MobilityDB 1.3 User's Manual

Esteban Zimányi

COLLABORATORS			
	TITLE : MobilityDB 1.3 User's Manual		
ACTION	*NAME*	*DATE*	*SIGNATURE*
WRITTEN BY	Esteban Zimányi	February 17, 2026	

REVISION HISTORY			
NUMBER	DATE	DESCRIPTION	NAME

Contents

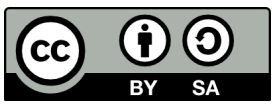
List of Figures

List of Tables

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. A binding for the Python programming language is also available on Github.

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



Chapter 1

Introduction

MobilityDB is an extension of PostgreSQL and PostGIS that provides *temporal types*. Such data types represent the evolution on time of values of some element type, called the *base type* of the temporal type. For instance, temporal integers may be used to represent the evolution on time of the the gear used by a moving car. 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 speed of a car. 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, topological and distance relationships for geometries) can be intuitively generalized when the values evolve in time.

MobilityDB provides the temporal types `tbool`, `tint`, `tfloat`, `ttext`, `tgeometry`, `tgeography`, `tgeompoint`, and `tgeogpoint`. These temporal types are based, respectively, on the `bool`, `integer`, `float`, and `text` base types provided by PostgreSQL, and on the `geometry` and `geography` base types provided by PostGIS, where `tgeometry` and `tgeography` accept arbitrary geometries/geographies, while `tgeompoint` and `tgeogpoint` only accept 2D or 3D points.[1] Furthermore, MobilityDB provides *set*, *span*, and *span set* template types for representing, respectively, sets of values, ranges of values, and sets of ranges of values of base types or time types. Examples of values of set types are `intset`, `floatset`, and `tstzset`, where the latter represents set of `timestamptz` values. Examples of values of span types are `intspan`, `floatspan`, and `tstzspan`. Examples of values of span set types are `intspanset`, `floatspanset`, and `tstzspanset`.

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: 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

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

- Esteban Zimányi (chair): co-founder of the MobilityDB project, overall project coordination, main contributor of the backend code, BerlinMOD generator

1.2 Other Code Contributors

- Arthur Lesuisse
- Xinyiang Li
- Maxime Schoemans

1.3 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

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 from Sources

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
```

If you do not preload the PostGIS library with the above configuration 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-1.1.so":
  undefined symbol: ST_Distance
```

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"
```

The two extensions PostGIS and MobilityDB can also be created using a single command.

```
psql mobility -c "CREATE EXTENSION MobilityDB cascade"
```

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-1.3.tar.gz https://github.com/MobilityDB/MobilityDB/archive/v1.3.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 v1.3
```

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:

- `CMake` cross-platform build system.
- C compiler `gcc` or `clang`. Other ANSI C compilers can be used but may cause problems compiling some dependencies.
- 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 12 or higher. PostgreSQL is available from http://www.postgresql.org.
- PostGIS version 3 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.

Notice that PostGIS has its own dependencies such as Proj, GEOS, LibXML2, or JSON-C, and these libraries are also used in MobilityDB. Refer to http://trac.osgeo.org/postgis/wiki/UsersWikiPostgreSQLPostGIS for a support matrix of PostGIS with PostgreSQL, GEOS, and Proj.

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-16 postgresql-server-dev-16 postgresql-16-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

Notice that the current version of MobilityDB has been tested on Linux, MacOS, and Windows systems. It may work on other Unix-like systems, but remain untested. 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 '021_tbox'
```

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

```
ctest -R '022_*'
```

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 Installation from Binaries

1.6.1 Debian-based Linux Distributions

Support for Debian-based Linux systems, such as Ubuntu and Arch Linux, is being developed.

1.6.2 Windows

Since PostGIS version 3.3.3, MobilityDB is distributed in the PostGIS Bundle for Windows, which is available on application stackbuilder and OSGeo website. For more information, refer to the PostGIS documentation.

1.7 Support

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

1.7.1 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;
```

1.7.2 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.

1.8 Migrating from Version 1.2 to Version 1.3

MobilityDB 1.3 is a major revision with respect to version 1.2. The most important change in version 1.3 was to enable new spatiotemporal types. While version 1.2 defined the spatiotemporal types `tgeompoint` (temporal geometry point), `tgeogpoint` (temporal geography point), and `tnpoint` (temporal network point), version 1.3 added the new types `tgeometry` (temporal geometry), `tgeography` (temporal geography), `tcbuffer` (temporal circular buffer), `tpose` (temporal pose), and `trgeometry` (temporal rigid geometry). At a conceptual level, all the spatiotemporal types are organized in the hierarchy depicted in Figure 7.1.

In order to enable the above, a refactoring of the code base was necessary. All the types provided by version 1.2 in addition to the new types `tgeometry` and `tgeography` are defined as core types and thus, are always included in the building process. Each one of the other spatiotemporal types can be included in the building process by setting its corresponding compilation flag. For example, when adding `-DCBUFFER=1` the types built upon the `cbuffer` base type, that is `cbuffer`, `cbufferset`, and `tcbuffer`, will be included in the building process. The possibility to select the types to be included in the executable program is important in particular for stream processing, due to the limited capability of the edge devices. Please notice that the new types `tcbuffer`, `tpose`, and `trgeometry` are still in development and will be finalized in a forthcoming release.

In addition, the API of MobilityDB and MEOS was extended with new functionality and streamlined to improve usability. Finally, version 1.3 also enables the latest versions PostgreSQL 18 and PostGIS 3.6.0. For all these reasons, the binary format of the temporal types have changed from version 1.2 to 1.3 and thus a backup and restore is needed for migrating to the newer version 1.3 of MobilityDB.

Chapter 2

Set and Span Types

MobilityDB provides *set*, *span*, and *span set* types for representing set of values another type, which is called the *base type*. Set types are akin to *array types* in PostgreSQL restricted to one dimension, but enforce the constraint that sets do not have duplicates. Span and span set types in MobilityDB correspond to the *range and multirange types* in PostgreSQL but have additional constraints. In particular, span types in MobilityDB are of fixed length and do not allow empty spans and infinite bounds. While span types provide similar functionality to range types, they enable increasing performance. In particular, the overhead of processing variable-length types is removed and, in addition, pointer arithmetics and binary search can be used.

The base types used for constructing set, span, and span set types are the types `integer`, `bigint`, `float`, `text`, `date`, and `timestamptz` (timestamp with time zone) provided by PostgreSQL, the types `geometry` and `geography` provided by PostGIS, and the type `npoint` (network point) provided by MobilityDB (see Chapter 12). MobilityDB provides the following set and span types:

- `set`: `intset`, `bigintset`, `floatset`, `textset`, `dateset`, `tstzset`, `geomset`, `geogset`, `npointset`.
- `span`: `intspan`, `bigintspan`, `floatspan`, `datespan`, `tstzspan`.
- `spanset`: `intspanset`, `bigintspanset`, `floatspanset`, `datespanset`, `tstzspanset`.

We present next the functions and operators for set and span 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 functions and operators, we use the following notation:

- `set` represents any set type, such as `intset` or `tstzset`.
- `span` represents any span type, such as `intspan` or `tstzspanset`.
- `spanset` represents any span set type, such as `intspanset` or `tstzspanset`.
- `spans` represents any span or span set type, such as `intspan` or `tstzspanset`.
- `base` represents any base type of a set or span type, such as `integer` or `timestamptz`
- `number` represents any base type of a number span type, such as `integer` or `float`,
- `numset` represents any number set type, such as `intset` or `floatset`.
- `numspans` represents any number span type, such as `intspan` or `floatspanset`.
- `numbers` represents any number set or range type, such as `integer`, `intset`, `intspan`, or `intspanset`,
- `dates` represents any time type with `date` granularity, that is, `date`, `dateset`, `datespan`, or `datespanset`,
- `times` represents any time type with `timestamptz` granularity, that is, `timestamptz`, `tstzset`, `tstzspan`, or `tstzspanset`,

- A set of types such as `{set,spans}` represents any of the types listed,
- A set of operators such as `{=, <>}` represents any of the operators listed,
- `type[]` represents an array of `type`.

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

```
{set,spans} @> {set,spans,base} → boolean
```

Notice that the signature above is an abridged version of the more precise signature below

```
set @> {set,base} → boolean
spans @> {spans,base} → boolean
```

since sets and spans cannot be mixed in operations and thus, for instance, we cannot ask wether a span contains a set. In the following, for conciseness, we use the abridged style of signatures above. Furthermore, the time part of the timestamps is omitted in most examples. Recall that in that case PostgreSQL assumes the time `00:00:00`.

In what follows, since span and span set types have similar functions and operators, when we speak about span types we mean both span and span set types, unless we explicitly refer to *unit* span types and span *set* types to distinguish them.

2.1 Input and Output

MobilityDB generalizes Open Geospatial Consortium's (OGC) Well-Known Text (WKT) and Well-Known Binary (WKB) input and output format for all its types. In this way, applications can exchange data between them using a standardized exchange format. The WKT format is human-readable while the WKB format is more compact and more efficient than the WKT format. The WKB format can be output either as a binary string or as a character string encoded in hexadecimal ASCII.

The set types represent an *ordered* set of *distinct* values. A set must contain at least one element. Examples of set values are as follows:

```
SELECT tstzset '{2001-01-01 08:00:00, 2001-01-03 09:30:00}';
-- Singleton set
SELECT textset '{"highway"}';
-- Erroneous set: unordered elements
SELECT floatset '{3.5, 1.2}';
-- Erroneous set: duplicate elements
SELECT geomset '{"Point(1 1)", "Point(1 1)"}';
```

Notice that the elements of the sets `textset`, `geomset`, `geogset`, and `npointset` must be enclosed between double quotes. Notice also that geometries and geographies follow the order defined in PostGIS.

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

```
SELECT intspan '[1, 3)';
SELECT floatspan '[1.5, 3.5]';
SELECT tstzspan '[2001-01-01 08:00:00, 2001-01-03 09:30:00)';
-- Instant spans
SELECT intspan '[1, 1]';
SELECT floatspan '[1.5, 1.5]';
SELECT tstzspan '[2001-01-01 08:00:00, 2001-01-01 08:00:00]';
-- Erroneous span: invalid bounds
SELECT tstzspan '[2001-01-01 08:10:00, 2001-01-01 08:00:00]';
-- Erroneous span: empty span
SELECT tstzspan '[2001-01-01 08:00:00, 2001-01-01 08:00:00)';
```

Values of `intspan`, `bigintspan`, and `datespan` are converted into *normal form* so that equivalent values have identical representations. In the canonical representation of these types, the lower bound is inclusive and the upper bound is exclusive as shown in the following examples:

```
SELECT intspan '[1, 1]';
-- [1, 2)
SELECT bigintspan '(1, 3]';
--[2, 4)
SELECT datespan '[2001-01-01, 2001-01-03]';
-- [2001-01-01, 2001-01-04)
```

A value of a span set type represents an *ordered* set of *disjoint* span values. A span set value must contain at least one element, in which case it corresponds to a single span value. Examples of span set values are as follows:

```
SELECT floatspanset '{[8.1, 8.5],[9.2, 9.4]}';
-- Singleton spanset
SELECT tstzspanset '{[2001-01-01 08:00:00, 2001-01-01 08:10:00]}';
-- Erroneous spanset: unordered elements
SELECT intspanset '{[3,4],[1,2]}';
-- Erroneous spanset: overlapping elements
SELECT tstzspanset '{[2001-01-01 08:00:00, 2001-01-01 08:10:00],
  [2001-01-01 08:05:00, 2001-01-01 08:15:00]}';
```

Values of the span set types are converted into *normal form* so that equivalent values have identical representations. For this, consecutive adjacent span values are merged when possible. Examples of transformation into normal form are as follows:

```
SELECT intspanset '{[1,2],[3,4]}';
-- {[1, 5)}
SELECT floatspanset '{[1.5,2.5],(2.5,4.5]}';
-- {[1.5, 4.5]}
SELECT tstzspanset '{[2001-01-01 08:00:00, 2001-01-01 08:10:00),
  [2001-01-01 08:10:00, 2001-01-01 08:10:00], (2001-01-01 08:10:00, 2001-01-01 08:20:00]}';
-- {[2001-01-01 08:00:00+00,2001-01-01 08:20:00+00]}
```

We give next the functions for input and output of set and span types in Well-Known Text and Well-Known Binary format. The default output format of all set and span types is the Well-Known Text format. The function `asText` given next enables to determine the output of floating point values.

- Return the Well-Known Text (WKT) representation

  `asText({floatset,floatspans},maxdecdigits=15)` → `text`

  The `maxdecdigits` argument can be used to set the maximum number of decimal places in the output of floating point values (default 15).

  ```
  SELECT asText(floatset '{1.123456789,2.123456789}', 3);
  -- {1.123, 2.123}
  SELECT asText(floatspanset '{[1.55,2.55],[4,5]}',0);
  -- {[2, 3], [4, 5]}
  ```

- Return the Well-Known Binary (WKB) or the Hexadecimal Well-Known Binary (HexWKB) representation

  `asBinary({set,spans},endian text='')` → `bytea`

  `asHexWKB({set,spans},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 the encoding of the machine is used.

  ```
  SELECT asBinary(dateset '{2001-01-01, 2001-01-03}');
  -- \x0105000102000000060e01000070010000
  SELECT asBinary(intspan '[1, 3)');
  -- \x011300010100000003000000
  SELECT asBinary(floatspanset '{[1, 2], [4, 5]}', 'XDR');
  ```

```
-- \x00000e00000002033ff0000000000004000000000000000034010000000000000401400000000000
SELECT asHexWKB(dateset '{2001-01-01, 2001-01-03}');
-- 01050001020000006E01000070010000
SELECT asHexWKB(intspan '[1, 3)');
-- 0113000101000000003000000
SELECT asHexWKB(floatspanset '{[1, 2], [4, 5]}', 'XDR');
-- 00000E00000002033FF0000000000004000000000000000034010000000000000401400000000000
```

- Input from the Well-Known Binary (WKB) representation

  `settypeFromBinary(bytea) → set`

  `spantypeFromBinary(bytea) → span`

  `spansettypeFromBinary(bytea) → spanset`

  There is one function per set or span (set) type, the name of the function has as prefix the name of the type

```
SELECT datesetFromBinary('\x01050001020000006e01000070010000');
-- {2001-01-01, 2001-01-03}
SELECT intspanFromBinary('\x0113000101000000003000000');
-- [1, 3)
SELECT floatspansetFromBinary(
  '\x00000e00000002033ff0000000000004000000000000000034010000000000000401400000000000');
-- {[1, 2], [4, 5]}
```

- Input from the Hexadecimal Well-Known Binary (HexWKB) representation

  `settypeFromHexWKB(text) → set`

  `spantypeFromHexWKB(text) → span`

  `spansettypeFromHexWKB(text) → spanset`

  There is one function per set or span (set) type, the name of the function has as prefix the name of the type.

```
SELECT datesetFromHexWKB('01050001020000006E01000070010000');
-- {2001-01-01, 2001-01-03}
SELECT intspanFromHexWKB('0113000101000000003000000');
-- [1, 3)
SELECT floatspanFromHexWKB('010600010000000000000F83F0000000000000440');
-- [1.5, 2.5)
SELECT floatspansetFromHexWKB(
  '00000E00000002033FF0000000000004000000000000000034010000000000000401400000000000');
-- {[1, 2], [4, 5]}
```

2.2 Constructors

The constructor function for the set types has a single argument that is an array of values of the corresponding base type. The values must be ordered and cannot have nulls or duplicates.

- Constructor for set types

  `set(base[]) → set`

```
SELECT set(ARRAY['highway', 'primary', 'secondary']);
-- {"highway", "primary", "secondary"}
SELECT set(ARRAY[timestamptz '2001-01-01 08:00:00', '2001-01-03 09:30:00']);
-- {2001-01-01 08:00:00+00, 2001-01-03 09:30:00+00}
```

The unit span types have a constructor function that accepts four arguments. The first two arguments specify, respectively, the lower and upper bound, and the last two arguments are Boolean values stating, respectively, whether the lower and upper bounds are inclusive or not. The last two arguments are assumed to be, respectively, true and false if not specified. Notice that integer spans are transformed into *normal form*, that is, with inclusive lower bound and exclusive upper bound.

- Constructor for span types

  `span(lower base,upper base,leftInc bool=true,rightInc bool=false) → span`

```
SELECT span(20.5, 25);
-- [20.5, 25)
SELECT span(20, 25, false, true);
-- [21, 26)
SELECT span(timestamptz '2001-01-01 08:00:00', '2001-01-03 09:30:00', false, true);
-- (2001-01-01 08:00:00, 2001-01-03 09:30:00]
```

The constructor function for span set types have a single argument that is an array of span values of the same subtype.

- Constructor for span set types

  `spanset(span[]) → spanset`

```
SELECT spanset(ARRAY[intspan '[10,12]', '[13,15]']);
-- {[10, 16)}
SELECT spanset(ARRAY[floatspan '[10.5,12.5]', '[13.5,15.5]']);
-- {[10.5, 12.5], [13.5, 15.5]}
SELECT spanset(ARRAY[tstzspan '[2001-01-01 08:00, 2001-01-01 08:10]',
  '[2001-01-01 08:20, 2001-01-01 08:40]']);
-- {[2001-01-01 08:00, 2001-01-01 08:10], [2001-01-01 08:20, 2001-01-01 08:40]};
```

2.3 Conversions

Values of set and span types can be converted to one another or converted to and from PostgreSQL range types using the function `CAST` or using the `::` notation.

- Convert a base value to a set, span, or span set value

  `base::{set,span,spanset}`

  `set(base) → set`

  `span(base) → span`

  `spanset(base) → spanset`

```
SELECT CAST(timestamptz '2001-01-01 08:00:00' AS tstzset);
-- {2001-01-01 08:00:00}
SELECT timestamptz '2001-01-01 08:00:00'::tstzspan;
-- [2001-01-01 08:00:00, 2001-01-01 08:00:00]
SELECT spanset(timestamptz '2001-01-01 08:00:00');
-- {[2001-01-01 08:00:00, 2001-01-01 08:00:00]}
```

- Convert a set value to a span set value

  `set::spanset`

  `spanset(set) → spanset`

```
SELECT spanset(tstzset '{2001-01-01 08:00:00, 2001-01-01 08:15:00,
  2001-01-01 08:25:00}');
/* {[2001-01-01 08:00:00, 2001-01-01 08:00:00],
    [2001-01-01 08:15:00, 2001-01-01 08:15:00],
    [2001-01-01 08:25:00, 2001-01-01 08:25:00]} */
```

- Convert a span value to a span set value

  `span::spanset`

  `spanset(span) → spanset`

```
SELECT floatspan '[1.5,2.5]'::floatspanset;
-- {[1.5, 2.5]}
SELECT tstzspan '[2001-01-01 08:00:00, 2001-01-01 08:30:00)'::tstzspanset;
-- {[2001-01-01 08:00:00, 2001-01-01 08:30:00)}
```

- Convert a set or a span set into a span, ignoring the potential time gaps

  `{set,spanset}::span`

  `span({set,spanset}) → span`

```
SELECT span(dateset '{2001-01-01, 2001-01-03, 2001-01-05}');
-- [2001-01-01, 2001-01-06)
SELECT span(tstzspanset '{[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-04)}');
-- [2001-01-01, 2001-01-04)
```

- Convert a span value to and from a PostgreSQL range value

  `span::range`

  `range::span`

  `range(span) → range`

  `span(range) → span`

  Notice that PostgreSQL range values accept empty ranges and ranges with infinite values, which are not allowed as span values in MobilityDB

```
SELECT intspan '[10, 20)'::int4range;
-- [10,20)
SELECT tstzspan '[2001-01-01 08:00:00, 2001-01-01 08:30:00)'::tstzrange;
-- ["2001-01-01 08:00:00","2001-01-01 08:30:00")
SELECT int4range '[10, 20)'::intspan;
-- [10,20)
SELECT int4range 'empty'::intspan;
-- ERROR:  Range cannot be empty
SELECT int4range '[10,)'::intspan;
-- ERROR:  Range bounds cannot be infinite
SELECT tstzrange '[2001-01-01 08:00:00, 2001-01-01 08:30:00)'::tstzspan;
-- [2001-01-01 08:00:00, 2001-01-01 08:30:00)
```

- Convert a span set value to and from a PostgreSQL multirange value

  `spanset::multirange`

  `multirange::spanset`

  `multirange(spanset) → multirange`

  `spanset(multirange) → spanset`

```
SELECT intspanset '{[1,2],[4,5]}'::int4multirange;
-- {[1,3),[4,6)}
SELECT tstzspanset '{[2001-01-01,2001-01-02],[2001-01-04,2001-01-05]}'::tstzmultirange;
-- {[2001-01-01,2001-01-02],[2001-01-04,2001-01-05]}
SELECT int4multirange '{[1,2],[4,5]}'::intspanset;
-- {[1, 3), [4, 6)}
SELECT tstzmultirange '{[2001-01-01,2001-01-02],[2001-01-04,2001-01-05]}'::tstzspanset;
-- {[2001-01-01, 2001-01-02], [2001-01-04, 2001-01-05]}
```

2.4 Accessors

- Return the memory size in bytes

  `memSize({set,spanset}) → integer`

```
SELECT memSize(tstzset '{2001-01-01, 2001-01-02, 2001-01-03}');
-- 48
SELECT memSize(tstzspanset '{[2001-01-01, 2001-01-02], [2001-01-03, 2001-01-04],
  [2001-01-05, 2001-01-06]}');
-- 112
```

- Return the lower or upper bound

  `lower(spans) → base`

  `upper(spans) → base`

```
SELECT lower(tstzspan '[2001-01-01, 2001-01-05)');
-- 2001-01-01
SELECT lower(intspanset '{[1,2],[4,5]}');
--  1
```

```
SELECT lower(tstzspan '[2001-01-01, 2001-01-05)');
-- 2001-01-01
SELECT upper(intspanset '{[1,2],[4,5]}');
--  6
SELECT lower(tstzspan '[2001-01-01, 2001-01-05)');
-- 2001-01-01
SELECT lower(intspanset '{[1,2],[4,5]}');
--  1
```

```
SELECT upper(floatspan '[20.5, 25.3)');
-- 25.3
SELECT upper(tstzspan '[2001-01-01, 2001-01-05)');
-- 2001-01-05
```

- Is the lower or upper bound inclusive?

  `lowerInc(spans) → boolean`

  `upperInc(spans) → boolean`

```
SELECT lowerInc(datespan '[2001-01-01, 2001-01-05)');
-- true
SELECT lowerInc(intspanset '{[1,2],[4,5]}');
-- true
```

```
SELECT upper(floatspan '[20.5, 25.3]');
-- true
SELECT upperInc(tstzspan '[2001-01-01, 2001-01-05)');
-- false
```

- Return the width of the span as a float

  `width(numspan) → float`

  `width(numspanset,boundspan=false) → float`

  An additional parameter can be set to true to compute the width of the bounding span, thus ignoring the potential value gaps

```
SELECT width(floatspan '[1, 3)');
-- 2
SELECT width(intspanset '{[1,3),[5,7)}');
-- 4
SELECT width(intspanset '{[1,3),[5,7)}', true);
-- 6
```

- Return the duration

  `duration({datespan,tstzspan}) → interval`

  `duration({datespanset,tstzspanset},boundspan bool=false) → interval`

  An additional parameter can be set to true to compute the duration of the bounding time span, thus ignoring the potential time gaps

```
SELECT duration(datespan '[2001-01-01, 2001-01-03)');
-- 2 days
SELECT duration(tstzspanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-05)}');
-- 3 days
SELECT duration(tstzspanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-05)}', true);
-- 4 days
```

- Return the (number of) values

  `numValues(set) → integer`

  `getValues(set) → base[]`

```
SELECT numValues(intset '{1,3,5,7}');
-- 4
SELECT getValues(tstzset '{2001-01-01, 2001-01-03, 2001-01-05, 2001-01-07}');
-- {"2001-01-01","2001-01-03","2001-01-05","2001-01-07"}
```

- Return the start, end, or n-th value

  `startValue(set) → base`

  `endValue(set) → base`

  `valueN(set,integer) → base`

```
SELECT startValue(intset '{1,3,5,7}');
-- 1
SELECT endValue(dateset '{2001-01-01, 2001-01-03, 2001-01-05, 2001-01-07}');
-- 2001-01-07
SELECT valueN(floatset '{1,3,5,7}',2);
-- 3
```

- Return the (number of) spans

  `numSpans(spanset) → integer`

  `spans(spanset) → span[]`

```
SELECT numSpans(intspanset '{[1,3),[4,5),[6,7)}');
-- 3
SELECT numSpans(datespanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-05),
  [2001-01-06, 2001-01-07)}');
-- 3
```

```
SELECT spans(floatspanset '{[1,3),[4,4],[6,7)}');
-- {"[1,3)","[4,4]","[6,7)"}
SELECT spans(tstzspanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-04],
  [2001-01-05, 2001-01-06)}');
-- {"[2001-01-01,2001-01-03)", "[2001-01-04,2001-01-04]", "[2001-01-05,2001-01-06)"}
```

- Return the start, end, or n-th span

  `startSpan(spanset) → span`

  `endSpan(spanset) → span`

  `spanN(spanset,integer) → span`

```
SELECT startSpan(intspanset '{[1,3),[4,5),[6,7)}');
-- [1,3)
SELECT startSpan(datespanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-05),
  [2001-01-06, 2001-01-07)}');
-- [2001-01-01,2001-01-03)
```

```
SELECT endSpan(floatspanset '{[1,3),[4,4],[6,7)}');
-- [6,7)
SELECT endSpan(tstzspanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-04],
  [2001-01-05, 2001-01-06)}');
-- [2001-01-05,2001-01-06)
```

```
SELECT spanN(floatspanset '{[1,3),[4,4],[6,7)}',2);
-- [4,4]
SELECT spanN(tstzspanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-04],
  [2001-01-05, 2001-01-06)}', 2);
-- [2001-01-04,2001-01-04]
```

- Return the (number of) different dates

  `numDates(datespanset) → integer`

  `dates(datespanset) → dateset`

```
SELECT numDates(datespanset '{[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-04)}');
-- 4
SELECT dates(datespanset '{[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-04)}');
-- {2001-01-01, 2001-01-02, 2001-01-03, 2001-01-04}
```

- Return the start, end, or n-th date

  `startDate(datespanset) → date`

  `endDate(datespanset) → date`

  `dateN(datespanset,integer) → date`

  The functions do not take into account whether the bounds are inclusive or not.

```
SELECT startDate(datespanset '{[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-04)}');
-- 2001-01-01
SELECT endDate(datespanset '{[2001-01-01, 2001-01-03), (2001-01-03, 2001-01-05)}');
-- 2001-01-05
SELECT dateN(datespanset '{[2001-01-01, 2001-01-03), (2001-01-03, 2001-01-05)}', 3);
-- 2001-01-05
```

