompoint_instset` containing random `tgeompoint` 2D values of instant set subtype having between 5 and 10 instants, where the x and y coordinates are in the range [0, 100] and in SRID 3812, the timestamps are in the range [2001-01-01, 2001-12-31], the maximum difference between successive coordinates is at most 10 units in the underlying SRID, and the maximum interval between two consecutive instants is 10 minutes is given next.

```
CREATE TABLE tbl_tgeompoint_instset AS
SELECT k, random_tgeompoint_instset(0, 100, 0, 100, '2001-01-01', '2001-12-31',
  10, 10, 5, 10, 3812) AS ti
FROM generate_series(1, size) k;
/* Add perc duplicates */
UPDATE tbl_tgeompoint_instset t1
SET ti = (SELECT ti FROM tbl_tgeompoint_instset t2 WHERE t2.k = t1.k+perc)
WHERE k IN (SELECT i FROM generate_series(1, perc) i);
/* Add perc tuples with the same timestamp */
UPDATE tbl_tgeompoint_instset t1
SET ti = (SELECT round(ti,6) FROM tbl_tgeompoint_instset t2 WHERE t2.k = t1.k+perc)
WHERE k IN (SELECT i FROM generate_series(1 + 2*perc, 3*perc) i);
/* Add perc tuples that meet */
UPDATE tbl_tgeompoint_instset t1
SET ti = (SELECT shift(ti, endTimestamp(ti)-startTimestamp(ti))
  FROM tbl_tgeompoint_instset t2 WHERE t2.k = t1.k+perc)
WHERE t1.k IN (SELECT i FROM generate_series(1 + 4*perc, 5*perc) i);
/* Add perc tuples that overlap */
UPDATE tbl_tgeompoint_instset t1
SET ti = (SELECT shift(ti, date_trunc('minute',(endTimestamp(ti)-startTimestamp(ti))/2))
  FROM tbl_tgeompoint_instset t2 WHERE t2.k = t1.k+2)
WHERE t1.k IN (SELECT i FROM generate_series(1 + 6*perc, 7*perc) i);
```

Finally, the creation of a table `tbl_tgeompoint3D_seqset` containing random `tgeompoint` 3D values of sequence set subtype having between 5 and 10 sequences, each one having between 5 and 10 timestamps, where the x, y, and z coordinates are in the range [0, 100] and in SRID 3812, the timestamps are in the range [2001-01-01, 2001-12-31], the maximum difference between successive coordinates is at most 10 units in the underlying SRID, and the maximum interval between two consecutive instants is 10 minutes is given next.

```
DROP TABLE IF EXISTS tbl_tgeompoint3D_seqset;
CREATE TABLE tbl_tgeompoint3D_seqset AS
SELECT k, random_tgeompoint3D_seqset(0, 100, 0, 100, 0, 100, '2001-01-01', '2001-12-31',
  10, 10, 5, 10, 5, 10, 3812) AS ts
FROM generate_series(1, size) AS k;
/* Add perc duplicates */
UPDATE tbl_tgeompoint3D_seqset t1
SET ts = (SELECT ts FROM tbl_tgeompoint3D_seqset t2 WHERE t2.k = t1.k+perc)
WHERE k IN (SELECT i FROM generate_series(1, perc) i);
/* Add perc tuples with the same timestamp */
UPDATE tbl_tgeompoint3D_seqset t1
SET ts = (SELECT round(ts,3) FROM tbl_tgeompoint3D_seqset t2 WHERE t2.k = t1.k+perc)
WHERE k IN (SELECT i FROM generate_series(1 + 2*perc, 3*perc) i);
```

```
/* Add perc tuples that meet */
UPDATE tbl_tgeompoint3D_seqset t1
SET ts = (SELECT shift(ts, timespan(ts)) FROM tbl_tgeompoint3D_seqset t2 WHERE t2.k = t1.k+ ↩
    perc)
WHERE t1.k IN (SELECT i FROM generate_series(1 + 4*perc, 5*perc) i);
/* Add perc tuples that overlap */
UPDATE tbl_tgeompoint3D_seqset t1
SET ts = (SELECT shift(ts, date_trunc('minute', timespan(ts)/2))
  FROM tbl_tgeompoint3D_seqset t2 WHERE t2.k = t1.k+perc)
WHERE t1.k IN (SELECT i FROM generate_series(1 + 6*perc, 7*perc) i);
```

B.6 Generator for Temporal Network Point Types

- `random_fraction`: Generate a random fraction in the range [0,1]
- `random_npoint`: Generate a random network point
- `random_nsegment`: Generate a random network segment
- `random_tnpoint_inst`: Generate a random temporal network point of instant subtype
- `random_tnpoint_instset`: Generate a random temporal network point of instant set subtype
- `random_tnpoint_seq`: Generate a random temporal network point of sequence subtype
- `random_tnpoint_seqset`: Generate a random temporal network point of sequence set subtype

The file `/datagen/npoint/create_test_tables_tnpoint.sql` provide usage examples for the functions generating random values listed above.