- Return the (number of) different timestamps

  `numTimestamps(tstzspanset) → integer`

  `timestamps(tstzspanset) → tstzset`

```
SELECT numTimestamps(tstzspanset '{[2001-01-01, 2001-01-03), (2001-01-03, 2001-01-05)}');
-- 3
SELECT timestamps(tstzspanset '{[2001-01-01, 2001-01-03), (2001-01-03, 2001-01-05)}');
-- {"2001-01-01 00:00:00", "2001-01-03 00:00:00", "2001-01-05 00:00:00"}
```

- Return the start, end, or n-th timestamp

  `startTimestamp(tstzspanset) → timestamptz`

  `endTimestamp(tstzspanset) → timestamptz`

  `timestampN(tstzspanset,integer) → timestamptz`

  The functions do not take into account whether the bounds are inclusive or not.

```
SELECT startTimestamp(tstzspanset '{[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-04)}');
-- 2001-01-01
SELECT endTimestamp(tstzspanset '{[2001-01-01, 2001-01-03), (2001-01-03, 2001-01-05)}');
-- 2001-01-05
SELECT timestampN(tstzspanset '{[2001-01-01, 2001-01-03), (2001-01-03, 2001-01-05)}', 3);
-- 2001-01-05
```

2.5 Transformations

- Expand or shrink the bounds by a value or an interval

  `expand(numspan,base) → numspan`

  `expand(tstzspan,interval) → tstzspan`

  The function returns NULL if the value or interval given as second argument is negative and the span resulting from shifting the bounds with the argument is empty.

```
SELECT expand(floatspan '[1, 3]', 1);
-- [0, 4]
SELECT expand(floatspan '[1, 3]', -1);
-- [2, 2]
SELECT expand(floatspan '[1, 3)', -1);
-- NULL
SELECT expand(tstzspan '[2001-01-01, 2001-01-03]', interval '1 day');
-- [2000-12-31, 2001-01-04]
SELECT expand(tstzspan '[2001-01-01, 2001-01-03]', interval '-1 day');
-- [2001-01-02, 2001-01-02]
SELECT expand(tstzspan '[2001-01-01, 2001-01-03]', interval '-2 day');
-- NULL
```

- Shift by a value or interval

  `shift(numbers,base) → numbers`

  `shift(dates,integer) → dates`

  `shift(times,interval) → times`

```
SELECT shift(dateset '{2001-01-01, 2001-01-03, 2001-01-05}', 1);
-- {2001-01-02, 2001-01-04, 2001-01-06}
SELECT shift(intspan '[1, 4)', -1);
-- [0, 3)
SELECT shift(tstzspan '[2001-01-01, 2001-01-03]', interval '1 day');
-- [2001-01-02, 2001-01-04]
SELECT shift(floatspanset '{[1, 2], [3, 4]}', -1);
-- {[0, 1], [2, 3]}
SELECT shift(tstzspanset '{[2001-01-01, 2001-01-03], [2001-01-04, 2001-01-05]}',
  interval '1 day');
-- {[2001-01-02, 2001-01-04], [2001-01-05, 2001-01-06]}
```

- Scale by a value or interval

  `scale(numbers,base) → numbers`

  `scale(dates,integer) → dates`

`scale(times,interval) → times`

If the width or time span of the input value is zero (for example, for a singleton timestamp set), the result is the input value. The given value or interval must be strictly greater than zero.

```
SELECT scale(tstzset '{2001-01-01}', '1 day');
-- {2001-01-01}
SELECT scale(tstzset '{2001-01-01, 2001-01-03, 2001-01-05}', '2 days');
-- {2001-01-01, 2001-01-02, 2001-01-03}
SELECT scale(intspan '[1, 4)', 4);
-- [1, 6)
SELECT scale(datespan '[2001-01-01, 2001-01-04)', 4);
-- [2001-01-01, 2001-01-06)
SELECT scale(tstzspan '[2001-01-01, 2001-01-03]', '1 day');
-- [2001-01-01, 2001-01-02]
SELECT scale(floatspanset '{[1, 2], [3, 4]}', 6);
-- {[1, 3], [5, 7]}
SELECT scale(tstzspanset '{[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 scale(tstzset '{2001-01-01}', '-1 day');
-- ERROR:  The duration must be a positive interval: -1 days
```

- Shift and scale by the values or intervals

`shiftScale(numbers,base,base) → numbers`

`shiftScale(dates,integer,integer) → dates`

`shiftScale(times,interval,interval) → times`

This function combines the functions shift and scale.

```
SELECT shiftScale(tstzset '{2001-01-01}', '1 day', '1 day');
-- {2001-01-02}
SELECT shiftScale(tstzset '{2001-01-01, 2001-01-03, 2001-01-05}', '1 day','2 days');
-- {2001-01-02, 2001-01-03, 2001-01-04}
SELECT shiftScale(intspan '[1, 4)', -1, 4);
-- [0, 5)
SELECT shiftScale(datespan '[2001-01-01, 2001-01-04)', -1, 4);
-- [2001-12-31, 2001-01-05)
SELECT shiftScale(tstzspan '[2001-01-01, 2001-01-03]', '1 day', '1 day');
-- [2001-01-02, 2001-01-03]
SELECT shiftScale(floatspanset '{[1, 2], [3, 4]}', -1, 6);
-- {[0, 2], [4, 6]}
SELECT shiftScale(tstzspanset '{[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 down or up to the nearest integer

`floor({floatset,floatspans}) → {floatset,floatspans}`

`ceil({floatset,floatspans}) → {floatset,floatspans}`

```
SELECT floor(floatset '{1.5,2.5}');
-- {1, 2}
SELECT ceil(floatspan '[1.5,2.5)');
-- [2, 3)
SELECT floor(floatspan '(1.5, 1.6)');
-- [1, 1]
SELECT ceil(floatspanset '{[1.5, 2.5],[3.5,4.5]}');
-- {[2, 3], [4, 5]}
```

- Round to a number of decimal places

  `round({floatset,floatspans},integer=0) → {floatset,floatspans}`

```
SELECT round(floatset '{1.123456789,2.123456789}', 3);
-- {1.123, 2.123}
SELECT round(floatspan '[1.123456789,2.123456789)', 3);
-- [1.123,2.123)
SELECT round(floatspan '[1.123456789, inf)', 3);
-- [1.123,Infinity)
SELECT round(floatspanset '{[1.123456789, 2.123456789],[3.123456789,4.123456789]}', 3);
-- {[1.123, 2.123], [3.123, 4.123]}
```

- Convert to degrees or radians

  `degrees({floatset,floatspans}, normalize=false) → {floatset,floatspans}`

  `radians({floatset,floatspans}) → {floatset,floatspans}`

  The additional parameter in the `degrees` function can be used to normalize the values between 0 and 360 degrees.

```
SELECT round(degrees(floatset '{0, 0.5, 0.7, 1.0}', true), 3);
-- {0, 28.648, 40.107, 57.296}
SELECT round(radians(floatspanset '{[0, 45], [90, 135]}'), 3);
-- {[0, 0.785], [1.571, 2.356]}
```

- Transform to lowercase, uppercase, or initcap

  `lower(textset) → textset`

  `upper(textset) → textset`

  `initcap(textset) → textset`

```
SELECT lower(textset '{"AAA", "BBB", "CCC"}');
-- {"aaa", "bbb", "ccc"}
SELECT upper(textset '{"aaa", "bbb", "ccc"}');
-- {"AAA", "BBB", "CCC"}
SELECT initcap(textset '{"aaa", "bbb", "ccc"}');
-- {"Aaa", "Bbb", "Ccc"}
```

- Text concatenation

  `{text,textset} || {text,textset} → textset`

```
SELECT textset '{aaa, bbb}' || text 'XX';
-- {"aaaXX", "bbbXX"}
SELECT text 'XX' || textset '{aaa, bbb}';
-- {"XXaaa", "XXbbb"}
```

- Set the temporal precision of the time value to the interval with respect to the origin

  `tprecision(times,interval,origin timestamptz='2000-01-03') → times`

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

```
SELECT tprecision(timestamptz '2001-12-03', '30 days');
-- 2001-11-23
SELECT tprecision(timestamptz '2001-12-03', '30 days', '2001-12-01');
-- 2001-12-01
SELECT tprecision(tstzset '{2001-01-01 08:00, 2001-01-01 08:10, 2001-01-01 09:00,
  2001-01-01 09:10}', '1 hour');
-- {"2001-01-01 08:00:00+01", "2001-01-01 09:00:00+01"}
SELECT tprecision(tstzspan '[2001-12-01 08:00, 2001-12-01 09:00]', '1 day');
-- [2001-12-01, 2001-12-02)
SELECT tprecision(tstzspan '[2001-12-01 08:00, 2001-12-15 09:00]', '1 day');
-- [2001-12-01, 2001-12-16)
```

```
SELECT tprecision(tstzspanset '{[2001-12-01 08:00, 2001-12-01 09:00],
  [2001-12-01 10:00, 2001-12-01 11:00]}', '1 day');
-- {[2001-12-01, 2001-12-02)}
SELECT tprecision(tstzspanset '{[2001-12-01 08:00, 2001-12-01 09:00],
  [2001-12-01 10:00, 2001-12-01 11:00]}', '1 day');
-- {[2001-12-01, 2001-12-02)}
```

2.6 Spatial Reference System

- Return or set the spatial reference identifier

  `SRID(spatialset) → integer`

  `setSRID(spatialset) → spatialset`

```
SELECT SRID(geomset '{"Point(1 1)", "Point(2 2)"}');
-- 0
SELECT SRID(geogset '{"Linestring(1 1,2 2)","Polygon((1 1,1 2,2 2,2 1,1 1))"}');
-- 4326
SELECT SRID(geomset 'SRID=5676;{"Linestring(1 1,2 2)","Polygon((1 1,1 2,2 2,2 1,1 1))"}');
-- 5676
SELECT asEWKT(setSRID(geomset '{Point(1 1), Point(2 2)}', 5676));
-- SRID=5676;{"POINT(1 1)", "POINT(2 2)"}
SELECT asEWKT(setSRID(poseset '{"Pose(Point(1 1),1)", "Pose(Point(2 2),3)"}', 5676));
-- SRID=5676;{"Pose(POINT(2 2),3)", "Pose(POINT(1 1),1)"}
```

- Transform to a spatial reference identifier

  `transform(spatialset,to_srid integer) → spatialset`

  `transformPipeline(spatialset,pipeline text,to_srid integer,is_forward bool=true) → spatialset`

  The `transform` function specifies the transformation with a target SRID. An error is raised when the input set has an unknown SRID (represented by 0). The `transformPipeline` function specifies the transformation with a coordinate transformation pipeline in the following format:

  `urn:ogc:def:coordinateOperation:AUTHORITY::CODE`

  The SRID of the input set is ignored, and the SRID of the output set will be set to zero unless a value is provided via the optional `to_srid` parameter. As stated by the last parameter, the pipeline is executed by default in a forward direction; by setting the parameter to false, the pipeline is executed in the inverse direction.

```
SELECT asEWKT(transform(geomset 'SRID=4326;{Point(2.340088 49.400250),
  Point(6.575317 51.553167)}', 3812), 6);
-- SRID=3812;{"POINT(502773.429981 511805.120402)", "POINT(803028.908265 751590.742629)"}
WITH test(geoset, pipeline) AS (
  SELECT geogset 'SRID=4326;{"Point(4.3525 50.846667 100.0)",
    "Point(-0.1275 51.507222 100.0)"}',
    text 'urn:ogc:def:coordinateOperation:EPSG::16031' )
SELECT asEWKT(transformPipeline(transformPipeline(geoset, pipeline, 4326), pipeline,
  4326, false), 6)
FROM test;
-- SRID=4326;{"POINT Z (4.3525 50.846667 100)", "POINT Z (-0.1275 51.507222 100)"}
```

2.7 Set Operations

The set and span types have associated set operators, namely union, difference, and intersection, which are represented, respectively by $+$, $-$, and $*$. The set operators for the set and span types are given next.

- Union, difference, or intersection of sets or spans

  `set {+, -, *} set → set`

  `spans {+, -, *} spans → spans`

```
SELECT dateset '{2001-01-01, 2001-01-03, 2001-01-05}' +
  dateset '{2001-01-03, 2001-01-06}';
-- {2001-01-01, 2001-01-03, 2001-01-05, 2001-01-06}
SELECT intspan '[1, 3)' + intspan '[3, 5)';
-- [1, 5)
SELECT floatspan '[1, 3)' + floatspan '[4, 5)';
-- {[1, 3), [4, 5)}
SELECT tstzspanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-05)}' +
  tstzspan '[2001-01-03, 2001-01-04)';
-- {[2001-01-01, 2001-01-05)}
```

```
SELECT intset '{1, 3, 5}' - intset '{3, 6}';
-- {1, 5}
SELECT datespan '[2001-01-01, 2001-01-05)' - datespan '[2001-01-03, 2001-01-07)';
-- {[2001-01-01, 2001-01-03)}
SELECT floatspan '[1, 5]' - floatspan '[3, 4]';
-- {[1, 3), (4, 5]}
SELECT tstzspanset '{[2001-01-01, 2001-01-06], [2001-01-07, 2001-01-10]}' -
  tstzspanset '{[2001-01-02, 2001-01-03], [2001-01-04, 2001-01-05],
  [2001-01-08, 2001-01-09]}';
/* {[2001-01-01,2001-01-02), (2001-01-03,2001-01-04), (2001-01-05,2001-01-06],
   [2001-01-07,2001-01-08), (2001-01-09,2001-01-10]} */
```

```
SELECT tstzset '{2001-01-01, 2001-01-03}' * tstzset '{2001-01-03, 2001-01-05}';
-- {2001-01-03}
SELECT intspan '[1, 5)' * intspan '[3, 6)';
-- [3, 5)
SELECT floatspanset '{[1, 5),[6, 8)}' * floatspan '[1, 6)';
-- {[1, 5)}
SELECT tstzspan '[2001-01-01, 2001-01-05)' * tstzspan '[2001-01-03, 2001-01-07)';
-- [2001-01-03, 2001-01-05)
```

2.8 Bounding Box Operations

2.8.1 Topological Operations

The topological operations available for the set and span types are given next.

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

  `{set,spans} && {set,spans} → boolean`

```
SELECT intset '{1, 3}' && intset '{2, 3, 4}';
-- true
SELECT floatspan '[1, 3)' && floatspan '[3, 4)';
-- false
SELECT tstzspan '[2001-01-01, 2001-01-05)' && tstzspan '[2001-01-02, 2001-01-07)';
-- true
SELECT floatspanset '{[1, 5),[6, 8)}' && floatspan '[1, 6)';
-- true
```

- Does the first value contain the second one?

  `{set,spans} @> {base,set,spans} → boolean`

```
SELECT floatset '{1.5, 2.5}' @> 2.5;
-- true
SELECT tstzspan '[2001-01-01, 2001-05-01)' @> timestamptz '2001-02-01';
-- true
SELECT floatspanset '{[1, 2),(2, 3)}' @> 2.0;
-- false
```

- Is the first value contained by the second one?

  `{base,set,spans} <@ {set,spans} → boolean`

```
SELECT timestamptz '2001-01-10' <@ tstzspan '[2001-01-01, 2001-05-01)';
-- true
SELECT floatspan '[2, 5]' <@ floatspan '[1, 5)';
-- false
SELECT tstzspan '[2001-02-01, 2001-03-01)' <@ tstzspan '[2001-01-01, 2001-05-01)';
-- true
SELECT floatspanset '{[1,2],[3,4]}' <@ floatspan '[1, 6]';
-- true
```

- Is the first value adjacent to the second one?

  `spans -|- spans → boolean`

```
SELECT intspan '[2, 6)' -|- intspan '[6, 7)';
-- true
SELECT floatspan '[2, 5)' -|- floatspan '(5, 6)';
-- false
SELECT floatspanset '{[2, 3],[4, 5)}' -|- floatspan '(5, 6)';
-- true
SELECT tstzspanset '{[2001-01-01, 2001-01-02]}' -|- tstzspan '[2001-01-02, 2001-01-03)';
-- false
```

2.8.2 Position Operations

The position operations available for set and span types are given next. Notice that the operators for time types have an additional # to distinguish them from the operators for number types.

- Is the first value strictly left of the second one?

  `numbers << numbers → boolean`

  `times <<# times → boolean`

```
SELECT intspan '[15, 20)' << 20;
-- true
SELECT intspanset '{[15, 17],[18, 20)}' << 20;
-- true
SELECT floatspan '[15, 20)' << floatspan '(15, 20)';
-- false
SELECT dateset '{2001-01-01, 2001-01-02}' <<# dateset '{2001-01-03, 2001-01-05}';
-- true
```

- Is the first value strictly to the right of the second one?

  `numbers >> numbers → boolean`

  `times #>> times → boolean`

```
SELECT intspan '[15, 20)' >> 10;
-- true
SELECT floatspan '[15, 20)' >> floatspan '[5, 10]';
-- true
SELECT floatspanset '{[15, 17], [18, 20)}' >> floatspan '[5, 10]';
-- true
SELECT tstzspan '[2001-01-04, 2001-01-05)' #>>
  tstzspanset '{[2001-01-01, 2001-01-04), [2001-01-05, 2001-01-06)}';
-- true
```

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

  `numbers &< numbers → boolean`

  `times &<# times → boolean`

```
SELECT intspan '[15, 20)' &< 18;
-- false
SELECT intspanset '{[15, 16],[17, 18)}' &< 18;
-- true
SELECT floatspan '[15, 20)' &< floatspan '[10, 20]';
-- true
SELECT dateset '{2001-01-02, 2001-01-05}' &<# dateset '{2001-01-01, 2001-01-04}';
-- false
```

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

  `numbers &> numbers → boolean`

  `times #&> times → boolean`

```
SELECT intspan '[15, 20)' &> 30;
-- true
SELECT floatspan '[1, 6]' &> floatspan '(1, 3)';
-- false
SELECT floatspanset '{[1, 2],[3, 4]}' &> floatspan '(1, 3)';
-- false
SELECT timestamp '2001-01-01' #&> tstzspan '[2001-01-01, 2001-01-05)';
-- true
```

2.8.3 Splitting Operations

When creating indexes for set or span set types, what is stored in the index is not the actual value but instead, a bounding box that *represents* the value. In this case, the index will provide a list of candidate values that *may* satisfy the query predicate, and a second step is needed to filter out candidate values by computing the query predicate on the actual values.

However, when the bounding boxes have a large empty space not covered by the actual values, the index will generate many candidate values that do not satisfy the query predicate, which reduces the efficiency of the index. In these situations, it may be better to represent a value not with a *single* bounding box, but instead with *multiple* bounding boxes. This increases considerably the efficiency of the index, provided that the index is able to manage multiple bounding boxes per value. The following functions are used for generating multiple spans from a single set or span set value.

- Return an array of N spans obtained by merging the elements of a set or the spans of a spanset

  `splitNSpans(set, integer) → span[]`

  `splitNSpans(spanset, integer) → span[]`

  The last argument specifies the number of output spans. If the number of input elements or spans is less than the given number, the resulting array will have one span per input element or span. Otherwise, the given number of output spans will be obtained by merging several consecutive input elements or spans.

```
SELECT splitNSpans(intset '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}', 1);
/* {"[1, 11)"} */
SELECT splitNSpans(intset '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}', 3);
-- {"[1, 5)","[5, 8)","[8, 11)"}
SELECT splitNSpans(intset '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}', 6);
-- {"[1, 3)","[3, 5)","[5, 7)","[7, 9)","[9, 10)","[10, 11)"}
SELECT splitNSpans(intset '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}', 12);
/* {"[1, 2)","[2, 3)","[3, 4)","[4, 5)","[5, 6)","[6, 7)","[7, 8)","[8, 9)",
    "[9, 10)","[10, 11)"} */
```

```
SELECT splitNSpans(intspanset '{[1, 2), [3, 4), [5, 6), [7, 8), [9, 10)}');
-- {"[1, 2)","[3, 4)","[5, 6)","[7, 8)","[9, 10)"}
SELECT splitNSpans(floatspanset '{[1, 2), [3, 4), [5, 6), [7, 8), [9, 10)}', 3);
-- {"[1, 4)","[5, 8)","[9, 10)"}
SELECT splitNSpans(datespanset '{[2000-01-01, 2000-01-04), [2000-01-05, 2000-01-10)}', 3);
-- {"[2000-01-01, 2000-01-04)","[2000-01-05, 2000-01-10)"}
```

- Return an array of spans obtained by merging N consecutive elements of a set or N consecutive spans of a spanset

  `splitEachNSpans(set, integer) → span[]`

  `splitEachNSpans(spanset, integer) → span[]`

  The last argument specifies the number of input elements that are merged to produce an output span. If the number of input elements is less than the given number, the resulting array will have one output span per element. Otherwise, the given number of consecutive input elements will be merged into a single output span in the answer. Notice that, contrary to the splitNSpans function, the number of spans in the result depends on the number of input elements or spans.

```
SELECT splitEachNSpans(intset '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}', 1);
/* {"[1, 2)","[2, 3)","[3, 4)","[4, 5)","[5, 6)","[6, 7)","[7, 8)","[8, 9)",
    "[9, 10)","[10, 11)"} */
SELECT splitEachNSpans(intspanset '{[1, 2), [3, 4), [5, 6), [7, 8), [9, 10)}', 3);
-- {"[1, 6)","[7, 10)"}
SELECT splitEachNSpans(intset '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}', 6);
-- {"[1, 7)","[7, 11)"}
SELECT splitEachNSpans(intset '{1, 2, 3, 4, 5, 6, 7, 8, 9, 10}', 12);
-- {"[1, 11)"}
```

2.9 Distance Operations

The distance operator <-> for set and span types consider the bounding span and returns a the smallest distance between the two values. In the case of time values, the operator returns the number of days or the number of seconds between the two time values. The distance operator can also be used for nearest neighbor searches using a GiST or an SP-GiST index (see Section 2.12).

- Return the smallest distance ever

  `numbers <-> numbers → base`

  `dates <-> dates → integer`

  `times <-> times → float`

```
SELECT 3 <-> intspan '[6, 8)';
-- 3
SELECT floatspan '[1, 3]' <-> floatspan '(5.5, 7]';
-- 2.5
SELECT floatspan '[1, 3]' <-> floatspanset '{(5.5, 7],[8, 9]}';
-- 2.5
SELECT tstzspan '[2001-01-02, 2001-01-06)' <-> timestamptz '2001-01-07';
-- 86400
SELECT dateset '{2001-01-01, 2001-01-03, 2001-01-05}' <->
```

```
  dateset '{2001-01-02, 2001-01-04}';
--  0
```

2.10 Comparisons

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 set and span types. For span values, the operators compare first the lower bound, then the upper bound. For set and span set values, the operators compare first the bounding spans, and if those are equal, they compare the first N values or spans, where N is the minimum of the number of composing values or spans of both values.

The comparison operators available for the set and span types are given next. Recall that integer spans are always represented by their canonical form.

- Traditional comparisons

  `set {=, <>, <, >, <=, >=} set → boolean`

  `spans {=, <>, <, >, <=, >=} spans → boolean`

  `set_cmp(set,set) → int`

  `span_cmp(span,span) → int`

  `spanset_cmp(spanset,spanset) → int`

  Functions `set_cmp`, `span_cmp`, and `spanset_cmp` return -1, 0, or 1 depending on whether the first argument is, respectively, less than, equal to, or greater than the second one.

```
SELECT intspan '[1,3]' = intspan '[1,4)';
-- true
SELECT floatspanset '{[1, 2),[2,3)}' = floatspanset '{[1,3)}';
-- true
SELECT tstzset '{2001-01-01, 2001-01-04}' <> tstzset '{2001-01-01, 2001-01-05}';
-- false
SELECT tstzspan '[2001-01-01, 2001-01-04)' <> tstzspan '[2001-01-03, 2001-01-05)';
-- true
SELECT floatspan '[3, 4]' < floatspan '(3, 4]';
-- true
SELECT intspanset '{[1,2],[3,4]}' < intspanset '{[3, 4]}';
-- true
SELECT floatspan '[3, 4]' > floatspan '[3, 4)';
-- true
SELECT tstzspan '[2001-01-03, 2001-01-04)' > tstzspan '[2001-01-02, 2001-01-05)';
-- true
SELECT floatspanset '{[1, 4)}' <= floatspanset '{[1, 5), [6, 7)}';
-- true
SELECT tstzspanset '{[2001-01-01, 2001-01-04)}' <=
  tstzspanset '{[2001-01-01, 2001-01-05), [2001-01-06, 2001-01-07)}';
-- true
SELECT tstzspan '[2001-01-03, 2001-01-05)' >= tstzspan '[2001-01-03, 2001-01-04)';
-- true
SELECT intspanset '{[1, 4)}' >= intspanset '{[1, 5), [6, 7)}';
-- false
SELECT spanset_cmp(intspanset '{[1, 4)}', intspanset '{[1, 5), [6, 7)}');
-- -1
```

2.11 Aggregations

There are several aggregate functions defined for set and span types. They are described next.

- Function `extent` returns a bounding span that encloses a set of set or span values.
- Union is a very useful operation for set and span types. As we have seen in Section 2.7, we can compute the union of two set or span 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. Functions `setUnion` and `spanUnion` can be used for this purpose.
- Function `tCount` generalizes the traditional aggregate function `count`. The temporal count can be used to compute at each point in time the number of available objects (for example, number of spans). Function `tCount` returns a temporal integer (see Chapter 4). The function has two optional parameters that specify the granularity (an `interval`) and the origin of time (a `timestamptz`). When these parameters are given, the temporal count is computed at time bins of the given granularity (see Section 9.5).

- Bounding span

  `extent({set,spans}) → span`

```
WITH spans(r) AS (
  SELECT floatspan '[1, 4)' UNION SELECT floatspan '(5, 8)' UNION
  SELECT floatspan '(7, 9)' )
SELECT extent(r) FROM spans;
-- [1,9)
WITH times(ts) AS (
  SELECT tstzset '{2001-01-01, 2001-01-03, 2001-01-05}' UNION
  SELECT tstzset '{2001-01-02, 2001-01-04, 2001-01-06}' UNION
  SELECT tstzset '{2001-01-01, 2001-01-02}' )
SELECT extent(ts) FROM times;
-- [2001-01-01, 2001-01-06]
WITH periods(ps) AS (
  SELECT tstzspanset '{[2001-01-01, 2001-01-02], [2001-01-03, 2001-01-04]}' UNION
  SELECT tstzspanset '{[2001-01-01, 2001-01-04], [2001-01-05, 2001-01-06]}' UNION
  SELECT tstzspanset '{[2001-01-02, 2001-01-06]}' )
SELECT extent(ps) FROM periods;
-- [2001-01-01, 2001-01-06]
```

- Aggregate union

  `setUnion({value,set}) → set`

  `spanUnion(spans) → spanset`

```
WITH times(ts) AS (
  SELECT tstzset '{2001-01-01, 2001-01-03, 2001-01-05}' UNION
  SELECT tstzset '{2001-01-02, 2001-01-04, 2001-01-06}' UNION
  SELECT tstzset '{2001-01-01, 2001-01-02}' )
SELECT setUnion(ts) FROM times;
-- {2001-01-01, 2001-01-02, 2001-01-03, 2001-01-04, 2001-01-05, 2001-01-06}
WITH periods(ps) AS (
  SELECT tstzspanset '{[2001-01-01, 2001-01-02], [2001-01-03, 2001-01-04]}' UNION
  SELECT tstzspanset '{[2001-01-02, 2001-01-03], [2001-01-05, 2001-01-06]}' UNION
  SELECT tstzspanset '{[2001-01-07, 2001-01-08]}' )
SELECT spanUnion(ps) FROM periods;
-- {[2001-01-01, 2001-01-04], [2001-01-05, 2001-01-06], [2001-01-07, 2001-01-08]}
```

- Temporal count

  `tCount(times) → {tintSeq,tintSeqSet}`

```
WITH times(ts) AS (
  SELECT tstzset '{2001-01-01, 2001-01-03, 2001-01-05}' UNION
  SELECT tstzset '{2001-01-02, 2001-01-04, 2001-01-06}' UNION
  SELECT tstzset '{2001-01-01, 2001-01-02}' )
SELECT tCount(ts) FROM times;
-- {2@2001-01-01, 2@2001-01-02, 1@2001-01-03, 1@2001-01-04, 1@2001-01-05, 1@2001-01-06}
```

```
WITH periods(ps) AS (
  SELECT tstzspanset '{[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-04)}' UNION
  SELECT tstzspanset '{[2001-01-01, 2001-01-04), [2001-01-05, 2001-01-06)}' UNION
  SELECT tstzspanset '{[2001-01-02, 2001-01-06)}' )
SELECT tCount(ps) FROM periods;
-- {[2@2001-01-01, 3@2001-01-03, 1@2001-01-04, 2@2001-01-05, 2@2001-01-06)}
```

2.12 Indexing

GiST and SP-GiST indexes can be created for table columns of the set and span 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 `tstzspan` in a table `Reservation` is as follows:

```
CREATE TABLE Reservation (ReservationID integer PRIMARY KEY, RoomID integer,
  During tstzspan);
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 set or span types. For these index types, basically the only useful operation is equality. There is a B-tree sort ordering defined for values of span 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.

Finally, hash indexes can be created for table columns of a set or span types. For these types of indexes, the only operation defined is equality.

Chapter 3

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`.

3.1 Input and Output

MobilityDB generalizes Open Geospatial Consortium's Well-Known Text (WKT) and Well-Known Binary (WKB) input and output format for all temporal types. We present next the functions for input and output box types.

A `tbox` is composed of a numeric and/or time dimensions. For each dimension, a span given, that is, either an `intspan` or a `floatspan` for the value dimension and a `tstzspan` for the time dimension. Examples of input of `tbox` values are as follows:

```
-- Both value and time dimensions
SELECT tbox 'TBOXINT XT([1,3),[2001-01-01,2001-01-02])';
SELECT tbox 'TBOXFLOAT XT([1.5,2.5],[2001-01-01,2001-01-02])';
-- Only value dimension
SELECT tbox 'TBOXINT X([1,3))';
SELECT tbox 'TBOXFLOAT X((1.5,2.5))';
-- Only time dimension
SELECT tbox 'TBOX T((2001-01-01,2001-01-02))';
```

An `stbox` is composed of a spatial and/or time dimensions, where the coordinates of the spatial dimension may be 2D or 3D. For the time dimension a `tstzspan` is given and for the spatial dimension minimum and maximum coordinate values are given, where the latter 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. Geodetic boxes always have a Z dimension to account for the curvature of the underlying sphere or spheroid. Examples of input of `stbox` values are as follows:

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

We give next the functions for input and output of box types in Well-Known Text and Well-Known Binary format.

- Return the Well-Known Text (WKT) representation

  `asText(box,maxdecdigits=15) → text`

  The `maxdecdigits` argument can be used to set the maximum number of decimal places in the output of floating point values (default 15).

```
SELECT asText(tbox 'TBOXFLOAT XT([1.123456789,2.123456789),[2001-01-01,2001-01-02))', 3);
-- TBOXFLOAT XT([1.123, 2.123),[2001-01-01 00:00:00+01, 2001-01-02 00:00:00+01))
SELECT asText(stbox 'STBOX Z((1.55,1.55,1.55),(2.55,2.55,2.55))', 0);
-- STBOX Z((2,2,2),(3,3,3))
```

- Return the Well-Known Binary (WKB) or the Hexadecimal Well-Known Binary (HexWKB) representation

  `asBinary(box,endian text='') → bytea`

  `asHexWKB(box,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 the encoding of the machine is used.

```
SELECT asBinary(tbox 'TBOXFLOAT XT([1,2),[2001-01-01,2001-01-02))');
-- \x0103270001009c57d3c11c000000fc2ef1d51c00000d0001000000000000f03f0000000000000040
SELECT asBinary(tbox 'TBOXFLOAT XT([1,2),[2001-01-01,2001-01-02))', 'XDR');
-- \x000300270100001cc1d3579c0000001cd5f12efc00000d013ff00000000000040000000000000000
SELECT asBinary(stbox 'STBOX X((1,1),(2,2))');
-- \x0101000000000000f03f0000000000000040000000000000f03f0000000000000040
SELECT asHexWKB(tbox 'TBOXFLOAT XT([1,2),[2001-01-01,2001-01-02))');
-- 0103270001009C57D3C11C000000FC2EF1D51C00000D0001000000000000F03F0000000000000040
SELECT asHexWKB(tbox 'TBOXFLOAT XT([1,2)[2001-01-01,2001-01-02))', 'XDR');
-- 000300270100001CC1D3579C0000001CD5F12EFC00000D013FF00000000000040000000000000000
SELECT asHexWKB(stbox 'STBOX X((1,1),(2,2))');
-- 0101000000000000F03F0000000000000040000000000000F03F0000000000000040
```

- Input from the Well-Known Binary (WKB) or from the Hexadecimal Well-Known Binary (HexWKB) representation

  `boxFromBinary(bytea) → box`

  `boxFromHexWKB(text) → box`

  In the signatures above, `box` replaces any box type, that is, `tbox` or `stbox`.

```
SELECT tboxFromBinary(
  '\x0103270001009c57d3c11c000000fc2ef1d51c00000d0001000000000000f03f0000000000000040');
-- TBOXFLOAT XT([1,2),[2001-01-01,2001-01-02))
SELECT stboxFromBinary(
  '\x0101000000000000f03f0000000000000040000000000000f03f0000000000000040');
--  STBOX X((1,1),(2,2))
SELECT tboxFromHexWKB(
  '0103270001009C57D3C11C000000FC2EF1D51C00000D0001000000000000F03F0000000000000040');
-- TBOXFLOAT XT([1,2),[2001-01-01,2001-01-02)))
SELECT stboxFromHexWKB(
  '0101000000000000F03F0000000000000040000000000000F03F0000000000000040');
-- STBOX X((1,1),(2,2))
```

3.2 Constructors

Type `tbox` has several constructor functions depending on whether the value and/or the time extent are given. The value extent can be specified by a number or a span, while the time extent can be specified by a time type.

- Constructor for `tbox`

  `tbox({number,numspan}) → tbox`

  `tbox({timestamptz,tstzspan}) → tbox`

  `tbox({number,numspan},{timestamptz,tstzspan}) → tbox`

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

Type `stbox` has several constructor functions depending on whether the space and/or the time extent are given. The coordinates for the spatial extent can be 2D or 3D and can be either Cartesian or geodetic. The spatial extent can be specified by the minimum and maximum coordinate 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. The spatial extent can also be specified by a geometry or a geography. The temporal extent can be specified by a time type.

- Constructor for `stbox`

  `stboxX(float,float,float,float,srid=0) → stbox`

  `stboxZ(float,float,float,float,float,float,srid=0) → stbox`

  `stboxT({timestamptz,tstzspan}) → stbox`

  `stboxXT(float,float,float,float,{timestamptz,tstzspan},srid=0) → stbox`

  `stboxZT(float,float,float,float,float,float,{timestamptz,tstzspan},srid=0) → stbox`

  `geodstboxZ(float,float,float,float,float,float,srid=4326) → stbox`

  `geodstboxT({timestamptz,tstzspan}) → stbox`

  `geodstboxZT(float,float,float,float,float,float,{timestamptz,tstzspan},srid=4326) → stbox`

  `stbox(geo) → stbox`

  `stbox(geo,{timestamptz,tstzspan}) → stbox`

```
-- Only value dimension with X and Y coordinates
SELECT stboxX(1.0,2.0,1.0,2.0);
-- Only value dimension with X, Y, and Z coordinates
SELECT stboxZ(1.0,2.0,3.0,1.0,2.0,3.0);
-- Only value dimension with X, Y, and Z coordinates and SRID
SELECT stboxZ(1.0,2.0,3.0,1.0,2.0,3.0,5676);
-- Only time dimension
SELECT stboxT(tstzspan '[2001-01-03,2001-01-03]');
-- Both value (with X and Y coordinates) and time dimensions
SELECT stboxXT(1.0,2.0, 1.0,2.0, tstzspan '[2001-01-03,2001-01-03]');
-- Both value (with X, Y, and Z coordinates) and time dimensions
SELECT stboxZT(1.0,2.0,3.0, 1.0,2.0,3.0, tstzspan '[2001-01-03,2001-01-03]');
-- Only value dimension with X, Y, and Z geodetic coordinates
SELECT geodstboxZ(1.0,2.0,3.0,1.0,2.0,3.0);
-- Only time dimension for geodetic box
SELECT geodstboxT(tstzspan '[2001-01-03,2001-01-03]');
```

```
--  Both value (with X, Y, and Z geodetic coordinates) and time dimensions
SELECT geodstboxZT(1.0,2.0,3.0, 1.0,2.0,3.0, tstzspan '[2001-01-03,2001-01-04]');
-- Geometry and time dimension
SELECT stbox(geometry 'Linestring(1 1 1,2 2 2)', tstzspan '[2001-01-03,2001-01-05]');
-- Geography and time dimension
SELECT stbox(geography 'Linestring(1 1 1,2 2 2)', tstzspan '[2001-01-03,2001-01-05]');
```

3.3 Conversions

- Convert a `tbox` to another type

  `tbox::{intspan,floatspan,tstzspan}`

  `intspan(tbox)` → `tbox`

  `floatspan(tbox)` → `tbox`

  `tstzspan(tbox)` → `tbox`

```
SELECT tbox 'TBOXINT XT([1,4),[2001-01-01,2001-01-02))'::intspan;
-- [1,4)
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02))'::floatspan;
-- (1, 2)
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02))'::tstzspan;
-- [2001-01-01, 2001-01-02)
```

- Convert another type to a `tbox`

  `{numbers,times,tnumber}::tbox`

  `tbox({numbers,times,tnumber})` → `tbox`

```
SELECT intset '{1,2}'::tbox;
-- TBOXINT X([1, 3))
SELECT intspan '[1,3)'::tbox;
-- TBOXINT X([1, 3))
SELECT floatspan '(1.0,2.0)'::tbox;
-- TBOXFLOAT X((1, 2))
SELECT tstzspanset '{(2001-01-01,2001-01-02),(2001-01-03,2001-01-04)}'::tbox;
-- TBOX T((2001-01-01,2001-01-04))
```

- Convert an `stbox` to a another type

  `stbox::{box2d,box3d,geo,tstzspan}`

  `box2d(stbox)` → `box2d`

  `box3d(stbox)` → `box2d`

  `geometry(stbox)` → `geometry`

  `geography(stbox)` → `geography`

  `tstzspan(stbox)` → `tstzspan`

```
SELECT stbox 'STBOX XT(((1.0,2.0),(3.0,4.0)),[2001-01-01,2001-01-03])'::box2d;
-- BOX(1 2,3 4)
SELECT ST_AsEWKT(stbox 'SRID=4326;STBOX XT(((1,1),(5,5)),[2001-01-01,2001-01-05])'::
  geometry);
-- SRID=4326;POLYGON((1 1,1 5,5 5,5 1,1 1))
SELECT ST_AsEWKT(stbox 'STBOX XT(((1,1),(1,5)),[2001-01-01,2001-01-05])'::geometry);
-- LINESTRING(1 1,1 5)
SELECT ST_AsEWKT(stbox 'GEODSTBOX XT(((1,1),(1,1)),[2001-01-01,2001-01-05])'::geography);
-- SRID=4326;POINT(1 1)
SELECT ST_AsEWKT(stbox 'STBOX ZT(((1,1,1),(5,5,5)),[2001-01-01,2001-01-05])'::
  geometry);
```

```
/* 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))) */
SELECT stbox 'STBOX XT(((1.0,2.0),(3.0,4.0)),[2001-01-01,2001-01-03])'::tstzspan;
-- [2001-01-01, 2001-01-03]
```

- Convert another type to an `stbox`

  `{box2d,box3d,geo,time,tgeo}::stbox`

  `stbox({box2d,box3d,geo,time,tgeo}) → stbox`

```
SELECT geometry 'Linestring(1 1 1,2 2 2)'::box3d::stbox;
-- STBOX Z((1,1,1),(2,2,2))
SELECT geography 'Linestring(1 1,2 2)'::stbox;
-- SRID=4326;GEODSTBOX X((1,1),(2,2))
SELECT tstzspanset '{(2001-01-01,2001-01-02),(2001-01-03,2001-01-04)}'::stbox;
-- STBOX T((2001-01-01,2001-01-04))
```

3.4 Accessors

- Has X/Z/T dimension?

  `hasX(box) → boolean`

  `hasZ(stbox) → boolean`

  `hasT(box) → boolean`

```
SELECT hasX(tbox 'TBOX T([2001-01-01,2001-01-03))');
-- false
SELECT hasX(stbox 'STBOX X((1.0,2.0),(3.0,4.0))');
-- true
SELECT hasZ(stbox 'STBOX X((1.0,2.0),(3.0,4.0))');
-- false
SELECT hasT(tbox 'TBOXFLOAT XT((1.0,3.0),[2001-01-01,2001-01-03])');
-- true
SELECT hasT(stbox 'STBOX X((1.0,2.0),(3.0,4.0))');
-- false
```

- Is geodetic?

  `isGeodetic(stbox) → boolean`

```
SELECT isGeodetic(stbox 'GEODSTBOX Z((1.0,1.0,0.0),(3.0,3.0,1.0))');
-- true
SELECT isGeodetic(stbox 'STBOX XT(((1.0,2.0),(3.0,4.0)),[2001-01-01,2001-01-02])');
-- false
```

- Return the minimum X/Y/Z/T value

  `xMin(box) → float`

  `yMin(stbox) → float`

  `zMin(stbox) → float`

  `tMin(box) → timestamptz`

```
SELECT xMin(tbox 'TBOXFLOAT XT((1.0,3.0),[2001-01-01,2001-01-03))');
-- 1
SELECT yMin(stbox 'STBOX X((1.0,2.0),(3.0,4.0))');
-- 2
```

```
SELECT zMin(stbox 'STBOX Z((1.0,2.0,3.0),(4.0,5.0,6.0))');
-- 3
SELECT tMin(stbox 'GEODSTBOX T([2001-01-01,2001-01-03))');
-- 2001-01-01
```

Notice that for `tbox` the result value for `xMin` and `xMin` is converted to a `float` for the temporal boxes with an integer span.

- Return the maximum X/Y/Z/T value

  `xMax(box) → float`

  `yMax(stbox) → float`

  `zMax(stbox) → float`

  `tMax(box) → timestamptz`

```
SELECT xMax(tbox 'TBOXINT X([1,4))');
-- 3
SELECT yMax(stbox 'STBOX X((1.0,2.0),(3.0,4.0))');
-- 4
SELECT zMax(stbox 'STBOX Z((1.0,2.0,3.0),(4.0,5.0,6.0))');
-- 6
SELECT tMax(stbox 'GEODSTBOX T([2001-01-01,2001-01-03))');
-- 2001-01-03
```

Notice that for `tbox` the result value for `xMin` and `xMin` is converted to a `float` for the temporal boxes with an integer span.

- Is the minimum X/T value inclusive?

  `xMinInc(tbox) → bool`

  `tMinInc(box) → bool`

```
SELECT xMinInc(tbox 'TBOXFLOAT XT((1.0,3.0),[2001-01-01,2001-01-03))');
-- false
SELECT tMinInc(stbox 'GEODSTBOX T([2001-01-01,2001-01-03))');
-- true
```

- Is the maximum X/T value inclusive?

  `xMaxInc(tbox) → bool`

  `tMaxInc(box) → bool`

```
SELECT xMaxInc(tbox 'TBOXFLOAT XT((1.0,3.0),[2001-01-01,2001-01-03))');
-- false
SELECT tMaxInc(stbox 'GEODSTBOX T([2001-01-01,2001-01-03))');
-- true
```

- Area, volume, perimeter

  `area(stbox, spheroid bool=true) → float`

  `volume(stbox) → float`

  `perimeter(stbox, spheroid bool=true) → float`

  For geodetic boxes, the computation is done on the WGS 84 spheroid by default. If the last argument is false, a faster spherical calculation is used. The `volume` function do not accept geodetic boxes.

```
SELECT area(stbox 'STBOX XT(((1,1),(3,3)),[2001-01-01,2001-01-03))');
-- 4
SELECT volume(stbox 'STBOX ZT(((1,1,1),(3,3,3)),[2001-01-01,2001-01-03))');
-- 8
SELECT perimeter(stbox 'STBOX XT(((1,1),(3,3)),[2001-01-01,2001-01-03))');
-- 8
SELECT area(stbox 'GEODSTBOX XT(((1,1),(3,3)),[2001-01-01,2001-01-03))');
-- 49209676328.36632
SELECT perimeter(stbox 'GEODSTBOX XT(((1,1),(3,3)),[2001-01-01,2001-01-03))', false);
-- 889221.9544681838
```

3.5 Transformations

- Shift and/or scale the span of a bounding box by one or two values

  `shiftValue(box,{integer,float}) → box`

  `scaleValue(box,{integer,float}) → box`

  `shiftScaleValue(tbox,{integer,float},{integer,float}) → box`

```
SELECT shiftValue(tbox 'TBOXFLOAT XT([1.5, 2.5],[2001-01-01,2001-01-02])', 1.0);
-- TBOXFLOAT XT([2.5, 3.5],[2001-01-01, 2001-01-02])
SELECT scaleValue(tbox 'TBOXFLOAT XT([1.5, 2.5],[2001-01-01,2001-01-02])', 2.0);
-- TBOXFLOAT XT([1.5, 3.5],[2001-01-01, 2001-01-02])
SELECT shiftScaleValue(tbox 'TBOXFLOAT XT([1.5, 2.5],[2001-01-01,2001-01-02])', 2.0, 3.0);
-- TBOXFLOAT XT([3.5, 6.5],[2001-01-01, 2001-01-02])
```

- Shift and/or scale the span or the period of the bounding box to a value or interval

  `shiftTime(box,interval) → box`

  `scaleTime(box,interval) → box`

  `shiftScaleTime(box,interval,interval) → box`

  For scaling, if the width or the time span of the period is zero (that is, the lower and upper bound are equal), the result is the box. The given value or interval must be strictly greater than zero.

```
SELECT shiftTime(tbox 'TBOXFLOAT XT([1.5, 2.5],[2001-01-01,2001-01-02])',
  interval '1 day');
-- TBOXFLOAT XT([1.5, 2.5],[2001-01-02, 2001-01-03])
SELECT shiftTime(stbox 'STBOX T([2001-01-01,2001-01-02])', interval '-1 day');
-- STBOX T([2001-12-31, 2001-01-01])
SELECT shiftTime(stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-02])',
  interval '1 day');
-- STBOX ZT(((1,1,1),(2,2,2)),[2001-01-02, 2001-01-03])
SELECT scaleTime(tbox 'TBOXFLOAT XT([1.5, 2.5],[2001-01-01,2001-01-02])',
  interval '2 days');
-- TBOXFLOAT XT([1.5, 2.5],[2001-01-01, 2001-01-03])
SELECT scaleTime(stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-02])',
  interval '1 hour');
-- STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01 00:00:00, 2001-01-01 01:00:00])
SELECT scaleTime(stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-02])',
  interval '-1 day');
-- ERROR:  The interval must be positive: -1 days
SELECT shiftScaleTime(tbox 'TBOXFLOAT XT([1.5, 2.5],[2001-01-01,2001-01-02])',
  interval '1 day', interval '3 days');
-- TBOXFLOAT XT([1.5, 2.5],[2001-01-02, 2001-01-05])
SELECT shiftScaleTime(stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-02])',
  interval '1 hour', interval '3 hours');
-- STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01 01:00:00, 2001-01-01 04:00:00])
```

- Return the spatial dimension of the bounding box, removing the temporal dimension if any

  `getSpace(stbox) → stbox`

```
SELECT getSpace(stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-03])');
-- STBOX Z((1,1,1),(2,2,2))
```

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, spatial, or temporal dimension of a bounding box by a value or an interval

  `expandValue(tbox,{integer,float}) → tbox`

  `expandSpace(stbox,float) → stbox`

  `expandTime(box,interval) → box`

  The function returns NULL if the value or interval given as second argument is negative and the span resulting from shifting the bounds with the argument is empty.

```
SELECT expandValue(tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-03])', 1.0);
-- TBOXFLOAT XT((0,3),[2001-01-01,2001-01-03])
SELECT expandValue(tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-03])', -1.0);
-- NULL
SELECT expandValue(tbox 'TBOX T([2001-01-01,2001-01-03))', 1);
-- The box must have value dimension
```

```
SELECT expandSpace(stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-03])', 1);
-- STBOX ZT(((0,0,0),(3,3,3)),[2001-01-01,2001-01-03])
SELECT expandSpace(stbox 'STBOX T([2001-01-01,2001-01-03))', 1);
-- The box must have space dimension
```

```
SELECT expandTime(tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-03])', interval '1 day');
-- TBOXFLOAT XT((1,2),[2000-12-31,2001-01-04])
SELECT expandTime(stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-03])',
  interval '-1 day');
-- STBOX ZT(((1,1,1),(2,2,2)),[2001-01-02,2001-01-02])
SELECT expandTime(tbox 'TBOX XT((1,2),[2001-01-01,2001-01-03])', interval '-2 days');
-- NULL
```

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

  `round(box,integer=0) → box`

```
SELECT round(tbox 'TBOXFLOAT XT((1.12345,2.12345),[2001-01-01,2001-01-02])', 2);
-- TBOXFLOAT XT((1.12,2.12),[2001-01-01, 2001-01-02])
SELECT round(stbox 'STBOX XT(((1.12345, 1.12345),(2.12345, 2.12345)),
  [2001-01-01,2001-01-02])', 2);
-- STBOX XT(((1.12,1.12),(2.12,2.12)),[2001-01-01, 2001-01-02])
SELECT round(tstzspan '[2000-01-01, 2001-01-02]'::tbox);
-- The tbox must have X dimension
```

3.6 Spatial Reference System

- Return or set the spatial reference identifier

  `SRID(stbox) → integer`

  `setSRID(stbox) → stbox`

```
SELECT SRID(stbox 'STBOX ZT(((1.0,2.0,3.0),(4.0,5.0,6.0)),[2001-01-01,2001-01-02])');
-- 0
SELECT SRID(stbox 'SRID=5676;STBOX XT(((1.0,2.0),(4.0,5.0)),[2001-01-01,2001-01-02])');
-- 5676
SELECT SRID(stbox 'GEODSTBOX T([2001-01-01,2001-01-02))');
-- ERROR:  The box must have space dimension
```

```
SELECT setSRID(stbox 'STBOX ZT(((1.0,2.0,3.0),(4.0,5.0,6.0)),
  [2001-01-01,2001-01-02])', 5676);
-- SRID=5676;STBOX ZT(((1,2,3),(4,5,6)),[2001-01-01,2001-01-02])
```

- Transform to a spatial reference identifier

  `transform(stbox,to_srid integer) → stbox`

  `transformPipeline(stbox,pipeline text,to_srid integer,is_forward bool=true) → stbox`

  The `transform` function specifies the transformation with a target SRID. An error is raised when the input box has an unknown SRID (represented by 0). The `transformPipeline` function specifies the transformation with a defined coordinate transformation pipeline represented with the following string format:

  `urn:ogc:def:coordinateOperation:AUTHORITY::CODE`

  The SRID of the input box is ignored, and the SRID of the output box will be set to zero unless a value is provided via the optional `to_srid` parameter. As stated by the last parameter, the pipeline is executed by default in a forward direction; by setting the parameter to false, the pipeline is executed in the inverse direction.

```
SELECT round(transform(stbox 'SRID=4326;STBOX XT(((2.340088, 49.400250),
  (6.575317, 51.553167)),[2001-01-01,2001-01-02])', 3812), 6);
/* SRID=3812;STBOX XT(((502773.429981,511805.120402),(803028.908265,751590.742629)),
  [2001-01-01, 2001-01-02]) */
WITH test(box, pipeline) AS (
  SELECT stbox 'SRID=4326;GEODSTBOX Z((-0.1275,50.846667,100),(4.3525,51.507222,100))',
    text 'urn:ogc:def:coordinateOperation:EPSG::16031' )
SELECT asEWKT(transformPipeline(transformPipeline(box, pipeline, 4326),
  pipeline, 4326, false), 6)
FROM test;
-- SRID=4326;GEODSTBOX Z((-0.1275,50.846667,100),(4.3525,51.507222,100))
```

3.7 Splitting Operations

- Split the bounding box in quadrants or octants {}

  `quadSplit(stbox) → {stbox}`

  As indicated by the {} symbol, this function is a *set-returning function* (also known as a *table function*) since it typically return more than one value.

```
SELECT quadSplit(stbox 'STBOX XT(((0,0),(4,4)),[2001-01-01,2001-01-05])');
/* {"STBOX XT(((0,0),(2,2)),[2001-01-01, 2001-01-05])",
   "STBOX XT(((2,0),(4,2)),[2001-01-01, 2001-01-05])",
   "STBOX XT(((0,2),(2,4)),[2001-01-01, 2001-01-05])",
   "STBOX XT(((2,2),(4,4)),[2001-01-01, 2001-01-05])"} */
SELECT quadSplit(stbox 'STBOX Z((0,0,0),(4,4,4))');
/* {"STBOX Z((0,0,0),(2,2,2))","STBOX Z((2,0,0),(4,2,2))","STBOX Z((0,2,0),(2,4,2))",
   "STBOX Z((2,2,0),(4,4,2))","STBOX Z((0,0,2),(2,2,4))","STBOX Z((2,0,2),(4,2,4))",
   "STBOX Z((0,2,2),(2,4,4))","STBOX Z((2,2,2),(4,4,4))"} */
```

This function is typically used for multiresolution grids, where the space is split in cells such that the cells have a maximum number of elements. Figure 3.1 shows an example of the result of using this function using synthetic trajectories in Brussels.