Chapter 7

Index

_
*, 16, 35, 67
+, 16, 34, 66
-, 16, 67
-|-, 17, 18, 36, 110
/, 67
/>>, 38
/&>, 38
::, 10, 11, 29, 30, 45–47, 101, 102, 105, 108, 109
<, 15, 34, 62, 102
<->, 18, 79, 110
<<, 17, 36, 37
<</, 38
<<#, 18, 39
<<|, 37
<=, 15, 34, 62, 102
<>, 15, 34, 62, 102
<@, 17, 35, 110
=, 15, 33, 62, 102, 104, 105
>, 15, 34, 62, 102
>=, 15, 34, 62, 103
>>, 17, 36, 37
?<, 63
?<=, 63
?<>, 63
?=, 63, 109
?>, 63
?>=, 64
@>, 16, 35, 110
#<, 65
#<=, 66
#<>, 65
#=, 65
#>, 66
#>=, 66
#>>, 18, 39
#&>, 18, 39
%<, 64
%<=, 64
%<>, 64
%=, 64
%>, 64
%>=, 65
&, 68
&<, 17, 37
&</, 38
&<#, 18, 39
&<|, 38
&=, 109
&>, 17, 37
&&, 16, 35, 110
|, 68
|=|, 19, 77, 110
|>>, 38
|&>, 38
||, 68
~, 68
~=, 36, 110

A
appendInstant, 54
asBinary, 70
asEWKB, 70
asEWKT, 69
asHexEWKB, 70
asMFJSON, 69
asMVTGeom, 77
asText, 69
atGeometry, 58, 108
atMax, 57
atMin, 57
atPeriod, 58
atPeriodSet, 58
atRange, 57
atRanges, 57
atStbox, 59
atTbox, 58
atTimestamp, 58
atTimestampSet, 58
atValue, 56, 108
atValues, 57
azimuth, 74, 107

B
bearing, 74
bucketList, 86

C
contains, 81, 111
cumulativeLength, 74, 106

D
degrees, 67
derivative, 68
disjoint, 81, 111
duration, 12, 50
dwithin, 81, 111
dynamicTimeWarp, 84
dynamicTimeWarpPath, 84

E
endInstant, 51
endPeriod, 13
endPosition, 101
endSequence, 52
endTimestamp, 12, 52
endValue, 49
expandSpatial, 32
expandTemporal, 32
expandValue, 32
extent, 20, 33, 92

F
frechetDistance, 83
frechetDistancePath, 84

G
geoMeasure, 76
getPosition, 100
getTime, 50
getTimestamp, 50
getValue, 48
getValues, 48, 106
getX, 73
getY, 73
getZ, 73

H
hasT, 30
hasX, 30
hasZ, 30

I
instantN, 51
instants, 51
interpolation, 48
intersects, 82, 111
intersectsPeriod, 53
intersectsPeriodSet, 53
intersectsTimestamp, 53
intersectsTimestampSet, 53
isSimple, 73

L
length, 73, 106
lower, 11, 69
lower_inc, 11

M
makeSimple, 75
maxInstant, 49
maxValue, 49
memSize, 11, 47
merge, 55
minInstant, 49
minusGeometry, 60, 108
minusMax, 60
minusMin, 59
minusPeriod, 61
minusPeriodSet, 61
minusRange, 59
minusRanges, 59
minusStbox, 61
minusTbox, 61
minusTimestamp, 60
minusTimestampSet, 60
minusValue, 59, 108
minusValues, 59
minValue, 49
mobilitydb_full_version, 94
mobilitydb_version, 94
multidimGrid, 87
multidimTile, 88

N
nearestApproachDistance, 107
nearestApproachInstant, 78, 107
npoint, 100
nsegment, 100
numInstants, 51
numPeriods, 13
numSequences, 52
numTimestamps, 12, 51

P
period, 9, 12, 50
periodBucket, 87
periodN, 13
periods, 13
periodset, 10

R
rangeBucket, 86
round, 14, 32, 67, 75, 100, 106
route, 100

S
segments, 53
sequenceN, 53
sequences, 53
setSRID, 33, 72
shift, 14, 55
shiftTscale, 14, 56

shortestLine, 79, 108
simplify, 75
spaceSplit, 90
spaceTimeSplit, 91
speed, 74, 106
SRID, 32, 72, 101
startInstant, 51
startPeriod, 13
startPosition, 100
startSequence, 52
startTimestamp, 12, 52
startValue, 49
stbox, 28, 107

T
tand, 92
tavg, 93
tbox, 28
tcentroid, 93, 112
tcontains, 82, 111
tcount, 19, 92, 111
tdisjoint, 82, 111
tdwithin, 83, 111
tempSubtype, 47
tgeogpointFromBinary, 71
tgeogpointFromEWKB, 72
tgeogpointFromEWKT, 71
tgeogpointFromHexEWKB, 72
tgeogpointFromMFJSON, 71
tgeogpointFromText, 70
tgeompointFromBinary, 71
tgeompointFromEWKB, 72
tgeompointFromEWKT, 71
tgeompointFromHexEWKB, 72
tgeompointFromMFJSON, 71
tgeompointFromText, 70
timeBucket, 87
timespan, 12, 50
timeSplit, 89
timestampN, 13, 52
timestamps, 13, 52
timestampset, 10
tintersects, 83, 111
Tmax, 31
tmax, 92
Tmin, 31
tmin, 92
tnpoint_inst, 105
tnpoint_instset, 105
tnpoint_seq, 105
tnpoint_seqset, 105
toLinear, 54
tor, 92
touches, 82, 111
transform, 33, 72
tscale, 14, 56
tsum, 92
ttouches, 83, 111
ttype_inst, 43, 54
ttype_instset, 44, 54
ttype_seq, 44, 54
ttype_seqset, 44, 54
tunion, 20
twAvg, 54
twCentroid, 74, 107

U
upper, 11, 69
upper_inc, 11

V
valueAtTimestamp, 50, 106
valueBucket, 86
valueRange, 50
valueSplit, 89
valueTimeSplit, 90

W
wavg, 93
wcount, 93, 112
wmax, 93
wmin, 93
wsum, 93

X
Xmax, 30
Xmin, 30

Y
Ymax, 31
Ymin, 31

Z
Zmax, 31
Zmin, 31