3.8 Set Operations

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.

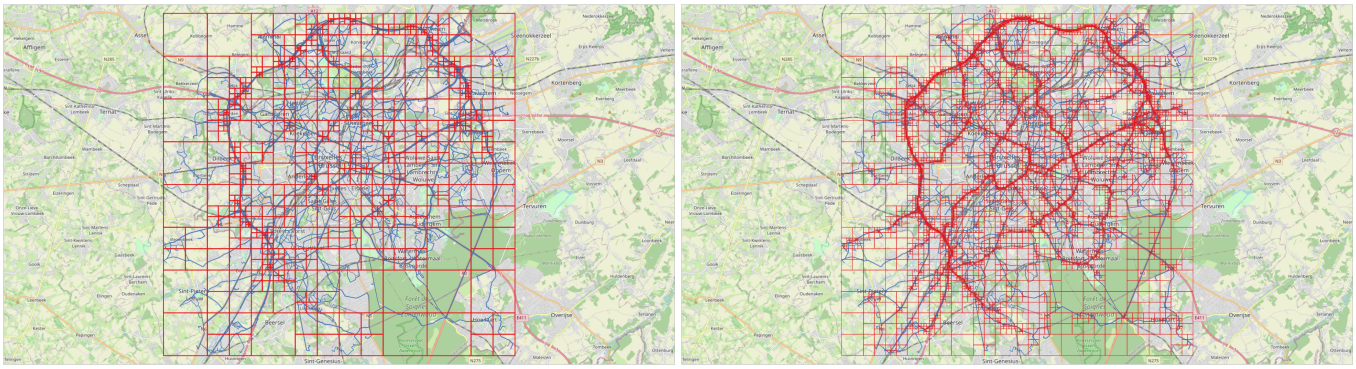


Figure 3.1: Multiresolution grid on Brussels data obtained using the BerlinMOD generator. Each cell contains at most 10,000 (left) and 1,000 (right) instants across the entire simulation period (four days in this case). On the left, we can see the high density of the traffic in the ring around Brussels, while on the right we can see other main axes in the city.

- Union and intersection of two bounding boxes

`box {+, *} box → box`

```
SELECT tbox 'TBOXINT XT([1,3),[2001-01-01,2001-01-03])' +
  tbox 'TBOXINT XT([2,4),[2001-01-02,2001-01-04])';
-- TBOXINT XT([1,4),[2001-01-01,2001-01-04])
SELECT stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-02])' +
  stbox 'STBOX XT(((2,2),(3,3))),[2001-01-01,2001-01-03]';
-- ERROR:  The arguments must be of the same dimensionality
SELECT tbox 'TBOXFLOAT XT((1,3),[2001-01-01,2001-01-02])' +
  tbox 'TBOXFLOAT XT((3,4),[2001-01-03,2001-01-04])';
-- ERROR:  Result of box union would not be contiguous
```

```
SELECT tbox 'TBOXINT XT([1,3),[2001-01-01,2001-01-03])' *
  tbox 'TBOX T([2001-01-02,2001-01-04))';
-- TBOX T([2001-01-02,2001-01-03))
SELECT stbox 'STBOX ZT(((1,1,1),(3,3,3)),[2001-01-01,2001-01-02])' *
  stbox 'STBOX X((2,2),(4,4))';
-- STBOX X((2,2),(3,3))
```

3.9 Bounding Box Operations

3.9.1 Topological Operations

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.

- Do the bounding boxes overlap?

`box && box → boolean`

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

- Does the first bounding box contain the second one?

  `box @> box → boolean`

```
SELECT tbox 'TBOXFLOAT XT((1,4),[2001-01-01,2001-01-04])' @>
  tbox 'TBOXFLOAT XT((2,3),[2001-01-01,2001-01-02])';
-- true
SELECT stbox 'STBOX Z((1,1,1),(3,3,3))' @>
  stbox 'STBOX XT(((1,1),(2,2)),[2001-01-01,2001-01-02])';
-- true
```

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

  `box <@ box → boolean`

```
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])' <@
  tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])';
-- true
SELECT stbox 'STBOX XT(((1,1),(2,2)),[2001-01-01,2001-01-02])' <@
  stbox 'STBOX ZT(((1,1,1),(2,2,2)),[2001-01-01,2001-01-02])';
-- true
```

- Are the bounding boxes equal in their common dimensions?

  `box ~= box → boolean`

```
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])' ~=
  tbox 'TBOXFLOAT T([2001-01-01,2001-01-02])';
-- true
SELECT stbox 'STBOX XT(((1,1),(3,3)),[2001-01-01,2001-01-03])' ~=
  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 'TBOXINT XT([1,2),[2001-01-01,2001-01-02])' -|-
  tbox 'TBOXINT XT([2,3),[2001-01-02,2001-01-03])';
-- true
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])' -|-
  tbox 'TBOX T([2001-01-02,2001-01-03])';
-- true
SELECT stbox 'STBOX XT(((1,1),(3,3)),[2001-01-01,2001-01-03])' -|-
  stbox 'STBOX XT(((2,2),(4,4)),[2001-01-03,2001-01-04])';
-- true
```

3.9.2 Position Operations

The position 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 dimension of the `tbox` type are given next.

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

  `tbox {<<, <<#} tbox → boolean`

  `stbox {<<, <<|, <</, <<#} stbox → boolean`

```
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])' <<
  tbox 'TBOXFLOAT XT((3,4),[2001-01-03,2001-01-04])';
-- true
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])' <<
  tbox 'TBOXFLOAT T([2001-01-03,2001-01-04])';
-- ERROR:  The box must have value dimension
SELECT stbox 'STBOX Z((1,1,1),(2,2,2))' <<| stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
SELECT stbox 'STBOX Z((1,1,1),(2,2,2))' <</ stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])' <<#
  tbox 'TBOXFLOAT XT((3,4),[2001-01-03,2001-01-04])';
-- true
```

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

  `tbox {>>, #>>} tbox → boolean`

  `stbox {>>, |>>, />>, #>>} stbox → boolean`

```
SELECT tbox 'TBOXFLOAT XT((3,4),[2001-01-03,2001-01-04])' >>
  tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])';
-- true
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' |>> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' />> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
SELECT stbox 'STBOX XT(((3,3),(4,4)),[2001-01-03,2001-01-04])'  #>>
  stbox 'STBOX XT(((1,1),(2,2)),[2001-01-01,2001-01-02])';
-- true
```

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

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

  `stbox {&<, &<|, &</, &<#} stbox → boolean`

```
SELECT tbox 'TBOXFLOAT XT((1,4),[2001-01-01,2001-01-04])' &<
  tbox 'TBOXFLOAT XT((3,4),[2001-01-03,2001-01-04])';
-- true
SELECT stbox 'STBOX Z((1,1,1),(4,4,4))' &<| stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
SELECT stbox 'STBOX Z((1,1,1),(4,4,4))' &</ stbox 'STBOX Z((3,3,3),(4,4,4))';
-- true
SELECT tbox 'TBOXFLOAT XT((1,4),[2001-01-01,2001-01-04])' &<#
  tbox 'TBOXFLOAT XT((3,4),[2001-01-03,2001-01-04])';
-- true
```

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

  `tbox {&>, #&>} tbox → boolean`

  `stbox {&>, |&>, /&>, #&>} stbox} → boolean`

```
SELECT tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-02])' &>
  tbox 'TBOXFLOAT XT((1,4),[2001-01-01,2001-01-04])';
-- true
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' |&> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- false
SELECT stbox 'STBOX Z((3,3,3),(4,4,4))' /&> stbox 'STBOX Z((1,1,1),(2,2,2))';
-- true
SELECT stbox 'STBOX XT(((1,1),(2,2)),[2001-01-01,2001-01-02])' #&>
  stbox 'STBOX XT(((1,1),(4,4)),[2001-01-01,2001-01-04])';
-- true
```

3.10 Comparisons

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.

- Traditional comparisons

  `box {=, <>, <, <=, >=} box → boolean`

  `tbox_cmp(tbox,tbox) → int`

  `stbox_cmp(stbox,stbox) → int`

  Functions `tbox_cmp` and `stbox_cmp` returns -1, 0, or 1 depending on whether the first argument is, respectively, less than, equal to, or greater than the second one.

```
SELECT tbox 'TBOXINT XT([1,1],[2001-01-01,2001-01-04])' =
  tbox 'TBOXINT XT([2,2],[2001-01-03,2001-01-05])';
-- false
SELECT tbox 'TBOXFLOAT XT([1,1],[2001-01-01,2001-01-04])' <>
  tbox 'TBOXFLOAT XT([2,2],[2001-01-03,2001-01-05])';
-- true
SELECT tbox 'TBOXINT XT([1,1],[2001-01-01,2001-01-04])' <
  tbox 'TBOXINT XT([1,2],[2001-01-03,2001-01-05])';
-- true
SELECT tbox 'TBOXFLOAT XT([1,1],[2001-01-03,2001-01-04])' >
  tbox 'TBOXFLOAT XT((1,2),[2001-01-01,2001-01-05])';
-- true
SELECT tbox 'TBOXINT XT([1,1],[2001-01-01,2001-01-04])' <=
  tbox 'TBOXINT XT([2,2],[2001-01-03,2001-01-05])';
-- true
SELECT tbox 'TBOXFLOAT XT([1,1],[2001-01-01,2001-01-04])' >=
  tbox 'TBOXFLOAT XT([2,2],[2001-01-03,2001-01-05])';
-- false
SELECT tbox_cmp(tbox 'TBOXFLOAT XT([1,1],[2001-01-01,2001-01-04])',
  tbox 'TBOXFLOAT XT([2,2],[2001-01-03,2001-01-05])');
-- -1
```

3.11 Aggregations

- Bounding box extent

  `extent(box) → box`

```
WITH boxes(b) AS (
  SELECT tbox 'TBOXFLOAT XT((1,3),[2001-01-01,2001-01-03])' UNION
  SELECT tbox 'TBOXFLOAT XT((5,7),[2001-01-05,2001-01-07])' UNION
  SELECT tbox 'TBOXFLOAT XT((6,8),[2001-01-06,2001-01-08])' )
SELECT extent(b) FROM boxes;
-- TBOXFLOAT XT((1,8),[2001-01-01,2001-01-08])
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))
```

3.12 Indexing

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 4

Temporal Types (Part 1)

4.1 Introduction

MobilityDB provides several temporal types based on PostgreSQL or PostGIS types, namely, `tbool`, `tint`, `tfloat`, `ttext`, `tgeometry`, `tgeography`, `tgeompoint`, and `tgeogpoint`, which are, respectively, based on the base types `bool`, `integer`, `float`, `text`, `geometry`, and `geography`, where `tgeometry` and `tgeography` accept arbitrary geometries/geographies, while `tgeompoint` and `tgeogpoint` only accept 2D or 3D points with Z dimension. In addition, MobilityDB provides other specific spatial types, namely, `cbuffer` (circular buffer), `npoint` (network point), and `pose`, and its corresponding spatiotemporal types, namely, `tcbuffer`, `tnpoint`, `tpose` and `trgeometry` (temporal rigid geometry). In this chapter and the next one we present the functions and operators available for all temporal types, while the specific functions for the spatiotemporal types will be given in subsequenct chapters.

The *interpolation* of a temporal value states how the value evolves between successive instants. The interpolation is *discrete* when the value is unknown between two successive instants. They can represent, for example, checkins/checkouts when using an RFID card reader to enter or exit a building. The interpolation is *step* when the value remains constant between two successive instants. For example, the gear used by a moving car 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 speed of a car may be represented with a temporal float, which indicates that the values are known at the 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 step manner. On the other hand, temporal types based on continuous base types, that is `tfloat`, `tgeompoint`, or `tgeogpoint` may evolve in a step or linear manner. Note that the types `tgeometry` and `tgeography` only support discrete or step interpolation, since it is not possible to linearly interpolate two arbitrary geometries/geographies.

The *subtype* of a temporal value states the temporal extent at which the evolution of values is recorded. Temporal values come in three subtypes, explained next.

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 *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 a discrete, step, or a linear function (see above). An example is as follows:

```
-- Discrete interpolation
SELECT tfloat '{17@2018-01-01 08:00:00, 17.5@2018-01-01 08:05:00, 18@2018-01-01 08:10:00}';
-- Step interpolation
SELECT tfloat 'Interp=Step;(10@2018-01-01 08:00:00, 20@2018-01-01 08:05:00,
  15@2018-01-01 08:10:00]';
-- Linear interpolation
SELECT tfloat '(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 '`)`'). By definition, both bounds must be inclusive when the interpolation is discrete or when the sequence has a single instant (called an *instantaneous sequence*), as the next example

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

Sequence values must be *uniform*, that is, they must be composed of instant values of the same base type. Sequence values with step or linear interpolation are referred to as *continuous sequences*.

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 step 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]}';
```

As shown in the above examples, sequence set values can only be of step or linear interpolation. Furtheremore, all composing sequences of a sequence set value must be of the same base type and the same interpolation.

Continuous sequence values are converted into *normal form* so that equivalent values have identical representations. For this, consecutive instant values are merged when possible. For step 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 asText(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 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 asText(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 tint '{[1@2001-01-01, 1@2001-01-03), [2@2001-01-03, 2@2001-01-05)}';
-- '{[1@2001-01-01, 2@2001-01-03, 2@2001-01-05)}'
SELECT tfloat '{[1@2001-01-01, 2@2001-01-03), [2@2001-01-03, 3@2001-01-05]}';
-- '{[1@2001-01-01, 3@2001-01-05]}'
SELECT tfloat '{[1@2001-01-01, 3@2001-01-05), [3@2001-01-05]}';
-- '{[1@2001-01-01, 3@2001-01-05]}'
SELECT asText(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 asText(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 asText(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`, `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 geometry types (that is, `tgeompoint` `tgeogpoint`, `tgeometry`, or `tgeography`) the type modifier may be used to specify the subtype, the geometry type, and/or the spatial reference identifier (SRID). For example, in the following table definition:

```
CREATE TABLE Flight(FlightNo integer, Route tgeogpoint(Sequence, PointZ, 4326));
CREATE TABLE Storm(StormId integer, Route tgeography(Sequence, Linestring, 4326));
```

the type modifiers for the `tgeogpoint` and `tgeography` types are 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 geographies, 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 `tstzspan` type for the `tbool` and `ttext` types, where only the temporal extent is considered.
- The `tbox` (temporal box) type for the `tint` and `tfloat` types, where the value and the time extents are defined, respectively, by a number span and a time span.
- The `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 time extent by a time span.

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

4.2 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@2001-01-01, 12@2001-04-01, 12@2001-08-01)'),
  (20, 'Human Resources', tint '[4@2001-02-01, 6@2001-06-01, 6@2001-10-01)');
CREATE TABLE Temperature(RoomNo integer, Temp tfloat);
INSERT INTO Temperature VALUES
  (1001, tfloat '{18.5@2001-01-01 08:00:00, 20.0@2001-01-01 08:10:00}'),
  (2001, tfloat '{19.0@2001-01-01 08:00:00, 22.5@2001-01-01 08:10:00}');
```

```
-- Value at a timestamp
SELECT RoomNo, valueAtTimestamp(Temp, '2001-01-01 08:10:00')
FROM temperature;
-- 1001 | 20
-- 2001 | 22.5

-- Restriction to a value
SELECT DeptNo, atValues(NoEmps, 10)
FROM Department;
-- 10 | [10@2001-01-01, 10@2001-04-01)
-- 20 |

-- Restriction to a period
SELECT DeptNo, atTime(NoEmps, tstzspan '[2001-01-01, 2001-04-01]')
FROM Department;
-- 10 | [10@2001-01-01, 12@2001-04-01]
-- 20 | [4@2001-02-01, 4@2001-04-01]

-- Temporal comparison
SELECT DeptNo, NoEmps #<= 10
FROM Department;
-- 10 | [t@2001-01-01, f@2001-04-01, f@2001-08-01)
-- 20 | [t@2001-02-01, t@2001-10-01)

-- Temporal aggregation
SELECT tsum(NoEmps)
FROM Department;
/* {[10@2001-01-01, 14@2001-02-01, 16@2001-04-01,
   18@2001-06-01, 6@2001-08-01, 6@2001-10-01)} */
```

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)@2001-01-01 08:00:00, Point(2 0)@2001-01-01 08:10:00,
Point(2 1)@2001-01-01 08:15:00)}'),
  (20, 1, tgeompoint '{[Point(0 0)@2001-01-01 08:05:00, Point(1 1)@2001-01-01 08:10:00,
  Point(3 3)@2001-01-01 08:20:00)}');

-- Value at a given timestamp
SELECT CarId, ST_AsText(valueAtTimestamp(Trip, timestamptz '2001-01-01 08:10:00'))
FROM Trips;
-- 10 | POINT(2 0)
-- 20 | POINT(1 1)

-- Restriction to a value
SELECT CarId, asText(atValues(Trip, geometry 'Point(2 0)'))
FROM Trips;
-- 10 | {"[POINT(2 0)@2001-01-01 08:10:00+00]"}
-- 20 |

-- Restriction to a period
SELECT CarId, asText(atTime(Trip, tstzspan '[2001-01-01 08:05:00,2001-01-01 08:10:00]'))
FROM Trips;
-- 10 | {[POINT(1 0)@2001-01-01 08:05:00+00, POINT(2 0)@2001-01-01 08:10:00+00]}
-- 20 | {[POINT(0 0)@2001-01-01 08:05:00+00, POINT(1 1)@2001-01-01 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@2001-01-01 08:05:00+00, 1.4142135623731@2001-01-01 08:10:00+00,
   1@2001-01-01 08:15:00+00)} */
```

4.3 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 or a sequence value with discrete interpolation 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 step interpolation, the last two values must be equal if upper bound is exclusive.
- A sequence set value must be composed of at least one sequence value.
- In a sequence set value, the composing sequence values must be non overlapping and ordered.

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 for step interpolation
SELECT tint '[1@2001-01-01 08:00:00, 2@2001-01-01 08:10:00)';
-- Empty sequence set
SELECT tint '{[]}';
-- Duplicate timestamps
SELECT tint '{1@2001-01-01 08:00:00, 2@2001-01-01 08:00:00}';
-- Overlapping periods
SELECT 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)}';
```

4.4 Temporalizing Operations

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 temporal 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 tfloat '[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, 0.7451840337945570@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 or 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 eIntersects(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 aIntersects(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 example, the `eIntersects` 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.

4.5 Notation

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:

- `time` represents any time type, that is, `timestamptz`, `tstzspan`, `tstzset`, or `tstzspanset`,
- `ttype` represents any temporal type,

- `ttypeInst`, `ttypeSeq`, and `ttypeSeqSet` represent any temporal type with, respectively, instant, sequence, and sequence set subtype,
- `tdisc` represents any temporal type with a discrete base type, that is, `tbool`, `tint`, or `ttext`,
- `tcont` represents any temporal type with a continuous base type, that is, `tfloat`, `tgeompoint`, or `tgeogpoint`,
- `ttypeDiscSeq` and `ttypeContSeq` represent any temporal type with sequence subtype and, respectively, discrete and continuous interpolation,
- `base` represents any base type of a temporal type, that is, `boolean`, `integer`, `float`, `text`, `geometry`, or `geography`,
- `values` represents any set of values of a base type of a temporal type, for example, `integer`, `intset`, `intspan`, and `intspanset` for the base type `integer`
- `type[]` represents an array of `type`.
- `<type>` in the name of a function represents the functions obtained by replacing `<type>` by a specific `type`. For example, `tintSeq` or `tfloatSeq` are represented by `ttypeSeq`.

4.6 Input and Output

MobilityDB generalizes Open Geospatial Consortium's Well-Known Text (WKT), Well-Known Binary (WKB), and Moving Features JSON (MF-JSON) input and output format for all temporal types. We start by describing the WKT format.

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. The temporal extent of an instant value is a single timestamp. Examples of instant 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 tgeometry 'Linestring(0 0,1 1)@2017-01-01 08:00:05';
SELECT tgeography 'Polygon((0 0,1 1,2 0,0 0))@2017-01-01 08:00:05';
```

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 '`)`'). A sequence value composed of a single couple `v@t` is called an *instantaneous sequence*. Sequence values have an associated *interpolation function* which may be discrete, linear, or step. By definition, the lower and upper bounds of an instantaneous sequence or of a sequence value with discrete interpolation are inclusive. The temporal extent of a sequence value with discrete interpolation is a timestamp set. Examples of sequence values with discrete interpolation 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 tint '{1@2001-01-01 08:00:00}'; -- Instantaneous sequence
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 tgeometry '{Point(0 0)@2017-01-01 08:00:00,
  Linestring(0 0,0 1)@2017-01-02 08:05:00}';
SELECT tgeography '{Point(0 0)@2017-01-01 08:00:00,
  Polygon((0 0,1 1,2 0,0 0))@2017-01-02 08:05:00}';
```

The temporal extent of a sequence value with linear or step interpolation is a period defined by the first and last instants as well as the lower and upper bounds. Examples of sequence values with linear interpolation are as follows:

```
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]'; -- Instantaneous 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)';
SELECT tgeometry '[Point(0 0)@2017-01-01 08:00:00,
  Linestring(0 0,0 1)@2017-01-02 08:05:00]';
SELECT tgeography '[Point(0 0)@2017-01-01 08:00:00,
  Polygon((0 0,1 1,2 0,0 0))@2017-01-02 08:05:00]';
```

Sequence values whose base type is continuous may specify that the interpolation is step with the prefix `Interp=Step`. If this is not specified, it is supposed that the interpolation is linear by default. Example of sequence values with step interpolation are given next:

```
SELECT tfloat 'Interp=Step;[2.5@2001-01-01 08:00:00, 3@2001-01-01 08:10:00]';
SELECT tgeompoint 'Interp=Step;[Point(0 0)@2017-01-01 08:00:00,
  Point(1 1)@2017-01-01 08:05:00, Point(1 1)@2017-01-01 08:10:00)';
SELECT tgeompoint 'Interp=Step;[Point(0 0)@2017-01-01 08:00:00,
 Point(1 1)@2017-01-01 08:05:00, Point(0 0)@2017-01-01 08:10:00)';
ERROR:  Invalid end value for temporal sequence with step interpolation
SELECT tgeogpoint 'Interp=Step;[Point(0 0)@2017-01-01 08:00:00,
  Point(1 1)@2017-01-01 08:10:00]';
```

The last two instants of a sequence value with discrete interpolation and exclusive upper bound must have the same base value, as shown in the second and third examples above.

A *sequence set value* is a set `{v1,...,vn}` where every `vi` is a sequence value. The interpolation of sequence set values can only be linear or step, not discrete. All sequences composing a sequence set value must have the same interpolation. The temporal extent of a sequence set value is a set of periods. Examples of sequence set values with linear interpolation are as follows:

```
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)}';
SELECT tgeometry
  '{[Point(0 0)@2017-01-01 08:00:00, Linestring(0 0,1 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 tgeography
  '{[Point(0 0)@2017-01-01 08:00:00, Polygon((0 0,1 1,2 0,0 0))@2017-01-01 08:05:00),
    [Point(0 1)@2017-01-01 08:10:00, Linestring(0 0,1 1)@2017-01-01 08:15:00)}';
```

Sequence set values whose base type is continuous may specify that the interpolation is step with the prefix `Interp=Step`. If this is not specified, it is supposed that the interpolation is linear by default. Example of sequence set values with step interpolation are given next:

```
SELECT tfloat 'Interp=Step;{[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 tgeompoint 'Interp=Step;{[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(0 1)@2017-01-01 08:15:00)}';
SELECT tgeogpoint 'Interp=Step;{[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(0 1)@2017-01-01 08:15:00)}';
```

For temporal geometries, 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)@2001-01-01,Point(0 1)@2001-01-02]'
SELECT tgeography 'SRID=7844;[Point(0 0)@2001-01-01,Linestring(1 0,0 1)@2001-01-02]'
```

All component geometries of a temporal geometry 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)@2001-01-01,SRID=5435;Point(0 1)@2001-01-02]'
```

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, as shown below.

```
SELECT tgeompoint '[SRID=5435;Point(0 0)@2001-01-01,SRID=4326;Point(0 1)@2001-01-02]';
-- ERROR: Geometry SRID (4326) does not match temporal type SRID (5435)
SELECT tgeography 'SRID=7844;[SRID=4326;Point(0 0)@2001-01-01,
  SRID=4326;Linestring(1 0,0 1)@2001-01-02]';
-- ERROR: Geometry SRID (4326) does not match temporal type SRID (7844)
```

If not specified, the default SRID for temporal geometric points is 0 (unknown) and for temporal geographies is 4326 (WGS 84). Temporal geometries with step interpolation may also specify the SRID, as shown next.

```
SELECT tgeompoint 'SRID=5435,Interp=Step;[Point(0 0)@2001-01-01,
  Point(0 1)@2001-01-02]';
SELECT tgeogpoint 'Interp=Step;[SRID=4326;Point(0 0)@2001-01-01,
  SRID=4326;Point(0 1)@2001-01-02]';
```

We give below the input and output functions in Well-Known Text (WKT), Well-Known Binary (WKB), and Moving Features JSON (MF-JSON) format for temporal alphanumeric types. The corresponding functions for temporal geometries are detailed in Section 7.3. The default output format of all temporal alphanumeric types is the Well-Known Text format. The function `asText` given next enables to determine the output of temporal float values.

- Return the Well-Known Text (WKT) representation

  `asText(ttype,maxdecdigits=15)` → `text`

  The `maxdecdigits` argument can be used to set the maximum number of decimal places in the output of floating point values (default 15).

  ```
  SELECT asText(tfloat '[10.55@2001-01-01, 25.55@2001-01-02]', 0);
  -- [11@2001-01-01, 26@2001-01-02]
  SELECT asText(tgeometry
    '[Point(1.55 1.55)@2001-01-01,Linestring(1.55 1.55,3.55 3.55)@2001-01-02]', 0);
  -- [Point(1 1)@2001-01-01, Linestring(1 1,3 3)@2001-01-02]
  ```

- Return the Moving Features JSON (MF-JSON) representation

  `asMFJSON(ttype,options=0,flags=0,maxdecdigits=15)` → `bytea`

  The `options` argument can be used to add a bounding box in the MFJSON output:

  – 0: means no option (default value)

  – 1: MFJSON BBOX

  The `flags` argument can be used to customize the JSON output, for example, to produce an easy-to-read (for human readers) JSON output. Refer to the documentation of the `json-c` library for the possible values. Typical values are as follows:

- 0: means no option (default value)
- 1: JSON_C_TO_STRING_SPACED
- 2: JSON_C_TO_STRING_PRETTY

The `maxdecdigits` argument can be used to set the maximum number of decimal places in the output of floating point values (default 15).

```
SELECT asMFJSON(tbool 't@2001-01-01 18:00:00', 1);
/* {"type":"MovingBoolean","period":{"begin":"2001-01-01T18:00:00+01",
  "end":"2001-01-01T18:00:00+01","lowerInc":true,"upperInc":true},
  "values":[true],"datetimes":["2001-01-01T18:00:00+01"],"interpolation":"None"} */
SELECT asMFJSON(tint '{10@2001-01-01 18:00:00, 25@2001-01-01 18:10:00}', 1);
/* {"type":"MovingInteger","bbox":[10,25],"period":{"begin":"2001-01-01T18:00:00+01",
   "end":"2001-01-01T18:10:00+01"},"values":[10,25],"datetimes":["2001-01-01T18:00:00+01",
   "2001-01-01T18:10:00+01"],"lowerInc":true,"upperInc":true,
   "interpolation":"Discrete"} */
SELECT asMFJSON(tfloat '[10.5@2001-01-01 18:00:00+02, 25.5@2001-01-01 18:10:00+02]');
/* {"type":"MovingFloat","values":[10.5,25.5],"datetimes":["2001-01-01T17:00:00+01",
  "2001-01-01T17:10:00+01"],"lowerInc":true,"upperInc":true,"interpolation":"Linear"} */
SELECT asMFJSON(ttext '{[walking@2001-01-01 18:00:00+02,
  driving@2001-01-01 18:10:00+02]}');
/* {"type":"MovingText","sequences":[{"values":["walking","driving"],
  "datetimes":["2001-01-01T17:00:00+01","2001-01-01T17:10:00+01"],
  "lowerInc":true,"upperInc":true}],"interpolation":"Step"} */
SELECT asMFJSON(tgeometry '{[Point(1 1)@2001-01-01 18:00:00+02,
  Linestring(1 1,2 2)@2001-01-01 18:10:00+02]}');
/* {"type":"MovingGeometry","sequences":[{"values":[{"type":"Point",
    "coordinates":[1,1]},{"type":"LineString","coordinates":[[1,1],[2,2]]}],
    "datetimes":["2001-01-01T17:00:00+01","2001-01-01T17:10:00+01"],
    "lower_inc":true,"upper_inc":true}],"interpolation":"Step"} */
```

- Return the Well-Known Binary (WKB) or the Hexadecimal Well-Known Binary (WKB) representation

  `asBinary(ttype,endian text='') → bytea`

  `asHexWKB(ttype,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 the encoding of the machine is used.

```
SELECT asBinary(tbool 'true@2001-01-01');
-- \x011a000101009c57d3c11c0000
SELECT asBinary(tfloat '1.5@2001-01-01');
-- \x012100010000000000000f83f009c57d3c11c0000
SELECT asHexWKB(tint '1@2001-01-01', 'XDR');
-- 00002301000000100001CC1D3579C00
SELECT asHexWKB(ttext 'AAA@2001-01-01');
-- 012900010400000000000000414141000009C57D3C11C0000
```

- Input from the Moving Features JSON (MF-JSON) representation

  `ttypeFromMFJSON(bytea) → ttype`

  There is one function per temporal type, the name of the function has as prefix the name of the type, which is `tbool` or `tint` in the examples below.

```
SELECT tboolFromMFJSON(text
  '{"type":"MovingBoolean","period":{"begin":"2001-01-01T18:00:00+01",
  "end":"2001-01-01T18:00:00+01","lowerInc":true,"upperInc":true},
  "values":[true],"datetimes":["2001-01-01T18:00:00+01"],"interpolation":"None"}');
-- t@2001-01-01 18:00:00
SELECT tintFromMFJSON(text
  '{"type":"MovingInteger","bbox":[10,25],"period":{"begin":"2001-01-01T18:00:00+01",
   "end":"2001-01-01T18:10:00+01"},"values":[10,25],"datetimes":["2001-01-01T18:00:00+01",
```

```
   "2001-01-01T18:10:00+01"],"lowerInc":true,"upperInc":true,
   "interpolation":"Discrete"}');
-- {10@2001-01-01 18:00:00, 25@2001-01-01 18:10:00}
SELECT tfloatFromMFJSON(text
  '{"type":"MovingFloat","values":[10.5,25.5],"datetimes":["2001-01-01T17:00:00+01",
   "2001-01-01T17:10:00+01"],"lowerInc":true,"upperInc":true,
   "interpolation":"Linear"}');
-- [10.5@2001-01-01 18:00:00, 25.5@2001-01-01 18:10:00]'
SELECT ttextFromMFJSON(text
  '{"type":"MovingText","sequences":[{"values":["walking","driving"],
  "datetimes":["2001-01-01T17:00:00+01","2001-01-01T17:10:00+01"],
  "lowerInc":true,"upperInc":true}],"interpolation":"Step"}');
-- {["walking"@2001-01-01 18:00:00, "driving"@2001-01-01 18:10:00]}');
SELECT asText(tgeometryFromMFJSON(text
  '{"type":"MovingGeometry","sequences":[{"values":[{"type":"Point",
  "coordinates":[1,1]},{"type":"LineString","coordinates":[[1,1],[2,2]]}],
  "datetimes":["2001-01-01T17:00:00+01","2001-01-01T17:10:00+01"],
  "lower_inc":true,"upper_inc":true}],"interpolation":"Step"}'));
-- {[POINT(1 1)@2001-01-01 17:00:00, LINESTRING(1 1,2 2)@2001-01-01 17:10:00]}
```

- Input from the Well-Known Binary (WKB) or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `ttypeFromBinary(bytea)` → `ttype`

  `ttypeFromHexWKB(text)` → `ttype`

  There is one function per temporal type, the name of the function has as prefix the name of the type, which is `tbool` or `tint` in the examples below.

```
SELECT tboolFromBinary('\x011a000101009c57d3c11c0000');
-- t@2001-01-01
SELECT tintFromBinary('\x0000230100000000100001cc1d3579c00');
-- 1@2001-01-01
SELECT tfloatFromBinary('\x012100010000000000000f83f009c57d3c11c0000');
-- 1.5@2001-01-01
SELECT ttextFromBinary('\x012900010400000000000000041414100009c57d3c11c0000');
-- "AAA"@2001-01-01
SELECT tboolFromHexWKB('011A000101009C57D3C11C0000');
-- t@2001-01-01
SELECT tintFromHexWKB('0000230100000000100001CC1D3579C00');
-- 1@2001-01-01
SELECT tfloatFromHexWKB('012100010000000000000F83F009C57D3C11C0000');
-- 1.5@2001-01-01
SELECT ttextFromHexWKB('012900010400000000000000041414100009C57D3C11C0000');
-- "AAA"@2001-01-01
```

4.7 Constructors

We give next the constructor functions for the various subtypes. Using the constructor function is frequently more convenient than writing a literal constant.

- Constructors for temporal types having a constant value

  These contructors have two arguments, a base type and a time value, where the latter is a `timestamptz`, a `tstzset`, a `tstzspan`, or a `tstzspanset` value for constructing, respectively, an instant, a sequence with discrete interpolation, a sequence with linear or step interpolation, or a sequence set value. The functions for sequence or sequence set values with continuous base type have an optional third argument stating whether the resulting temporal value has linear or step interpolation. Linear interpolation is assumed by default if the argument is not specified.

  `ttype(base,timestamptz)` → `ttypeInst`

`ttype(base,tstzset) → ttypeDiscSeq`

`ttype(base,tstzspan,interp='linear') → ttypeContSeq`

`ttype(base,tstzspanset,interp='linear') → ttypeSeqSet`

```
SELECT tbool(true, timestamptz '2001-01-01');
SELECT tint(1, timestamptz '2001-01-01');
SELECT tfloat(1.5, tstzset '{2001-01-01, 2001-01-02}');
SELECT ttext('AAA', tstzset '{2001-01-01, 2001-01-02}');
SELECT tfloat(1.5, tstzspan '[2001-01-01, 2001-01-02]');
SELECT tfloat(1.5, tstzspan '[2001-01-01, 2001-01-02]', 'step');
SELECT tgeompoint('Point(0 0)', tstzspan '[2001-01-01, 2001-01-02]');
SELECT tgeography('SRID=7844;Point(0 0)', tstzspanset '{[2001-01-01, 2001-01-02],
  [2001-01-03, 2001-01-04]}', 'step');
```

- Constructors for temporal types of sequence subtype

  These contructors have a first mandatory argument, which is an array of values of the corresponding instant values and additional optional arguments. The second argument states the interpolation. If the argument is not given, it is by default step for discrete base types such as integer, and linear for continuous base types such as float. An error is raised when linear interpolation is stated for temporal values with discrete base types. For continuous sequences, the third and fourth arguments are Boolean values stating, respectively, whether the left and right bounds are inclusive or exclusive. These arguments are assumed to be true by default if they are not specified.

  `ttypeSeq(ttypeInst[],interp={'step','linear'},leftInc bool=true,`
  `rightInc bool=true) → ttypeSeq`

```
SELECT tboolSeq(ARRAY[tbool 'true@2001-01-01 08:00:00','false@2001-01-01 08:05:00'],
  'discrete');
SELECT tintSeq(ARRAY[tint '1@2001-01-01 08:00:00', '2@2001-01-01 08:05:00']);
SELECT tintSeq(ARRAY[tint '1@2001-01-01 08:00:00', '2@2001-01-01 08:05:00'], 'linear');
-- ERROR:  The temporal type cannot have linear interpolation
SELECT tfloatSeq(ARRAY[tfloat '1.0@2001-01-01 08:00:00', '2.0@2001-01-01 08:05:00'],
  'step', false, true);
SELECT ttextSeq(ARRAY[ttext 'AAA@2001-01-01 08:00:00', 'BBB@2001-01-01 08:05:00']);
SELECT tgeompointSeq(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'], 'discrete');
SELECT tgeographySeq(ARRAY[tgeography 'Point(1 1)@2001-01-01 08:00:00',
  'Point(2 2)@2001-01-01 08:05:00']);
```

- Constructors for temporal types of sequence set subtype

  `ttypeSeqSet(ttypeContSeq[]) → ttypeSeqSet`

  `ttypeSeqSetGaps(ttypeInst[],maxt=NULL,maxdist=NULL,interp='linear') → ttypeSeqSet`

  A first set of contructors have a single argument, which is an array of values of the corresponding *sequence* values. 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.

```
SELECT tboolSeqSet(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 tintSeqSet(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 tfloatSeqSet(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 tfloatSeqSet(ARRAY[tfloat 'Interp=Step;[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=Step;[3.0@2001-01-01 08:15:00, 3.0@2001-01-01 08:20:00)']);
SELECT ttextSeqSet(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 tgeompointSeqSet(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 tgeographySeqSet(ARRAY[tgeography
  '[Point(0 0)@2001-01-01 08:00:00, Point(0 0)@2001-01-01 08:05:00)',
  '[Point(1 1)@2001-01-01 08:10:00, Point(1 1)@2001-01-01 08:15:00)']);
SELECT tfloatSeqSet(ARRAY[tfloat 'Interp=Step;[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:  The temporal values must have the same interpolation
```

Another set of constructors for sequence set values have as first argument an array of the corresponding *instant* values, and two optional arguments stating a maximum time interval and a maximum distance such that a gap is introduced between sequences of the result whenever two consecutive input instants have a time gap or a distance greater than these values. For temporal geometries, the distance is specified in units of the coordinate system. When none of the gap arguments are given, the resulting value will be a singleton sequence set. In addition, when the base type is continuous, an additional last argument states whether the interpolation is step or linear. It this argument is not specified it is assumed to be linear by default.

The parameters of the function depend on the temporal type. For example, the interpolation parameter is not allowed for temporal types with discrete subtype such as `tint`. Similarly, the parameter `maxdist` is not allowed for scalar types such as `ttext` that do not have a standard distance function.

```
SELECT tintSeqSetGaps(ARRAY[tint '1@2001-01-01', '3@2001-01-02', '4@2001-01-03',
  '5@2001-01-05']);
-- {[1@2001-01-01, 3@2001-01-02, 4@2001-01-03, 5@2001-01-05]}
SELECT tintSeqSetGaps(ARRAY[tint '1@2001-01-01', '3@2001-01-02', '4@2001-01-03',
  '5@2001-01-05'], '1 day', 1);
-- {[1@2001-01-01], [3@2001-01-02, 4@2001-01-03], [5@2001-01-05]}
SELECT ttextSeqSetGaps(ARRAY[ttext 'AA@2001-01-01', 'BB@2001-01-02', 'AA@2001-01-03',
  'CC@2001-01-05'], '1 day');
-- {["AA"@2001-01-01, "BB"@2001-01-02, "AA"@2001-01-03], ["CC"@2001-01-05]}
SELECT asText(tgeometrySeqSetGaps(ARRAY[tgeometry 'Point(1 1)@2001-01-01',
  'Linestring(2 2,3 3)@2001-01-02', 'Point(4 2)@2001-01-03',
  'Polygon((5 0,6 1,7 0,5 0))@2001-01-05'], '1 day', 1));
/* {[POINT(1 1)@2001-01-01], [LINESTRING(2 2,3 3)@2001-01-02],
   [POINT(4 2)@2001-01-03], [POLYGON((5 0,6 1,7 0,5 0))@2001-01-05]} */
SELECT asText(tgeompointSeqSetGaps(ARRAY[tgeompoint 'Point(1 1)@2001-01-01',
  'Point(2 2)@2001-01-02', 'Point(3 2)@2001-01-03', 'Point(3 2)@2001-01-05'],
  '1 day', 1, 'step'));
/* Interp=Step;{[POINT(1 1)@2001-01-01], [POINT(2 2)@2001-01-02, POINT(3 2)@2001-01-03],
   [POINT(3 2)@2001-01-05]} */
```

4.8 Conversions

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

- Convert a temporal value to a timestamptz span

  `ttype::tstzspan`

```
SELECT tint '[1@2001-01-01, 2@2001-01-03]'::tstzspan;
-- [2001-01-01, 2001-01-03]
SELECT ttext '(A@2001-01-01, B@2001-01-03, C@2001-01-05]'::tstzspan;
-- (2001-01-01, 2001-01-05]
SELECT tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03]'::tstzspan;
-- [2001-01-01, 2001-01-03]
```

4.9 Accessors

- Return the memory size in bytes

  `memSize(ttype) → integer`

```
SELECT memSize(tint '{1@2001-01-01, 2@2001-01-02, 3@2001-01-03}');
-- 176
```

- Return the temporal type

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

```
SELECT tempSubtype(tint '[1@2001-01-01, 2@2001-01-02, 3@2001-01-03]');
-- Sequence
```

- Return the interpolation

  `interp(ttype) → {'None','Discrete','Step','Linear'}`

```
SELECT interp(tbool 'true@2001-01-01');
-- None
SELECT interp(tfloat '{1@2001-01-01, 2@2001-01-02, 3@2001-01-03}');
-- Discrete
SELECT interp(tint '[1@2001-01-01, 2@2001-01-02, 3@2001-01-03]');
-- Step
SELECT interp(tfloat '[1@2001-01-01, 2@2001-01-02, 3@2001-01-03]');
-- Linear
SELECT interp(tfloat 'Interp=Step;[1@2001-01-01, 2@2001-01-02, 3@2001-01-03]');
-- Step
SELECT interp(tgeompoint 'Interp=Step;[Point(1 1)@2001-01-01,
  Point(2 2)@2001-01-02, Point(3 3)@2001-01-03]');
-- Step
SELECT interp(tgeometry '[Point(1 1)@2001-01-01,
  Linestring(1 1,2 2)@2001-01-02, Polygon((3 3,4 4,5 3,3 3))@2001-01-03]');
-- Step
```

- Return the value or the timestamp of an instant

  `getValue(ttypeInst) → base`

  `getTimestamp(ttypeInst) → timestamptz`

```
SELECT getValue(tint '1@2001-01-01');
-- 1
SELECT getTimestamp(tfloat '1@2001-01-01');
-- 2001-01-01
```

- Return the values or the time on which a temporal value is defined

  `getValues(ttype) → baseset`

  `getTime(ttype) → tstzspanset`

```
SELECT getValues(tbool '[false@2001-01-01, true@2001-01-02, false@2001-01-03]');
-- {f,t}
SELECT getValues(tint '[1@2001-01-01, 3@2001-01-02, 1@2001-01-03]');
-- {[1, 2), [3, 4)}
SELECT getValues(tint '{[1@2001-01-01, 2@2001-01-02, 1@2001-01-03],
  [4@2001-01-04, 4@2001-01-05]}');
-- {[1, 3), [4, 5)}
SELECT getValues(tfloat '{1@2001-01-01, 2@2001-01-02, 1@2001-01-03}');
-- {[1, 1], [2, 2]}
SELECT getValues(tfloat 'Interp=Step;{[1@2001-01-01, 2@2001-01-02, 1@2001-01-03],
  [3@2001-01-04, 3@2001-01-05]}');
-- {[1, 1], [2, 2], [3, 3]}
SELECT getValues(tfloat '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]');
-- {[1, 2]}
SELECT getValues(tfloat '{[1@2001-01-01, 2@2001-01-02, 1@2001-01-03],
  [3@2001-01-04, 3@2001-01-05]}');
-- {[1, 2], [3, 3]}
```

```
SELECT getTime(ttext 'walking@2001-01-01');
-- {[2001-01-01, 2001-01-01]}
SELECT getTime(tfloat '{1@2001-01-01, 2@2001-01-02, 1@2001-01-03}');
-- {[2001-01-01, 2001-01-01], [2001-01-02, 2001-01-02], [2001-01-03, 2001-01-03]}
SELECT getTime(tint '[1@2001-01-01, 1@2001-01-15)');
-- {[2001-01-01, 2001-01-15)}
SELECT getTime(tfloat '{[1@2001-01-01, 1@2001-01-10), [12@2001-01-12, 12@2001-01-15]}');
-- {[2001-01-01, 2001-01-10), [2001-01-12, 2001-01-15]}
```

- Return the start, end, or n-th value

  `startValue(ttype) → base`

  `endValue(ttype) → base`

  `valueN(ttype,int) → base`

  The functions do not take into account whether the bounds are inclusive or not.

```
SELECT startValue(tfloat '(1@2001-01-01, 2@2001-01-03)');
-- 1
SELECT endValue(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}');
-- 5
SELECT valueN(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}', 3);
-- 3
```

- Return the value at a timestamp

  `valueAtTimestamp(ttype,timestamptz) → base`

```
SELECT valueAtTimestamp(tfloat '[1@2001-01-01, 4@2001-01-04)', '2001-01-02');
-- 2
```

- Return the time interval

  `duration(ttype,boundspan=false) → interval`

  An additional parameter can be set to true to compute the duration of the bounding time span, thus ignoring the potential time gaps

```
SELECT duration(tfloat '{1@2001-01-01, 2@2001-01-03, 2@2001-01-05}');
-- 00:00:00
SELECT duration(tfloat '{1@2001-01-01, 2@2001-01-03, 2@2001-01-05}', true);
-- 4 days
SELECT duration(tfloat '[1@2001-01-01, 2@2001-01-03, 2@2001-01-05)');
-- 4 days
SELECT duration(tfloat '{[1@2001-01-01, 2@2001-01-03), [2@2001-01-04, 2@2001-01-05)}');
-- 3 days
SELECT duration(tfloat '{[1@2001-01-01, 2@2001-01-03), [2@2001-01-04, 2@2001-01-05)}',
  true);
-- 4 days
```

- Is the start/end instant inclusive?

  `lowerInc(ttype) → bool`

  `upperInc(ttype) → bool`

```
SELECT lowerInc(tint '[1@2001-01-01, 2@2001-01-02)');
-- true
SELECT upperInc(tfloat '{[1@2001-01-01, 2@2001-01-02), (2@2001-01-02, 3@2001-01-03)}');
-- false
```

- Return the (number of) different instants

  `numInstants(ttype) → integer`

  `instants(ttype) → ttypeInst[]`

```
SELECT numInstants(tfloat '{[1@2001-01-01, 2@2001-01-02), (2@2001-01-02, 3@2001-01-03)}');
-- 3
SELECT instants(tfloat '{[1@2001-01-01, 2@2001-01-02), (2@2001-01-02, 3@2001-01-03)}');
-- {"1@2001-01-01","2@2001-01-02","3@2001-01-03"}
```

- Return the start, end, or n-th instant

  `startInstant(ttype) → ttypeInst`

  `endInstant(ttype) → ttypeInst`

  `instantN(ttype,integer) → ttypeInst`

  The functions do not take into account whether the bounds are inclusive or not.

```
SELECT startInstant(tfloat '{[1@2001-01-01, 2@2001-01-02),
  (2@2001-01-02, 3@2001-01-03)}');
-- 1@2001-01-01
SELECT endInstant(tfloat '{[1@2001-01-01, 2@2001-01-02), (2@2001-01-02, 3@2001-01-03)}');
-- 3@2001-01-03
SELECT instantN(tfloat '{[1@2001-01-01, 2@2001-01-02), (2@2001-01-02, 3@2001-01-03)}', 3);
-- 3@2001-01-03
```

- Return the (number of) different timestamps

  `numTimestamps(ttype) → integer`

  `timestamps(ttype) → timestamptz[]`

```
SELECT numTimestamps(tfloat '{[1@2001-01-01, 2@2001-01-03),
  [3@2001-01-03, 5@2001-01-05)}');
-- 3
SELECT timestamps(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}');
-- {"2001-01-01", "2001-01-03", "2001-01-05"}
```

- Return the start, end, or n-th timestamp

  `startTimestamp(ttype) → timestamptz`

  `endTimestamp(ttype) → timestamptz`

  `timestampN(ttype,integer) → timestamptz`

  The functions do not take into account whether the bounds are inclusive or not.

```
SELECT startTimestamp(tfloat '[1@2001-01-01, 2@2001-01-03)');
-- 2001-01-01
SELECT endTimestamp(tfloat '{[1@2001-01-01, 2@2001-01-03),
  [3@2001-01-03, 5@2001-01-05)}');
-- 2001-01-05
SELECT timestampN(tfloat '{[1@2001-01-01, 2@2001-01-03),
  [3@2001-01-03, 5@2001-01-05)}', 3);
-- 2001-01-05
```

- Return the (number of) sequences

  `numSequences({ttypeContSeq,ttypeSeqSet}) → integer`

  `sequences({ttypeContSeq,ttypeSeqSet}) → ttypeContSeq[]`

```
SELECT numSequences(tfloat '{[1@2001-01-01, 2@2001-01-03),
  [3@2001-01-03, 5@2001-01-05)}');
-- 2
SELECT sequences(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}');
-- {"[1@2001-01-01, 2@2001-01-03)", "[3@2001-01-03, 5@2001-01-05)"}
```

- Return the start, end, or n-th sequence

  `startSequence({ttypeContSeq,ttypeSeqSet}) → ttypeContSeq`

  `endSequence({ttypeContSeq,ttypeSeqSet}) → ttypeContSeq`

  `sequenceN({ttypeContSeq,ttypeSeqSet},integer) → ttypeContSeq`

```
SELECT startSequence(tfloat '{[1@2001-01-01, 2@2001-01-03),
  [3@2001-01-03, 5@2001-01-05)}');
-- [1@2001-01-01, 2@2001-01-03)
SELECT endSequence(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}');
-- [3@2001-01-03, 5@2001-01-05)
SELECT sequenceN(tfloat '{[1@2001-01-01, 2@2001-01-03),
  [3@2001-01-03, 5@2001-01-05)}', 2);
-- [3@2001-01-03, 5@2001-01-05)
```

- Return the segments

  `segments({ttypeContSeq,ttypeSeqSet}) → ttypeContSeq[]`

```
SELECT segments(tint '{[1@2001-01-01, 3@2001-01-02, 2@2001-01-03],
  (3@2001-01-03, 5@2001-01-05]}');
/* {"[1@2001-01-01, 1@2001-01-02)","[3@2001-01-02, 3@2001-01-03)","[2@2001-01-03]",
   "(3@2001-01-03, 3@2001-01-05)","[5@2001-01-05]"} */
SELECT segments(tfloat '{[1@2001-01-01, 3@2001-01-02, 2@2001-01-03],
  (3@2001-01-03, 5@2001-01-05]}');
/* {"[1@2001-01-01, 3@2001-01-02)","[3@2001-01-02, 2@2001-01-03]",
   "(3@2001-01-03, 5@2001-01-05]"} */
```

4.10 Transformations

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

- Transform a temporal value to another subtype

  `ttypeInst(ttype) → ttypeInst`

  `ttypeSeq(ttype) → ttypeSeq`

  `ttypeSeqSet(ttype) → ttypeSeqSet`

```
SELECT tboolInst(tbool '{[true@2001-01-01]}');
-- t@2001-01-01
SELECT tboolInst(tbool '{[true@2001-01-01, true@2001-01-02]}');
-- ERROR: Cannot transform input value to a temporal instant
SELECT tintSeq(tint '1@2001-01-01');
-- [1@2001-01-01]
SELECT tfloatSeqSet(tfloat '2.5@2001-01-01');
-- {[2.5@2001-01-01]}
SELECT tfloatSeqSet(tfloat '{2.5@2001-01-01, 1.5@2001-01-02, 3.5@2001-01-03}');
-- {[2.5@2001-01-01],[1.5@2001-01-02],[3.5@2001-01-03]}
```

- Transform a temporal value to another interpolation

  `setInterp(ttype, interp) → ttype`

```
SELECT setInterp(tbool 'true@2001-01-01','discrete');
-- {t@2001-01-01}
SELECT setInterp(tfloat '{[1@2001-01-01], [2@2001-01-02], [1@2001-01-03]}', 'discrete');
-- {1@2001-01-01, 2@2001-01-02, 1@2001-01-03}
SELECT setInterp(tfloat 'Interp=Step;[1@2001-01-01, 2@2001-01-02,
  1@2001-01-03, 2@2001-01-04]', 'linear');
```

```
/* {[1@2001-01-01, 1@2001-01-02), [2@2001-01-02, 2@2001-01-03),
   [1@2001-01-03, 1@2001-01-04), [2@2001-01-04]} */
SELECT asText(setInterp(tgeompoint 'Interp=Step;{[Point(1 1)@2001-01-01,
  Point(2 2)@2001-01-02], [Point(3 3)@2001-01-05, Point(4 4)@2001-01-06]}', 'linear'));
/* {[POINT(1 1)@2001-01-01, POINT(1 1)@2001-01-02), [POINT(2 2)@2001-01-02],
   [POINT(3 3)@2001-01-05, POINT(3 3)@2001-01-06), [POINT(4 4)@2001-01-06]} */
SELECT setInterp(tgeometry '[Point(1 1)@2001-01-01,
  Linestring(1 1,2 2)@2001-01-02]', 'linear');
-- ERROR:  The temporal type cannot have linear interpolation
```

- Return a temporal boolean stating whether each segment of a continuous temporal sequence (set) has, respectively, at least or at most a given duration

  `segmentMinDuration(temp,interval,bool strict=true)` → `tbool`

  `segmentMaxDuration(temp,interval,bool strict=true)` → `tbool`

  If `strict` is not specified, the value `true` is assumed by default, and therefore the function checks whether the segment duration is strictly less than or greater than the interval. If the argument is set to `false`, the function checks whether the segment duration is less/greater than or *equal to* the interval.

```
SELECT segmentMinDuration(tfloat '[1@2001-01-01, 0@2001-01-03, -1@2001-01-04,
  -1@2001-01-06]', '1 day');
-- {[t@2001-01-01, f@2001-01-03, t@2001-01-04, t@2001-01-06)}
SELECT segmentMinDuration(tfloat '[1@2001-01-01, 0@2001-01-03, -1@2001-01-04,
  -1@2001-01-06]', '1 day', false);
-- {[t@2001-01-01, t@2001-01-06)}
SELECT segmentMaxDuration(tfloat '[1@2001-01-01, 0@2001-01-03, -1@2001-01-04,
  -1@2001-01-06]', '1 day');
-- {[f@2001-01-01 00:00:00+01, f@2001-01-06 00:00:00+01)}
SELECT segmentMaxDuration(tfloat '[1@2001-01-01, 0@2001-01-03, -1@2001-01-04,
  -1@2001-01-06]', '1 day', false);
-- {[f@2001-01-01, t@2001-01-03, f@2001-01-04, f@2001-01-06)}
```

- Shift and/or scale the time span of a temporal value by one or two intervals

  The given intervals must be strictly greater than zero. If the time span of the temporal value is zero (for example, for a temporal instant), the result of a scale operation is the temporal value.

  `shiftTime(ttype,interval)` → `ttype`

  `scaleTime(ttype,interval)` → `ttype`

  `shiftScaleTime(ttype,interval,interval)` → `ttype`

```
SELECT shiftTime(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 shiftTime(tfloat '[1@2001-01-01, 2@2001-01-03]', '1 day');
-- [1@2001-01-02, 2@2001-01-04]
SELECT asText(shiftTime(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]} */
SELECT scaleTime(tint '1@2001-01-01', '1 day');
-- 1@2001-01-01
SELECT scaleTime(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 scaleTime(tfloat '[1@2001-01-01, 2@2001-01-03]', '1 day');
-- [1@2001-01-01, 2@2001-01-02]
SELECT asText(scaleTime(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 scaleTime(tint '1@2001-01-01', '-1 day');
```

```
-- ERROR:  The interval must be positive: -1 days
SELECT shiftScaleTime(tint '1@2001-01-01', '1 day', '1 day');
-- 1@2001-01-02
SELECT shiftScaleTime(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 shiftScaleTime(tfloat '[1@2001-01-01, 2@2001-01-03]', '1 day', '1 day');
-- [1@2001-01-02, 2@2001-01-03]
SELECT asText(shiftScaleTime(tgeometry '{[Point(1 1)@2001-01-01,
  Linestring(1 1,2 2)@2001-01-02, Point(1 1)@2001-01-03], [Point(2 2)@2001-01-04,
  Polygon((1 1,2 2,3 1,1 1))@2001-01-05]}', '1 day', '1 day'));
/* {[POINT(1 1)@2001-01-02 00:00:00, LINESTRING(1 1,2 2)@2001-01-02 06:00:00,
    POINT(1 1)@2001-01-02 12:00:00], [POINT(2 2)@2001-01-02 18:00:00,
    POLYGON((1 1,2 2,3 1,1 1))@2001-01-03 00:00:00]} */
```

- Transform a nonlinear temporal value into a set of rows, each one is a pair composed of a base value and a period set during which the temporal value has the base value `{}`

  `unnest(ttype) → {(value,time)}`

```
SELECT (un).value, (un).time
FROM (SELECT unnest(tfloat '{1@2001-01-01, 2@2001-01-02, 1@2001-01-03}') AS un) t;
-- 1 | {[2001-01-01, 2001-01-01], [2001-01-03, 2001-01-03]}
-- 2 | {[2001-01-02, 2001-01-02]}
SELECT (un).value, (un).time
FROM (SELECT unnest(tint '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]') AS un) t;
-- 1 | {[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-03]}
-- 2 | {[2001-01-02, 2001-01-03)}
SELECT (un).value, (un).time
FROM (SELECT unnest(tfloat '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]') AS un) t;
-- ERROR:  The temporal value cannot have linear interpolation
SELECT ST_AsText((un).value), (un).time
FROM (SELECT unnest(tgeompoint 'Interp=Step;[Point(1 1)@2001-01-01,
  Point(2 2)@2001-01-02, Point(1 1)@2001-01-03]') AS un) t;
--  POINT(1 1) | {[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-03]}
--  POINT(2 2) | {[2001-01-02, 2001-01-03)}
SELECT ST_AsText((un).value), (un).time
FROM (SELECT unnest(tgeometry '[Point(1 1)@2001-01-01,
  Linestring(1 1,2 2)@2001-01-02, Point(1 1)@2001-01-03]') AS un) t;
-- POINT(1 1)          | {[2001-01-01, 2001-01-02), [2001-01-03, 2001-01-03]}
-- LINESTRING(1 1,2 2) | {[2001-01-02, 2001-01-03)}
```

Chapter 5

Temporal Types (Part 2)

5.1 Modifications

We explain next the semantics of the modification operations (that is, `insert`, `update`, and `delete`) for temporal types. These operations have similar semantics as the corresponding operations for application-time temporal tables introduced in the SQL:2011 standard. The main difference is that SQL uses *tuple timestamping* (where timestamps are attached to tuples), while temporal values in MobilityDB use *attribute timestamping* (where timestamps are attached to attribute values). Please refer to this article for a detailed discussion about tuple versus attribute timestamping in temporal databases.

The `insert` operation adds to a temporal value the instants of another one without modifying the existing instants, as illustrated in Figure 5.1.

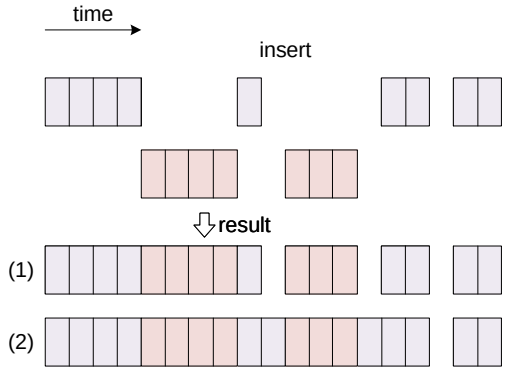


Figure 5.1: Insert operation for temporal values.

As shown in the figure, the temporal values may only intersect at their boundary, and in that case, they must have the same base value at their common timestamps, otherwise an error is raised. The result of the operation is the union of the instants for both temporal values, as shown in the first result of the figure. This is equivalent to a `merge` operation explained below. Alternatively, as shown in the second result of the figure, the inserted fragments that are disjoint with the original value are connected to the last instant before and the first instant after the fragment. A Boolean parameter `connect` is used to choose between the two results, and the parameter is set to true by default. Notice that this only applies to continuous temporal values.

The `update` operation replaces the instants in the first temporal value with those of the second one as illustrated in Figure 5.2.

As in the case of an `insert` operation, an additional Boolean parameter determines whether the replaced disconnected fragments are connected in the resulting value, as shown in the two possible results in the figure. When the two temporal values are either disjoint or only overlap at their boundary, this corresponds to an `insert` operation as explained above. In this case, the `update` operation behaves as an `upsert` operation in SQL.

The `deleteTime` operation removes the instants of a temporal value that intersect a time value. This operation can be used in two different situations, illustrated in Figure 5.2.

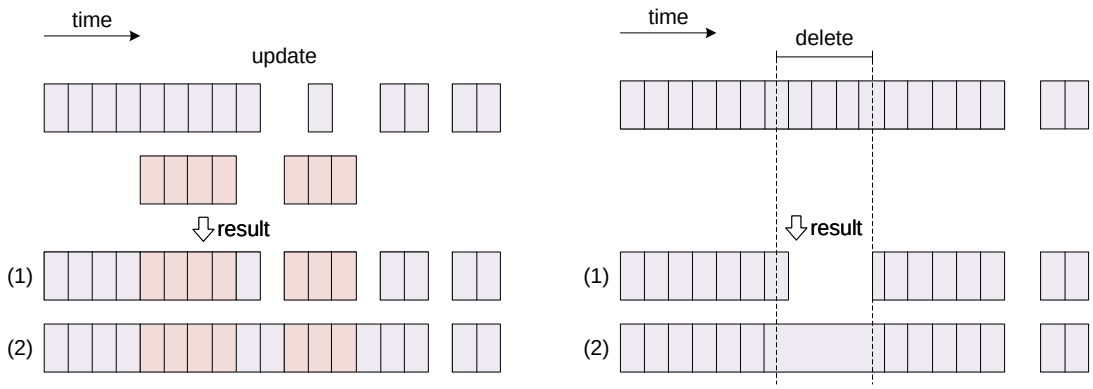


Figure 5.2: Update and delete operation for temporal values.

1. In the first case, shown as the top result in the figure, the meaning of operation is to introduce time gaps after removing the instants of the temporal value intersecting the time value. This is equivalent to the restriction operations (Section 5.2), which restrict a temporal value to the complement of the time value.
2. The second case, shown as the bottom result in the figure, is used for removing erroneous values (e.g., detected as outliers) without introducing a time gap, or for removing time gaps. In this case, the instants of the temporal value are deleted and the last instant before and the first instant after a removed fragment are connected. This behaviour is specified by setting an additional Boolean parameter of the operation. Notice that this only applies to continuous temporal values.

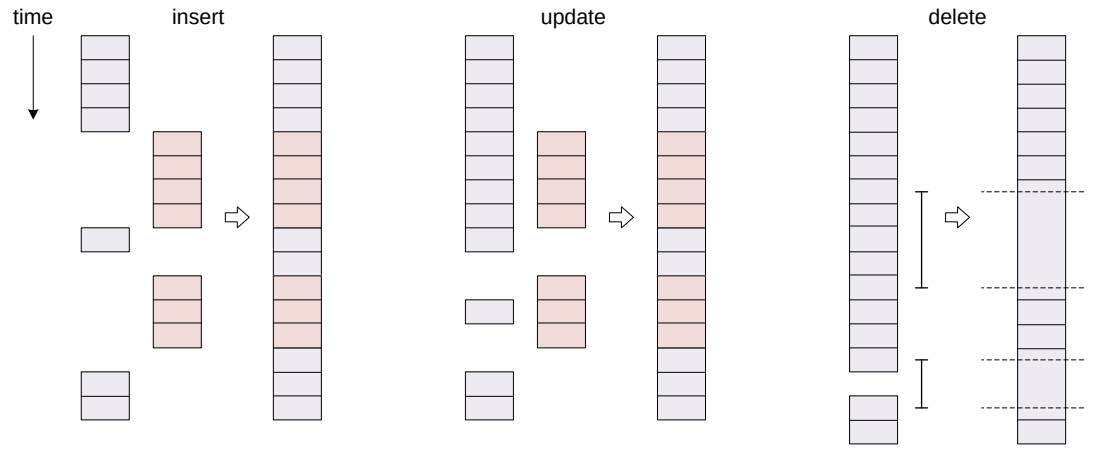


Figure 5.3: Modification operations for temporal tables in SQL

Figure 5.3 shows the equivalent modification operations for temporal tables in the SQL standard. Intuitively, these figures are obtained by rotating 90 degrees clockwise the corresponding figures for temporal values (Figure 5.1 and Figure 5.2). This follows from the fact that in SQL consecutive tuples ordered by time are typically connected through the `LEAD` and `LAG` window functions.

- Insert a temporal value into another one

`insert(ttype,ttype,connect=true) → ttype`

```
SELECT insert(tint '{1@2001-01-01, 3@2001-01-03, 5@2001-01-05}',
  tint '{3@2001-01-03, 7@2001-01-07}');
-- {1@2001-01-01, 3@2001-01-03, 5@2001-01-05, 7@2001-01-07}
SELECT insert(tint '{1@2001-01-01, 3@2001-01-03, 5@2001-01-05}',
  tint '{5@2001-01-03, 7@2001-01-07}');
```

```
-- ERROR: The temporal values have different value at their overlapping instant 2001-01-03
SELECT insert(tfloat '[1@2001-01-01, 2@2001-01-02]',
  tfloat '[1@2001-01-03, 1@2001-01-05]');
-- [1@2001-01-01, 2@2001-01-02, 1@2001-01-03, 1@2001-01-05]
SELECT insert(tfloat '[1@2001-01-01, 2@2001-01-02]',
  tfloat '[1@2001-01-03, 1@2001-01-05]', false);
-- {[1@2001-01-01, 2@2001-01-02], [1@2001-01-03, 1@2001-01-05]}
SELECT asText(insert(tgeompoint '{[Point(1 1 1)@2001-01-01, Point(2 2 2)@2001-01-02],
  [Point(3 3 3)@2001-01-04],[Point(1 1 1)@2001-01-05]}',
  tgeompoint 'Point(1 1 1)@2001-01-03'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02, POINT Z (1 1 1)@2001-01-03,
 POINT Z (3 3 3)@2001-01-04], [POINT Z (1 1 1)@2001-01-05]} */
```

- Update a temporal value with another one

  `update(ttype,ttype,connect=true) → ttype`

```
SELECT update(tint '{1@2001-01-01, 3@2001-01-03, 5@2001-01-05}',
  tint '{5@2001-01-03, 7@2001-01-07}');
-- {1@2001-01-01, 5@2001-01-03, 5@2001-01-05, 7@2001-01-07}
SELECT update(tfloat '[1@2001-01-01, 1@2001-01-05]',
  tfloat '[1@2001-01-02, 3@2001-01-03, 1@2001-01-04]');
-- {[1@2001-01-01, 1@2001-01-02, 3@2001-01-03, 1@2001-01-04, 1@2001-01-05]}
SELECT asText(update(tgeompoint '{[Point(1 1 1)@2001-01-01, Point(3 3 3)@2001-01-03,
  Point(1 1 1)@2001-01-05], [Point(1 1 1)@2001-01-07]}',
  tgeompoint '[Point(2 2 2)@2001-01-02, Point(2 2 2)@2001-01-04]'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02, POINT Z (2 2 2)@2001-01-04,
  POINT Z (1 1 1)@2001-01-05], [POINT Z (1 1 1)@2001-01-07]} */
SELECT asText(update(
  tgeometry '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03, Point(1 1)@2001-01-05]',
  tgeometry '[Linestring(2 2,3 3)@2001-01-02, Linestring(3 3,2 2)@2001-01-04]'));
/* [POINT(1 1)@2001-01-01, LINESTRING(2 2,3 3)@2001-01-02,
    LINESTRING(3 3,2 2)@2001-01-04], (POINT(3 3)@2001-01-04, POINT(1 1)@2001-01-05] */
```

- Delete the instants of a temporal value that intersect a time value

  `deleteTime(ttype,time,connect=true) → ttype`

```
SELECT deleteTime(tint '[1@2001-01-01, 1@2001-01-03]', timestamptz '2001-01-02', false);
-- {[1@2001-01-01, 1@2001-01-02), (1@2001-01-02, 1@2001-01-03]}
SELECT deleteTime(tint '[1@2001-01-01, 1@2001-01-03]', timestamptz '2001-01-04');
-- [1@2001-01-01, 1@2001-01-03]
SELECT deleteTime(tfloat '[1@2001-01-01, 4@2001-01-02, 2@2001-01-04, 5@2001-01-05]',
  tstzspan '[2001-01-02, 2001-01-04]');
-- [1@2001-01-01, 5@2001-01-05]
SELECT asText(deleteTime(tgeompoint '{[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02], [Point(3 3 3)@2001-01-04, Point(3 3 3)@2001-01-05]}',
  tstzspan '[2001-01-02, 2001-01-04]'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02, POINT Z (3 3 3)@2001-01-04,
  POINT Z (3 3 3)@2001-01-05]} */
```

- Append a temporal instant to a temporal value

  `appendInstant(ttype,ttypeInst,interp=NULL) → ttype`

  `appendInstant(ttypeInst,interp=NULL,maxdist=NULL,maxt=NULL) → ttypeSeq`

  The first version of the function returns the result of appending the second argument to the first one. If either input is NULL, then NULL is returned.

  The second version of the function above is an *aggregate* function that returns the result of successively appending a set of rows of temporal values. This means that it operates in the same way the `SUM()` and `AVG()` functions do and like most aggregates, it also ignores NULL values. Two optional arguments state a maximum distance and a maximum time interval such that a gap is introduced whenever two consecutive instants have a distance or a time gap greater than these values. For temporal points

the distance is specified in units of the coordinate system. If one of the arguments are not given, it is not taken into account for determining the gaps.

```
SELECT appendInstant(tint '1@2001-01-01', tint '1@2001-01-02');
-- [1@2001-01-01, 1@2001-01-02]
SELECT appendInstant(tint '1@2001-01-01', tint '1@2001-01-02', 'discrete');
-- {1@2001-01-01, 1@2001-01-02}
SELECT appendInstant(tint '[1@2001-01-01]', tint '1@2001-01-02');
-- [1@2001-01-01, 1@2001-01-02]
SELECT asText(appendInstant(tgeompoint '{[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02], [Point(3 3 3)@2001-01-04, Point(3 3 3)@2001-01-05]}',
  tgeompoint 'Point(1 1 1)@2001-01-06'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02],
   [POINT Z (3 3 3)@2001-01-04, POINT Z (3 3 3)@2001-01-05,
   POINT Z (1 1 1)@2001-01-06]} */
SELECT asText(appendInstant(tgeometry '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02],
  [Linestring(2 2,3 3)@2001-01-04, Point(3 3)@2001-01-05]}',
  tgeometry 'Linestring(1 1,2 2)@2001-01-06'));
/* {[POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02], [LINESTRING(2 2,3 3)@2001-01-04,
     POINT(3 3)@2001-01-05, LINESTRING(1 1,2 2)@2001-01-06]} */
```

```
WITH temp(inst) AS (
  SELECT tfloat '1@2001-01-01' UNION SELECT tfloat '2@2001-01-02' UNION
  SELECT tfloat '3@2001-01-03' UNION SELECT tfloat '4@2001-01-04' UNION
  SELECT tfloat '5@2001-01-05' )
SELECT appendInstant(inst ORDER BY inst) FROM temp;
-- [1@2001-01-01, 5@2001-01-05]
WITH temp(inst) AS (
  SELECT tfloat '1@2001-01-01' UNION SELECT tfloat '2@2001-01-02' UNION
  SELECT tfloat '3@2001-01-03' UNION SELECT tfloat '4@2001-01-04' UNION
  SELECT tfloat '5@2001-01-05' )
SELECT appendInstant(inst, 'discrete' ORDER BY inst) FROM temp;
-- {1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 4@2001-01-04, 5@2001-01-05}
WITH temp(inst) AS (
  SELECT tgeogpoint 'Point(1 1)@2001-01-01' UNION
  SELECT tgeogpoint 'Point(2 2)@2001-01-02' UNION
  SELECT tgeogpoint 'Point(3 3)@2001-01-03' UNION
  SELECT tgeogpoint 'Point(4 4)@2001-01-04' UNION
  SELECT tgeogpoint 'Point(5 5)@2001-01-05' )
SELECT asText(appendInstant(inst ORDER BY inst)) FROM temp;
/* [POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02, POINT(3 3)@2001-01-03,
   POINT(4 4)@2001-01-04, POINT(5 5)@2001-01-05] */
```

Notice that in the first query above with `tfloat`, the intermediate observations were removed by the normalization process since they were redundant due to linear interpolation. This is not the case for the second query with discrete interpolation or the third query with `tgeogpoint`, where geodetic coordinates are used.

```
WITH temp(inst) AS (
  SELECT tfloat '1@2001-01-01' UNION SELECT tfloat '2@2001-01-02' UNION
  SELECT tfloat '4@2001-01-04' UNION SELECT tfloat '5@2001-01-05' UNION
  SELECT tfloat '7@2001-01-07' )
SELECT appendInstant(inst, NULL, 0.0, '1 day' ORDER BY inst) FROM temp;
-- {[1@2001-01-01, 2@2001-01-02], [4@2001-01-04, 5@2001-01-05], [7@2001-01-07]}
WITH temp(inst) AS (
  SELECT tgeompoint 'Point(1 1)@2001-01-01' UNION
  SELECT tgeompoint 'Point(2 2)@2001-01-02' UNION
  SELECT tgeompoint 'Point(4 4)@2001-01-04' UNION
  SELECT tgeompoint 'Point(5 5)@2001-01-05' UNION
  SELECT tgeompoint 'Point(7 7)@2001-01-07' )
SELECT asText(appendInstant(inst, NULL, sqrt(2), '1 day' ORDER BY inst)) FROM temp;
/* {[POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02],
   [POINT(4 4)@2001-01-04, POINT(5 5)@2001-01-05], [POINT(7 7)@2001-01-07]} */
```

- Append a temporal sequence to a temporal value

  `appendSequence(ttype,ttypeSeq) → {ttypeSeq,ttypeSeqSet}`

  `appendSequence(ttypeSeq) → {ttypeSeq,ttypeSeqSet}`

  The first version of the function returns the result of appending the second argument to the first one. If either input is NULL, then NULL is returned.

  The second version of the function above is an *aggregate* function that returns the result of successively appending a set of rows of temporal values. This means that it operates in the same way the `SUM()` and `AVG()` functions do and like most aggregates, it also ignores NULL values.

```
SELECT appendSequence(tint '1@2001-01-01', tint '{2@2001-01-02, 3@2001-01-03}');
-- {1@2001-01-01, 2@2001-01-02, 3@2001-01-03}
SELECT appendSequence(tint '[1@2001-01-01, 2@2001-01-02]',
  tint '[2@2001-01-02, 3@2001-01-03]');
-- [1@2001-01-01, 2@2001-01-02, 3@2001-01-03]
SELECT asText(appendSequence(tgeompoint '{[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02], [Point(3 3 3)@2001-01-04, Point(3 3 3)@2001-01-05]}',
  tgeompoint '[Point(3 3 3)@2001-01-05, Point(1 1 1)@2001-01-06]'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02],
   [POINT Z (3 3 3)@2001-01-04, POINT Z (3 3 3)@2001-01-05,
   POINT Z (1 1 1)@2001-01-06]} */
SELECT asText(appendSequence(tgeometry '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02],
  [Linestring(3 3,2 2)@2001-01-04, Point(3 3)@2001-01-05]}',
  tgeometry '[Point(3 3)@2001-01-05, Linestring(3 3,1 1)@2001-01-06]'));
/* {[POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02], [LINESTRING(3 3,2 2)@2001-01-04,
     POINT(3 3)@2001-01-05, LINESTRING(3 3,1 1)@2001-01-06]} */
```

- Merge the temporal values

  `merge(ttype,ttype) → ttype`

  `merge(ttype[]) → ttype`

  `merge(ttype) → ttype`

  The temporal values may only intersect at their boundary. In that case, for all temporal types excepted `tgeometry` and `tgeography`, their base values at the common timestamps must be the same, otherwise an error is raised. For the `tgeometry` and `tgeography` types, if the value at their common timestamps is different, the resulting value after the `merge` will be the spatial union of the values. The reason for a different behaviour for these types is that they are the only temporal types that are not *scalar* types, that is, their value at a given timestamp is not a single element of the domain of the base type, but rather a subset of their domain.

  The third version of the function above is an *aggregate* function that returns the result of successively appending a set of rows of temporal values. This means that it operates in the same way the `SUM()` and `AVG()` functions do and like most aggregates, it also ignores NULL values.

```
SELECT merge(tint '1@2001-01-01', tint '1@2001-01-02');
-- {1@2001-01-01, 1@2001-01-02}
SELECT merge(tint '[1@2001-01-01, 2@2001-01-02]', tint '[2@2001-01-02, 1@2001-01-03]');
-- [1@2001-01-01, 2@2001-01-02, 1@2001-01-03]
SELECT merge(tint '[1@2001-01-01, 2@2001-01-02]', tint '[3@2001-01-03, 1@2001-01-04]');
-- {[1@2001-01-01, 2@2001-01-02], [3@2001-01-03, 1@2001-01-04]}
SELECT merge(tint '[1@2001-01-01, 2@2001-01-02]', tint '[1@2001-01-02, 2@2001-01-03]');
-- ERROR:  The temporal values have different value at their common timestamp 2001-01-02
SELECT asText(merge(tgeompoint '{[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02], [Point(3 3 3)@2001-01-04, Point(3 3 3)@2001-01-05]}',
  tgeompoint '{[Point(3 3 3)@2001-01-05, Point(1 1 1)@2001-01-06]}'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02],
   [POINT Z (3 3 3)@2001-01-04, POINT Z (3 3 3)@2001-01-05,
   POINT Z (1 1 1)@2001-01-06]} */
SELECT asText(merge(tgeometry 'Linestring(1 1,5 1)@2001-01-01',
  tgeometry '{Linestring(5 1,10 1)@2001-01-01, Point(3 3)@2001-01-02}'));
-- {LINESTRING(1 1,5 1,10 1)@2001-01-01, POINT(3 3)@2001-01-02}
```

```
SELECT merge(ARRAY[tint '1@2001-01-01', '1@2001-01-02']);
-- {1@2001-01-01, 1@2001-01-02}
SELECT merge(ARRAY[tint '{1@2001-01-01, 2@2001-01-02}', '{2@2001-01-02, 3@2001-01-03}']);
-- {1@2001-01-01, 2@2001-01-02, 3@2001-01-03}
SELECT merge(ARRAY[tint '{1@2001-01-01, 2@2001-01-02}', '{3@2001-01-03, 4@2001-01-04}']);
-- {1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 4@2001-01-04}
SELECT merge(ARRAY[tint '[1@2001-01-01, 2@2001-01-02]', '[2@2001-01-02, 1@2001-01-03]']);
-- [1@2001-01-01, 2@2001-01-02, 1@2001-01-03]
SELECT merge(ARRAY[tint '[1@2001-01-01, 2@2001-01-02]', '[3@2001-01-03, 4@2001-01-04]']);
-- {[1@2001-01-01, 2@2001-01-02], [3@2001-01-03, 4@2001-01-04]}
SELECT asText(merge(ARRAY[tgeompoint '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02],
  [Point(3 3)@2001-01-03, Point(4 4)@2001-01-04]}', '{[Point(4 4)@2001-01-04,
  Point(3 3)@2001-01-05], [Point(6 6)@2001-01-06, Point(7 7)@2001-01-07]}']));
/* {[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02], [Point(3 3)@2001-01-03,
   Point(4 4)@2001-01-04, Point(3 3)@2001-01-05],
   [Point(6 6)@2001-01-06, Point(7 7)@2001-01-07]} */
SELECT asText(merge(ARRAY[tgeompoint '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02]}',
  '{[Point(2 2)@2001-01-02, Point(1 1)@2001-01-03]}']));
-- [Point(1 1)@2001-01-01, Point(2 2)@2001-01-02, Point(1 1)@2001-01-03]
SELECT asText(merge(ARRAY[tgeometry '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02]}',
  '{[Linestring(2 2,1 1)@2001-01-02, Point(1 1)@2001-01-03]}']));
-- {[POINT(1 1)@2001-01-01, LINESTRING(2 2,1 1)@2001-01-02, POINT(1 1)@2001-01-03]}
```

```
WITH temp(inst) AS (
SELECT tfloat '1@2001-01-01' UNION SELECT tfloat '2@2001-01-02' UNION
SELECT tfloat '3@2001-01-03' UNION SELECT tfloat '4@2001-01-04' UNION
SELECT tfloat '5@2001-01-05' )
SELECT merge(inst) FROM temp;
-- {1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 4@2001-01-04, 5@2001-01-05}
```

5.2 Restrictions

There are three complementary sets of restriction functions. The first set functions restricts the temporal value with respect to a value or a time extent. Examples are `atValues` or `atTime`. The second set functions restricts the temporal value with respect to the *complement* of a value or a time extent. Examples are `minusValues` or `minusTime`. Finally, the third set functions restricts the temporal value before or after a timestamptz. Examples are `beforeTimestamp` or `afterTimestamp`.

- Restrict to (the complement of) a set of values

  `atValues(ttype,values)` → `ttype`

  `minusValues(ttype,values)` → `ttype`

```
SELECT atValues(tint '[1@2001-01-01, 1@2001-01-15)', 1);
-- [1@2001-01-01, 1@2001-01-15)
SELECT atValues(tfloat '[1@2001-01-01, 4@2001-01-4)', floatset '{1, 3, 5}');
-- {[1@2001-01-01], [3@2001-01-03]}
SELECT atValues(tfloat '[1@2001-01-01, 4@2001-01-4)', floatspan '[1,3]');
-- [1@2001-01-01, 3@2001-01-03]
SELECT atValues(tfloat '[1@2001-01-01, 5@2001-01-05)',
  floatspanset '{[1,2], [3,4]}');
-- {[1@2001-01-01, 2@2001-01-02],[3@2001-01-03, 4@2001-01-04]}
SELECT asText(atValues(tgeompoint '[Point(0 0 0)@2001-01-01, Point(2 2 2)@2001-01-03)',
  geometry 'Point(1 1 1)'));
-- {[POINT Z (1 1 1)@2001-01-02]}
SELECT asText(atValues(tgeompoint '[Point(0 0)@2001-01-01, Point(2 2)@2001-01-03)',
  geomset '{"Point(0 0)", "Point(1 1)"}'));
-- {[POINT(0 0)@2001-01-01], [POINT(1 1)@2001-01-02]}
SELECT asText(atValues(tgeometry '[Point(0 0)@2001-01-01, Linestring(0 0,2 2)@2001-01-02,
```

```
  Linestring(0 0,2 2)@2001-01-03]', geometry 'Linestring(0 0,2 2)'));
-- {[LINESTRING(0 0,2 2)@2001-01-02, LINESTRING(0 0,2 2)@2001-01-03]}
```

```
SELECT minusValues(tint '[1@2001-01-01, 2@2001-01-02, 2@2001-01-03)', 1);
-- {[2@2001-01-02, 2@2001-01-03)}
SELECT minusValues(tfloat '[1@2001-01-01, 4@2001-01-4)', floatset '{2, 3}');
/* {[1@2001-01-01, 2@2001-01-02), (2@2001-01-02, 3@2001-01-03),
    (3@2001-01-03, 4@2001-01-04)} */
SELECT minusValues(tfloat '[1@2001-01-01, 4@2001-01-4)', floatspan '[2,3]');
-- {[1@2001-01-01, 2@2001-01-02), (3@2001-01-03, 4@2001-01-04)}
SELECT minusValues(tfloat '[1@2001-01-01, 5@2001-01-05)',
  floatspanset '{[1,2], [3,4]}');
-- {(2@2001-01-02, 3@2001-01-03), (4@2001-01-04, 5@2001-01-05)}
SELECT asText(minusValues(tgeompoint '[Point(0 0 0)@2001-01-01, Point(2 2 2)@2001-01-03)',
  geometry 'Point(1 1 1)'));
/* {[POINT Z (0 0 0)@2001-01-01, POINT Z (1 1 1)@2001-01-02),
    (POINT Z (1 1 1)@2001-01-02, POINT Z (2 2 2)@2001-01-03)} */
SELECT asText(minusValues(tgeompoint '[Point(0 0 0)@2001-01-01, Point(3 3 3)@2001-01-04)',
  geomset '{"Point(1 1 1)', 'Point(2 2 2)"}'));
/* {[POINT Z (0 0 0)@2001-01-01, POINT Z (1 1 1)@2001-01-02),
    (POINT Z (1 1 1)@2001-01-02, POINT Z (2 2 2)@2001-01-03),
   (POINT Z (2 2 2)@2001-01-03, POINT Z (3 3 3)@2001-01-04)} */
SELECT asText(minusValues(tgeometry '[Point(0 0)@2001-01-01, Linestring(0 0,2 2)@2001 ↩
    -01-02,
  Linestring(0 0,2 2)@2001-01-03]', geometry 'Linestring(0 0,2 2)'));
--  {[POINT(0 0)@2001-01-01, POINT(0 0)@2001-01-02)}
```

- Restrict to (the complement of) a time value

  `atTime(ttype,times) → ttype`

  `minusTime(ttype,times) → ttype`

```
SELECT atTime(tfloat '[1@2001-01-01, 5@2001-01-05)', timestamptz '2001-01-02');
-- 2@2001-01-02
SELECT atTime(tint '[1@2001-01-01, 1@2001-01-15)',
  tstzset '{2001-01-01, 2001-01-03}');
-- {1@2001-01-01, 1@2001-01-03}
SELECT atTime(tfloat '{[1@2001-01-01, 3@2001-01-03), [3@2001-01-04, 1@2001-01-06)}',
  tstzspan '[2001-01-02,2001-01-05)');
-- {[2@2001-01-02, 3@2001-01-03), [3@2001-01-04, 2@2001-01-05)}
SELECT atTime(tint '[1@2001-01-01, 1@2001-01-15)',
  tstzspanset '{[2001-01-01, 2001-01-03), [2001-01-04, 2001-01-05)}');
-- {[1@2001-01-01, 1@2001-01-03),[1@2001-01-04, 1@2001-01-05)}
```

```
SELECT minusTime(tfloat '[1@2001-01-01, 5@2001-01-05)', timestamptz '2001-01-02');
-- {[1@2001-01-01, 2@2001-01-02), (2@2001-01-02, 5@2001-01-05)}
SELECT minusTime(tint '[1@2001-01-01, 1@2001-01-15)',
  tstzset '{2001-01-02, 2001-01-03}');
/* {[1@2001-01-01, 1@2001-01-02), (1@2001-01-02, 1@2001-01-03),
    (1@2001-01-03, 1@2001-01-15)} */
SELECT minusTime(tfloat '{[1@2001-01-01, 3@2001-01-03), [3@2001-01-04, 1@2001-01-06)}',
  tstzspan '[2001-01-02,2001-01-05)');
-- {[1@2001-01-01, 2@2001-01-02), [2@2001-01-05, 1@2001-01-06)}
SELECT minusTime(tint '[1@2001-01-01, 1@2001-01-15)',
  tstzspanset '{[2001-01-02, 2001-01-03), [2001-01-04, 2001-01-05)}');
/* {[1@2001-01-01, 1@2001-01-02), [1@2001-01-03, 1@2001-01-04),
    [1@2001-01-05, 1@2001-01-15)} */
```

- Keep the instants of a temporal value that are before/after or equal a timestamp

  `beforeTimestamp(ttype,timestamptz,strict bool=TRUE) → ttype`

  `afterTimestamp(ttype,timestamptz,strict bool=TRUE) → ttype`

```
SELECT beforeTimestamp(tint '[1@2001-01-01, 1@2001-01-03]', timestamptz '2001-01-02');
-- [1@2001-01-01, 1@2001-01-02)
SELECT beforeTimestamp(tint '[1@2001-01-01, 1@2001-01-03]', timestamptz '2001-01-01');
-- NULL
SELECT beforeTimestamp(tint '[1@2001-01-01, 1@2001-01-03]', timestamptz '2001-01-01',
  false);
-- [1@2001-01-01]
SELECT afterTimestamp(tfloat '[1@2001-01-01, 3@2001-01-03]', timestamptz '2001-01-02]',
  false);
-- [2@2001-01-02, 3@2001-01-03]
SELECT asText(afterTimestamp(tgeompoint '{[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02], [Point(3 3 3)@2001-01-04, Point(3 3 3)@2001-01-05]}',
  timestamptz '2001-01-03'));
/* {[POINT Z (3 3 3)@2001-01-04, POINT Z (3 3 3)@2001-01-05]} */
```

5.3 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 4, the bounding box associated to a temporal type depends on the base type: It is the `tstzspan` 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 3.3, many PostgreSQL, PostGIS, or MobilityDB types can be converted to the `tbox` and `stbox` types. For example, numeric and span types can be converted to type `tbox`, types `geometry` and `geography` can be converted to type `stbox`, and time types and temporal types can be converted 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 3.9 for the bounding box operators.

5.4 Comparisons

5.4.1 Traditional Comparisons

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 2.10), then the bounding boxes (see Section 3.10) 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 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 tgeography '[Point(1 1)@2001-01-01, Linestring(1 1,2 2)@2001-01-02]' =
  tgeography '{[Point(1 1)@2001-01-01, Linestring(1 1,2 2)@2001-01-02]}';
-- true
```

- Traditional comparisons

  `ttype {=, <>, <, >, <=, >=} ttype → boolean`

  `temporal_cmp(ttype,ttype) → int`

  Function `temporal_cmp` returns -1, 0, or 1 depending on whether the first argument is, respectively, less than, equal to, or greater than the second one.

```
SELECT tint '[1@2001-01-01, 1@2001-01-04)' = tint '[2@2001-01-03, 2@2001-01-05)';
-- false
SELECT tint '[1@2001-01-01, 1@2001-01-04)' <> tint '[2@2001-01-03, 2@2001-01-05)';
-- true
SELECT tint '[1@2001-01-01, 1@2001-01-04)' < tint '[2@2001-01-03, 2@2001-01-05)';
-- true
SELECT ttfloat '[1@2001-01-01, 1@2001-01-04)' > ttfloat '[2@2001-01-03, 2@2001-01-05)';
-- false
SELECT tint '[1@2001-01-01, 1@2001-01-04)' <= tint '[2@2001-01-03, 2@2001-01-05)';
-- true
SELECT tint '[1@2001-01-01, 1@2001-01-04)' >= tint '[2@2001-01-03, 2@2001-01-05)';
-- false
SELECT temporal_cmp(tfloat '[1@2001-01-01, 1@2001-01-04)', '[2@2001-01-03, 2@2001-01-05)') ↩
    ;
-- -1
```

5.4.2 Ever and Always Comparisons

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 or two temporal values 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 and always equality and non-equality are available for all temporal types, while ever and 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 `%<=`.

- Ever and always comparisons

  `{base,ttype} {?=, ?<>, ?<, ?>, ?<=, ?>=} {base,ttype} → boolean`

  `{base,ttype} {%=, %<>, %<, %>, %<=, %>=} {base,ttype} → boolean`

  The operators do not take into account whether the bounds are inclusive or not.

```
SELECT tint '[1@2001-01-01, 3@2001-01-04]' ?= 2;
-- false
SELECT tgeometry '[Point(0 0)@2001-01-01, Linestring(0 0,1 1)@2001-01-04]' ?=
  geometry 'Linestring(1 1,0 0)';
-- true
SELECT tfloat '[1@2001-01-01, 1@2001-01-04)' %= 1;
```

```
-- true
SELECT tgeometry '[Linestring(0 0,1 1)@2001-01-01, Linestring(1 1,0 0)@2001-01-04)' %=
  geometry 'Linestring(0 0,1 1)';
-- true
```

```
SELECT tfloat '[1@2001-01-01, 3@2001-01-04)' ?<> 2;
-- true
SELECT tgeompoint '[Point(1 1)@2001-01-01, Point(1 1)@2001-01-04)' ?<>
  geometry 'Point(1 1)';
-- false
SELECT tfloat '[1@2001-01-01, 3@2001-01-04)' %<> 2;
-- false
SELECT tgeogpoint '[Point(1 1)@2001-01-01, Point(1 1)@2001-01-04)' %<>
  geography 'Point(2 2)';
-- true
```

```
SELECT tint '[1@2001-01-01, 4@2001-01-04]' ?< 2;
-- true
SELECT tfloat '[1@2001-01-01, 4@2001-01-04)' %< 2;
-- false
```

```
SELECT tint '[1@2001-01-03, 1@2001-01-05]' ?> 0;
-- true
SELECT tfloat '[1@2001-01-03, 1@2001-01-05)' %> 1;
-- false
```

```
SELECT tint '[1@2001-01-01, 1@2001-01-05]' ?<= 2;
-- true
SELECT tfloat '[1@2001-01-01, 1@2001-01-05)' %<= 4;
-- true
```

```
SELECT ttext '{[AAA@2001-01-01, AAA@2001-01-03), [BBB@2001-01-04, BBB@2001-01-05)}'
   ?> 'AAA'::text;
-- true
SELECT ttext '{[AAA@2001-01-01, AAA@2001-01-03), [BBB@2001-01-04, BBB@2001-01-05)}'
   %> 'AAA'::text;
-- false
```

5.4.3 Temporal Comparisons

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 comparisons

  `{base,ttype} {#=, #<>, #<, #>, #<=, #>=} {base,ttype} → boolean`

```
SELECT tfloat '[1@2001-01-01, 2@2001-01-04)' #= 3;
-- {[f@2001-01-01, f@2001-01-04)}
SELECT tfloat '[1@2001-01-01, 4@2001-01-04)' #= tfloat '[4@2001-01-02, 1@2001-01-05)';
-- {[f@2001-01-02, t@2001-01-03], (f@2001-01-03, f@2001-01-04)}
SELECT tgeompoint '[Point(0 0)@2001-01-01, Point(2 2)@2001-01-03)' #=
  geometry 'Point(1 1)';
-- {[f@2001-01-01, t@2001-01-02], (f@2001-01-02, f@2001-01-03)}
SELECT tgeometry '[Point(0 0)@2001-01-01, Linestring(1 1,2 2)@2001-01-03]' #=
  geometry 'Linestring(2 2,1 1)';
-- {[f@2001-01-01, t@2001-01-03]}
```

```
SELECT tfloat '[1@2001-01-01, 4@2001-01-04)' #<> 2;
-- {[t@2001-01-01, f@2001-01-02], (t@2001-01-02, 2001-01-04)}
SELECT tfloat '[1@2001-01-01, 4@2001-01-04)' #<> tfloat '[2@2001-01-02, 2@2001-01-05)';
-- {[f@2001-01-02], (t@2001-01-02, t@2001-01-04)}
SELECT tgeometry '[Point(0 0)@2001-01-01, Linestring(1 1,2 2)@2001-01-03]' #<>
  geometry 'Linestring(2 2,1 1)';
-- {[t@2001-01-01, f@2001-01-03]}
```

```
SELECT tfloat '[1@2001-01-01, 4@2001-01-04)' #< 2;
-- {[t@2001-01-01, f@2001-01-02, f@2001-01-04)}
SELECT 1 #> tint '[1@2001-01-03, 1@2001-01-05)';
-- [f@2001-01-03, f@2001-01-05)
SELECT tfloat '[1@2001-01-01, 1@2001-01-05)' #<= tfloat '{2@2001-01-03, 3@2001-01-04}';
-- {t@2001-01-03, t@2001-01-04}
SELECT 'AAA'::text #> ttext '{[AAA@2001-01-01, AAA@2001-01-03),
  [BBB@2001-01-04, BBB@2001-01-05)}';
-- {[f@2001-01-01, f@2001-01-03), [t@2001-01-04, t@2001-01-05)}
```

5.5 Miscellaneous

- Version of the MobilityDB extension

  `mobilitydb_version() → text`

```
SELECT mobilitydb_version();
-- MobilityDB 1.3.0
```

- Versions of the MobilityDB extension and its dependencies

  `mobilitydb_full_version() → text`

```
SELECT mobilitydb_full_version();
/* MobilityDB 1.3.0, PostgreSQL 17.2, PostGIS 3.5.2, GEOS 3.12.0-CAPI-1.18.0, PROJ 9.2.0,
   JSON-C 0.13.1, GSL 2.5 */
```

Chapter 6

Temporal Alphanumeric Types

6.1 Notation

We presented in Section 4.5 the notation used for defining the signature of the functions and operators for temporal alphanumeric types. We extend next this notations for temporal alphanumeric types.

- `torder` represents any temporal type whose base type has a total order defined, that is, `tint`, `tfloat`, or `ttext`,
- `talpha` represents any temporal alphanumeric type, such as, `tint` or `ttext`,
- `tnumber` represents any temporal number type, that is, `tint` or `tfloat`,
- `number` represents any number base type, that is, `integer` or `float`,
- `numspan` represents any number span type, that is, either `intspan` or `floatspan`,

6.2 Conversions

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

- Convert a temporal number to a span or to a temporal bounding box

  `tnumber::{span,tbox}`

  `valueSpan(tnumber)` → `numspan`

  `timeSpan(tnumber)` → `tstzspan`

  `tbox(tnumber)` → `tbox`

```
SELECT tint '[1@2001-01-01, 2@2001-01-03]'::intspan;
-- [1, 3)
SELECT tfloat '(1@2001-01-01, 3@2001-01-03, 2@2001-01-05]'::floatspan;
-- (1, 3]
SELECT valueSpan(tfloat '(1@2001-01-01, 3@2001-01-03, 2@2001-01-05]');
-- (1, 3]
SELECT timeSpan(tfloat 'Interp=Step;(1@2001-01-01, 3@2001-01-03, 2@2001-01-05]');
-- (2001-01-01, 2001-01-05]
```

```
SELECT tint '[1@2001-01-01, 2@2001-01-03]'::tbox;
-- TBOXINT XT((1,2),[2001-01-01,2001-01-03])
SELECT tfloat '(1@2001-01-01, 3@2001-01-03, 2@2001-01-05]'::tbox;
-- TBOXFLOAT XT((1,3),[2001-01-01,2001-01-05])
```

- Convert between a temporal Boolean and a temporal integer

  `tbool::tint`

  `tint(tbool) → tint`

```
SELECT tbool '[true@2001-01-01, false@2001-01-03]'::tint;
-- [1@2001-01-01, 0@2001-01-03]
```

- Convert between a temporal integer and a temporal float

  `tint::tfloat`

  `tfloat::tint`

  `tfloat(tint) → tfloat`

  `tint(tfloat) → tint`

```
SELECT tint '[1@2001-01-01, 2@2001-01-03]'::tfloat;
-- Interp=Step;[1@2001-01-01, 2@2001-01-03]
SELECT tint '[1@2001-01-01, 2@2001-01-03, 3@2001-01-05]'::tfloat;
-- Interp=Step;[1@2001-01-01, 2@2001-01-03, 3@2001-01-05]
SELECT tfloat 'Interp=Step;[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
```

6.3 Accessors

- Return the values of the temporal number or geometry as a set

  `valueSet(tnumber, tgeo) → {numset,geoset}`

```
SELECT valueSet(tint '[1@2001-01-01, 2@2001-01-03]');
-- {1, 2}
SELECT valueSet(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 4@2001-01-05)}');
-- {1, 2, 3, 4}
SELECT asText(valueSet(tgeompoint '{[Point(0 0)@2001-01-01, Point(0 1)@2001-01-02),
  [Point(0 1)@2001-01-03, Point(1 1)@2001-01-04)}'));
-- {"POINT(0 0)", "POINT(1 1)", "POINT(0 1)"}
SELECT asText(valueSet(tgeography
  '{[Point(0 0)@2001-01-01, Linestring(0 0,1 1)@2001-01-02],
    [Point(1 1)@2001-01-03, Linestring(0 0,1 1)@2001-01-04]}'));
-- {"POINT(0 0)", "LINESTRING(0 0,1 1)", "POINT(1 1)"}
```

- Return the minimum, maximum, or average value

  `minValue(torder) → base`

  `maxValue(torder) → base`

  `avgValue(tnumber) → base`

  The functions do not take into account whether the bounds are inclusive or not.

```
SELECT minValue(tfloat '{1@2001-01-01, 2@2001-01-03, 3@2001-01-05}');
-- 1
SELECT maxValue(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}');
-- 5
SELECT avgValue(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}');
-- 2.75
```

```
SELECT minValue(ttext '{[A@2001-01-01, B@2001-01-02], [C@2001-01-03, E@2001-01-05]}');
-- A
SELECT maxValue(ttext '{[A@2001-01-01, B@2001-01-02], [C@2001-01-03, E@2001-01-05]}');
-- E
```

- Return the instant with the minimum or maximum value

  `minInstant(torder)` → `base`

  `maxInstant(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@2001-01-01, 2@2001-01-03, 3@2001-01-05}');
-- 1@2001-01-01
SELECT maxInstant(tfloat '{[1@2001-01-01, 2@2001-01-03), [3@2001-01-03, 5@2001-01-05)}');
-- 5@2001-01-05
```

6.4 Transformations

- Shift and/or the value span of a temporal number by one or two numbers

  `shiftValue(tnumber,base)` → `tnumber`

  `scaleValue(tnumber,width)` → `tnumber`

  `shiftScaleValue(tnumber,base,base)` → `tnumber`

  For scaling, if the value span of the temporal value is a single value (for example, for a temporal instant), the result is the temporal value. Furthermore, the given width must be strictly greater than zero.

```
SELECT shiftValue(tint '{1@2001-01-01, 2@2001-01-03, 1@2001-01-05}', 1);
-- {2@2001-01-02, 3@2001-01-04, 2@2001-01-06}
SELECT shiftValue(tfloat '{[1@2001-01-01,2@2001-01-02],[3@2001-01-03,4@2001-01-04]}', -1);
--  {[0@2001-01-01, 1@2001-01-02], [2@2001-01-03, 3@2001-01-04]}
SELECT scaleValue(tint '1@2001-01-01', 1);
-- 1@2001-01-01
SELECT scaleValue(tfloat '{[1@2001-01-01,2@2001-01-02], [3@2001-01-03,4@2001-01-04]}', 6);
-- {[1@2001-01-01, 3@2001-01-03], [5@2001-01-05, 7@2001-01-07]}
SELECT scaleValue(tint '1@2001-01-01', -1);
-- ERROR:  The value must be strictly positive: -1
SELECT shiftScaleValue(tint '1@2001-01-01', 1, 1);
-- 2@2001-01-01
SELECT shiftScaleValue(tfloat '{[1@2001-01-01,2@2001-01-02],[3@2001-01-03,4@2001-01-04]}',
  -1, 6);
--  {[0@2001-01-01, 2@2001-01-02], [4@2001-01-03, 6@2001-01-04]}
```

- Extract from a temporal float with linear interpolation the subsequences where the values stay within a span of a given width for at least a given duration

  `stops(tfloat,maxDist=0.0,minDuration='0 minutes')` → `tfloat`

  If `maxDist` is not given, a value 0.0 is assumed by default and thus, the function extracts the constant segments of the temporal float.

```
SELECT stops(tfloat '[1@2001-01-01, 1@2001-01-02, 2@2001-01-03]');
-- {[1@2001-01-01, 1@2001-01-02)}
SELECT stops(tfloat '[1@2001-01-01, 1@2001-01-02, 2@2001-01-03]', 0.0, '2 days');
-- NULL
```

6.5 Restrictions

- Restrict to (the complement of) the minimum or maximum value

  `atMin(torder)` → `torder`

  `atMax(torder)` → `torder`

  `minusMin(torder)` → `torder`

  `minusMax(torder)` → `torder`

  The functions returns a NULL value if the minimum value only happens at exclusive bounds.

```
SELECT atMin(tint '{1@2001-01-01, 2@2001-01-03, 1@2001-01-05}');
-- {1@2001-01-01, 1@2001-01-05}
SELECT atMin(tint '(1@2001-01-01, 3@2001-01-03]');
-- {(1@2001-01-01, 1@2001-01-03)}
SELECT atMin(tfloat '(1@2001-01-01, 3@2001-01-03]');
-- NULL
SELECT atMin(ttext '{(AA@2001-01-01, AA@2001-01-03), (BB@2001-01-03, AA@2001-01-05]}');
-- {(AA@2001-01-01, AA@2001-01-03), [AA@2001-01-05]}
SELECT atMax(tint '{1@2001-01-01, 2@2001-01-03, 3@2001-01-05}');
-- {3@2001-01-05}
SELECT atMax(tfloat '(1@2001-01-01, 3@2001-01-03)');
-- NULL
SELECT atMax(tfloat '{(2@2001-01-01, 1@2001-01-03), [2@2001-01-03, 2@2001-01-05)}');
-- {[2@2001-01-03, 2@2001-01-05]}
SELECT atMax(ttext '{(AA@2001-01-01, AA@2001-01-03), (BB@2001-01-03, AA@2001-01-05]}');
-- {("BB"@2001-01-03, "BB"@2001-01-05)}
```

```
SELECT minusMin(tint '{1@2001-01-01, 2@2001-01-03, 1@2001-01-05}');
-- {2@2001-01-03}
SELECT minusMin(tfloat '[1@2001-01-01, 3@2001-01-03]');
-- {(1@2001-01-01, 3@2001-01-03]}
SELECT minusMin(tfloat '(1@2001-01-01, 3@2001-01-03)');
-- {(1@2001-01-01, 3@2001-01-03)}
SELECT minusMin(tint '{[1@2001-01-01, 1@2001-01-03), (1@2001-01-03, 1@2001-01-05)}');
-- NULL
SELECT minusMax(tint '{1@2001-01-01, 2@2001-01-03, 3@2001-01-05}');
-- {1@2001-01-01, 2@2001-01-03}
SELECT minusMax(tfloat '[1@2001-01-01, 3@2001-01-03]');
-- {[1@2001-01-01, 3@2001-01-03)}
SELECT minusMax(tfloat '(1@2001-01-01, 3@2001-01-03)');
-- {(1@2001-01-01, 3@2001-01-03)}
SELECT minusMax(tfloat '{[2@2001-01-01, 1@2001-01-03), [2@2001-01-03, 2@2001-01-05)}');
-- {(2@2001-01-01, 1@2001-01-03)}
SELECT minusMax(tfloat '{[1@2001-01-01, 3@2001-01-03), (3@2001-01-03, 1@2001-01-05)}');
-- {[1@2001-01-01, 3@2001-01-03), (3@2001-01-03, 1@2001-01-05)}
```

- Restrict to (the complement of) a `tbox`

  `atTbox(tnumber,tbox)` → `tnumber`

  `minusTbox(tnumber,tbox)` → `tnumber`

  Cuando el cuadro delimitador tiene dimensiones tanto de valores como temporales, las funciones restringen el número temporal con respecto al valor *y* las extensiones temporales del cuadro.

```
SELECT atTbox(tfloat '[0@2001-01-01, 3@2001-01-04)',
  tbox 'TBOXFLOAT XT((0,2),[2001-01-02, 2001-01-04])');
-- {[1@2001-01-02, 2@2001-01-03]}
```

```
SELECT minusTbox(tfloat '[1@2001-01-01, 4@2001-01-04)',
  'TBOXFLOAT XT((1,4),[2001-01-03, 2001-01-04])');
-- {[1@2001-01-01, 3@2001-01-03)}
WITH temp(temp, box) AS (
  SELECT tfloat '[1@2001-01-01, 4@2001-01-04)',
    tbox 'TBOXFLOAT XT((1,2),[2001-01-03, 2001-01-04])' )
SELECT minusValues(minusTime(temp, box::tstzspan), box::floatspan) FROM temp;
-- {[1@2001-01-01], [2@2001-01-02, 3@2001-01-03)}
```

6.6 Boolean Operations

- Temporal and, temporal or

  `{boolean,tbool} {&, |} {boolean,tbool} → tbool`

```
SELECT tbool '[true@2001-01-03, true@2001-01-05)' &
  tbool '[false@2001-01-03, false@2001-01-05)';
-- [f@2001-01-03, f@2001-01-05)
SELECT tbool '[true@2001-01-03, true@2001-01-05)' &
  tbool '{[false@2001-01-03, false@2001-01-04),
  [true@2001-01-04, true@2001-01-05)}';
-- {[f@2001-01-03, t@2001-01-04, t@2001-01-05)}
SELECT tbool '[true@2001-01-03, true@2001-01-05)' |
  tbool '[false@2001-01-03, false@2001-01-05)';
-- [t@2001-01-03, t@2001-01-05)
```

- Temporal not

  `~tbool → tbool`

```
SELECT ~tbool '[true@2001-01-03, true@2001-01-05)';
-- [f@2001-01-03, f@2001-01-05)
```

- Return the time when a temporal Boolean takes the value true

  `whenTrue(tbool) → tstzspanset`

```
SELECT whenTrue(tfloat '[1@2001-01-01, 4@2001-01-04, 1@2001-01-07]' #> 2);
-- {(2001-01-02, 2001-01-06)}
SELECT whenTrue(tdwithin(tgeompoint '[Point(1 1)@2001-01-01, Point(4 4)@2001-01-04,
  Point(1 1)@2001-01-07]', geometry 'Point(1 1)', sqrt(2)));
-- {[2001-01-01, 2001-01-02], [2001-01-06, 2001-01-07]}
```

6.7 Mathematical Operations

- Temporal addition, subtraction, multiplication, and division

  `{number,tnumber} {+, -, *, /} {number,tnumber} → tnumber`

  The temporal division will raise an error if the denominator is ever equal to zero during the common timespan of the arguments.

```
SELECT tint '[2@2001-01-01, 2@2001-01-04)' + 1;
-- [3@2001-01-01, 3@2001-01-04)
SELECT tfloat '[2@2001-01-01, 2@2001-01-04)' + tfloat '[1@2001-01-01, 4@2001-01-04)';
-- [3@2001-01-01, 6@2001-01-04)
SELECT tfloat '[1@2001-01-01, 4@2001-01-04)' +
  tfloat '{[1@2001-01-01, 2@2001-01-02), [1@2001-01-02, 2@2001-01-04)}';
-- {[2@2001-01-01, 4@2001-01-04), [3@2001-01-02, 6@2001-01-04)}
```

```
SELECT tint '[1@2001-01-01, 1@2001-01-04)' - tint '[2@2001-01-03, 2@2001-01-05)';
-- [-1@2001-01-03, -1@2001-01-04)
SELECT tfloat '[3@2001-01-01, 6@2001-01-04)' - tfloat '[2@2001-01-01, 2@2001-01-04)';
-- [1@2001-01-01, 4@2001-01-04)
```

```
SELECT tint '[1@2001-01-01, 4@2001-01-04]' * 2;
-- [2@2001-01-01, 8@2001-01-04]
SELECT tfloat '[1@2001-01-01, 4@2001-01-04)' * tfloat '[2@2001-01-01, 2@2001-01-04)';
-- [2@2001-01-01, 8@2001-01-04)
SELECT tfloat '[1@2001-01-01, 3@2001-01-03)' * '[3@2001-01-01, 1@2001-01-03)'
-- {[3@2001-01-01, 4@2001-01-02, 3@2001-01-03)}
```

```
SELECT 2 / tfloat '[1@2001-01-01, 3@2001-01-04)';
-- [2@2001-01-01, 0.6666666666666667@2001-01-04)
SELECT tfloat '[1@2001-01-01, 5@2001-01-05)' / tfloat '[5@2001-01-01, 1@2001-01-05)';
-- {[0.2@2001-01-01, 1@2001-01-03,2001-01-03, 5@2001-01-03,2001-01-05)}
SELECT 2 / tfloat '[-1@2001-01-01, 1@2001-01-02]';
-- ERROR:  Division by zero
SELECT tfloat '[-1@2001-01-04, 1@2001-01-05]' / tfloat '[-1@2001-01-01, 1@2001-01-05]';
-- [-2@2001-01-04, 1@2001-01-05]
```

- Return the absolute value of the temporal number

  `abs(tnumber) → tnumber`

```
SELECT abs(tfloat '[1@2001-01-01, -1@2001-01-03, 1@2001-01-05]');
-- [1@2001-01-01, 0@2001-01-02, 1@2001-01-03, 0@2001-01-04, 1@2001-01-05]
SELECT abs(tint '[1@2001-01-01, -1@2001-01-03, 1@2001-01-05]');
-- [1@2001-01-01, 1@2001-01-05]
```

- Round up or down to the neareast integer

  `floor(tfloat) → tfloat`

  `ceil(tfloat) → tfloat`

```
SELECT floor(tfloat '[0.5@2001-01-01, 1.5@2001-01-02]');
-- [0@2001-01-01, 1@2001-01-02]
SELECT ceil(tfloat '[0.5@2001-01-01, 0.6@2001-01-02, 0.7@2001-01-03]');
-- [1@2001-01-01, 1@2001-01-03]
```

- Round to a number of decimal places

  `round(tfloat,integer=0) → tfloat`

```
SELECT round(tfloat '[0.7853981633974480@2001-01-01, 2.3561944901923450@2001-01-02]', 2);
-- [0.79@2001-01-01, 2.36@2001-01-02]
```

- Convert to degrees or radians

  `degrees({float,tfloat},normalize=false) → tfloat`

  `radians(tfloat) → tfloat`

  The additional parameter in the `degrees` function can be used to normalize the values between 0 and 360 degrees.

```
SELECT degrees(pi() * 5);
-- 900
SELECT degrees(pi() * 5, true);
-- 180
SELECT round(degrees(tfloat '[0.7853981633974480@2001-01-01, 2.3561944901923450@2001-01-02]' ↩
    ));
-- [45@2001-01-01, 135@2001-01-02]
SELECT radians(tfloat '[45@2001-01-01, 135@2001-01-02]');
-- [0.7853981633974480@2001-01-01, 2.3561944901923450@2001-01-02]
```

- Return the value difference between consecutive instants of the temporal number

  `deltaValue(tnumber) → tnumber`

```
SELECT deltaValue(tint '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03]');
-- [1@2001-01-01, -1@2001-01-02, -1@2001-01-03)
SELECT deltaValue(tfloat '{[1.5@2001-01-01, 2@2001-01-02, 1@2001-01-03],
  [2@2001-01-04, 2@2001-01-05]}');
/* Interp=Step;{[0.5@2001-01-01, -1@2001-01-02, -1@2001-01-03),
  [0@2001-01-04, 0@2001-01-05)} */
```

- Return the trend of a temporal float with linear interpolation, which states whether its value is increasing, constant, or decreasing, represented, respectively, by 1, 0, and -1.

  `trend(tfloat) → tint`

  Note that the trend is NULL for instantaneous sequences.

```
SELECT trend(tfloat '[1@2001-01-01, 2@2001-01-02, 4@2001-01-03, 4@2001-01-04, 3@2001 ↩
    -01-05, 2@2001-01-06]');
-- [1@2001-01-01, 0@2001-01-03, -1@2001-01-04, -1@2001-01-06]
SELECT trend(tfloat '[1@2001-01-01]');
-- NULL
```

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

  `derivative(tfloat) → tfloat`

  The temporal float must have linear interpolation. Note that this function corresponds to the speed function for temporal points.

```
SELECT derivative(tfloat '{[0@2001-01-01, 10@2001-01-02, 5@2001-01-03],
  [1@2001-01-04, 0@2001-01-05]}') * 3600 * 24;
/* Interp=Step;{[-10@2001-01-01, 5@2001-01-02, 5@2001-01-03],
  [1@2001-01-04, 1@2001-01-05]} */
SELECT derivative(tfloat 'Interp=Step;[0@2001-01-01, 10@2001-01-02, 5@2001-01-03]');
-- ERROR:  The temporal value must have linear interpolation
```

- Return the area under the curve

  `integral(tnumber) → float`

```
SELECT  integral(tint '[1@2001-01-01,2@2001-01-02]') / (24 * 3600 * 1e6);
-- 1
SELECT integral(tfloat '[1@2001-01-01,2@2001-01-02]') / (24 * 3600 * 1e6);
-- 1.5
```

- Return the time-weighted average

  `twAvg(tnumber) → float`

```
SELECT twAvg(tfloat '{[1@2001-01-01, 2@2001-01-03), [2@2001-01-04, 2@2001-01-06)}');
-- 1.75
```

- Return the natural logarithm and the base 10 logarithm of a temporal float

  `ln(tfloat) → tfloat`

  `log10(tfloat) → tfloat`

  The temporal float cannot be zero or negative

```
SELECT ln(tfloat '{[1@2001-01-01, 10@2001-01-02, 5@2001-01-03],
  [1@2001-01-04, 1@2001-01-05]}');
/* {[0@2001-01-01, 2.3025850929940046@2001-01-02, 1.6094379124341@2001-01-03],
   [0@2001-01-04, 0@2001-01-05]} */
SELECT log10(tfloat 'Interp=Step;[-10@2001-01-01, 10@2001-01-02]');
-- ERROR:  Cannot take logarithm of zero or a negative number
```

- Return the exponential (e raised to the given power) of a temporal float

  `exp(tfloat) → tfloat`

```
SELECT exp(tfloat '{[1@2001-01-01, 10@2001-01-02],
  [1@2001-01-04, 1@2001-01-05]}');
/* {[2.7182818284590045@2001-01-01, 22026.465794806718@2001-01-02],
   [2.7182818284590045@2001-01-04, 2.7182818284590045@2001-01-05]} */
SELECT exp(tfloat '{-10@2001-01-01, 0@2001-01-02, 10@2001-01-03}');
-- {0.0000453999929762@2001-01-01, 1@2001-01-02, 22026.465794806718@2001-01-03}
```

6.8 Text Operations

- Text concatenation

  `{text,ttext} || {text,ttext} → ttext`

```
SELECT ttext '[AA@2001-01-01, AA@2001-01-04)' || text 'B';
-- ["AAB"@2001-01-01, "AAB"@2001-01-04)
SELECT ttext '[AA@2001-01-01, AA@2001-01-04)' || ttext '[BB@2001-01-02, BB@2001-01-05)';
-- ["AABB"@2001-01-02, "AABB"@2001-01-04)
SELECT ttext '[A@2001-01-01, B@2001-01-03, C@2001-01-04]' ||
  ttext '{[D@2001-01-01, D@2001-01-02), [E@2001-01-02, E@2001-01-04)}';
-- {["AD"@2001-01-01, "AE"@2001-01-02, "BE"@2001-01-03, "BE"@2001-01-04)}
```

- Transform in lowercase, uppercase, or initcap

  `upper(ttext) → ttext`

  `lower(ttext) → ttext`

  `initcap(ttext) → ttext`

```
SELECT lower(ttext '[AA@2001-01-01, bb@2001-01-02]');
-- ["aa"@2001-01-01, "bb"@2001-01-02]
SELECT upper(ttext '[AA@2001-01-01, bb@2001-01-02]');
-- ["AA"@2001-01-01, "BB"@2001-01-02]
SELECT initcap(ttext '[AA@2001-01-01, bb@2001-01-02]');
-- ["Aa"@2001-01-01, "Bb"@2001-01-02]
```

Chapter 7

Temporal Geometry Types (Part 1)

7.1 Spatiotemporal Types

MobilityDB provides a set of spatiotemporal types, namely, `tgeometry` (temporal geometry), `tgeography` (temporal geography), `tgeompoint` (temporal geometry point), `tgeogpoint` (temporal geography point), `tcbuffer` (temporal circular buffer), `tnpoint` (temporal network point), `tpose` (temporal pose), and `trgeometry` (temporal rigid geometry). At a conceptual level, these spatiotemporal types are organized in the hierarchy depicted in Figure 7.1.

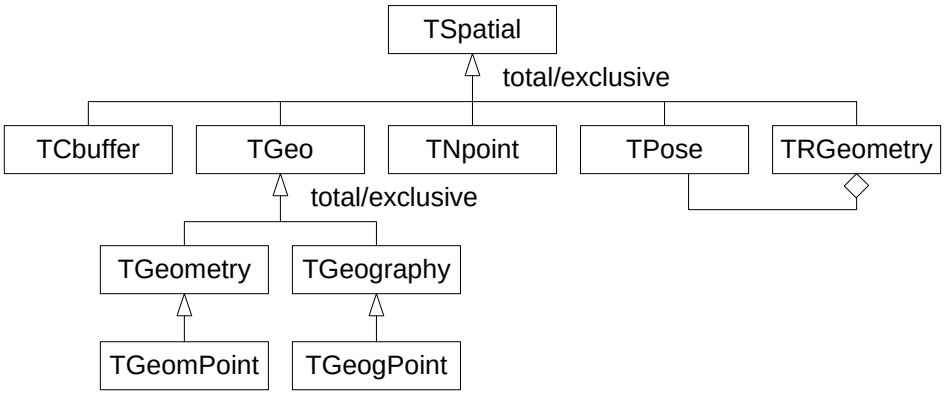


Figure 7.1: Hierarchy of spatiotemporal types in MobilityDB.

In this chapter and the following one we cover the type `TGeo` and its subtypes, that is, the spatiotemporal types derived from the PostGIS types `geometry` and `geography`. In the subsequent chapters we continue describing the remaining spatiotemporal types.

Figure 7.2 illustrates the use of the `tgeompoint` (top) and the `tgeometry` (bottom) types for modelling the trajectory and the wind swath of tropical storms in 2024. The data is obtained from the National Oceanic and Atmospherique Administration (NOAA). As illustrated in the figure, the type `tgeompoint` allows *linear* interpolation, while the type `tgeometry` only allows *step* interpolation.

7.2 Notation

We presented in Section 4.5 and in Section 6.1 the notation used for defining the signature of the functions and operators for temporal types. We extend next this notations for spatiotemporal types.

- `tspatial` represents a spatiotemporal type, such as, `tgeometry`, `tgeompoint`, `tpose`, or `tnpoint`,

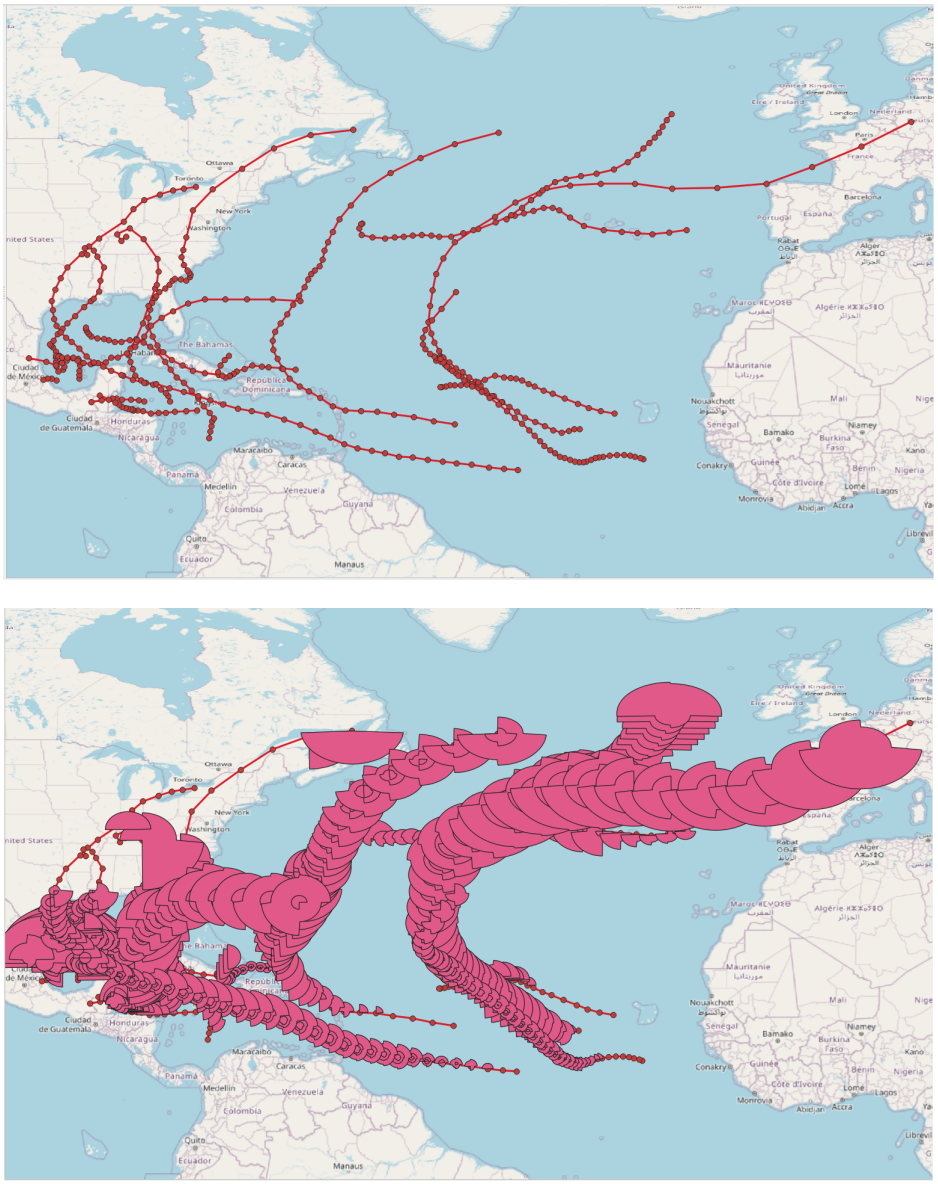


Figure 7.2: Illustration of the use of the `tgeompoint` (top) and the `tgeometry` (bottom) types for modelling the trajectory and the wind swath of tropical storms in 2024.

- `tgeo` represents a temporal geometry/geography type, that is, `tgeometry`, `tgeography`, `tgeompoint`, or `tgeogpoint`,
- `tgeom` represents a temporal geometry type that is, `tgeometry` or `tgeompoint`,
- `tpoint` represents a temporal point type, that is, `tgeompoint` or `tgeogpoint`,
- `spatial` represents a spatial base type, such as, `geometry`, `geography`, `pose`, or `npoint`,
- `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.
- `lines` represents the types `geometry` or `geography` restricted to a (multi)line.

In the following, we specify with the symbol 📦 that the function supports 3D geometries and with the symbol 🌐 that the function supports geographies.

7.3 Input and Output

- Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation 📦🌐

  `asText({tspatial,tspatial[],spatial[]}) → {text,text[]}`

  `asEWKT({tspatial,tspatial[],spatial[]}) → {text,text[]}`

```
SELECT asText(tgeompoint 'SRID=4326;[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02)');
-- [POINT Z (0 0 0)@2001-01-01, POINT Z (1 1 1)@2001-01-02)
SELECT asText(ARRAY[tgeometry 'SRID=4326;[Point(0 0)@2001-01-01,
  Linestring(1 1,2 1)@2001-01-02]', 'Polygon((1 1,2 2,3 1,1 1))@2001-01-01']);
/* {"[POINT(0 0)@2001-01-01, LINESTRING(1 1,2 1)@2001-01-02]",
    "POLYGON((1 1,2 2,3 1,1 1))@2001-01-01"} */
SELECT asText(ARRAY[geometry 'Point(0 0)', 'Point(1 1)']);
-- {"POINT(0 0)","POINT(1 1)"}
```

```
SELECT asEWKT(tgeompoint 'SRID=4326;[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02)');
-- SRID=4326;[POINT Z (0 0 0)@2001-01-01, POINT Z (1 1 1)@2001-01-02)
SELECT asEWKT(ARRAY[tgeometry 'SRID=4326;[Point(0 0)@2001-01-01,
  Linestring(1 1,2 1)@2001-01-02]', 'Polygon((1 1,2 2,3 1,1 1))@2001-01-01']);
-- {"SRID=4326;[POINT(0 0)@2001-01-01, LINESTRING(1 1,2 1)@2001-01-02]",
    "POLYGON((1 1,2 2,3 1,1 1))@2001-01-01"}
SELECT asEWKT(ARRAY[geometry 'SRID=5676;Point(0 0)', 'SRID=5676;Point(1 1)']);
-- {"SRID=5676;POINT(0 0)","SRID=5676;POINT(1 1)"}
```

- Return the Moving Features JSON representation 📦🌐

  `asMFJSON(tspatial,options=0,flags=0,maxdecdigits=15) → bytea`

  The `options` argument can 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)

  The `flags` argument can be used to customize the JSON output, for example, to produce an easy-to-read (for human readers) JSON output. Refer to the documentation of the `json-c` library for the possible values. Typical values are as follows:

  - 0: means no option (default value)

- 1: JSON_C_TO_STRING_SPACED
- 2: JSON_C_TO_STRING_PRETTY

The `maxdecdigits` argument can be used to set the maximum number of decimal places in the output of floating point values (default 15).

```
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"],
    "interpolation":"None"} */
SELECT asMFJSON(tgeometry 'SRID=3812;Linestring(1 1,1 2)@2019-01-01 18:00:00.15+02');
/* {"type":"MovingGeometry",
    "crs":{"type":"Name","properties":{"name":"EPSG:3812"}},
    "values":[{"type":"LineString","coordinates":[[1,1],[1,2]]}],
    "datetimes":["2019-01-01T17:00:00.15+01"],"interpolation":"None"} */
SELECT asMFJSON(tgeompoint 'SRID=4326;
  Point(50.813810 4.384260)@2019-01-01 18:00:00.15+02', 3, 0, 2);
/* {"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":"None"} */
```

- Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB) representation, or the Hexadecimal Extended Well-Known Binary (EWKB) representation

`asBinary(tspatial,endian text='') → bytea`

`asEWKB(tspatial,endian text='') → bytea`

`asHexEWKB(tspatial,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 the encoding of the machine is used.

```
SELECT asBinary(tgeompoint 'Point(1 2 3)@2001-01-01');
-- \x012e0011000000000000f03f000000000000004000000000000000840009c57d3c11c0000
SELECT asEWKB(tgeogpoint 'SRID=7844;Point(1 2 3)@2001-01-01');
-- \x012f0071a41e0000000000000000f03f000000000000004000000000000000840009c57d3c11c0000
SELECT asHexEWKB(tgeompoint 'SRID=3812;Point(1 2 3)@2001-01-01');
-- 012E0051E40E0000000000000000F03F000000000000004000000000000000840009C57D3C11C0000
```

- Input from the Well-Known Text (WKT) or from the Extended Well-Known Text (EWKT) representation

`tspatialFromText(text) → tspatial`

`tspatialFromEWKT(text) → tspatial`

In the above functions, `tspatial` replaces any spatial type, such as `tgeompoint`, `tgeometry`, or `tpose`.

```
SELECT asEWKT(tgeompointFromText(text '[POINT(1 2)@2001-01-01, POINT(3 4)@2001-01-02]'));
-- [POINT(1 2)@2001-01-01, POINT(3 4)@2001-01-02]
SELECT asEWKT(tgeographyFromText(text
  '[Point(1 2)@2001-01-01, Linestring(1 2,3 4)@2001-01-02]'));
-- SRID=4326;[POINT(1 2)@2001-01-01, LINESTRING(1 2,3 4)@2001-01-02]
```

```
SELECT asEWKT(tgeompointFromEWKT(text 'SRID=3812;[Point(1 2)@2001-01-01,
  Point(3 4)@2001-01-02]'));
-- SRID=3812;[POINT(1 2)@2001-01-01, POINT(3 4)@2001-01-02]
SELECT asEWKT(tgeographyFromEWKT(text 'SRID=7844;[Point(1 2)@2001-01-01,
  Linestring(1 2,3 4)@2001-01-02]'));
-- SRID=7844;[POINT(1 2)@2001-01-01, LINESTRING(1 2,3 4)@2001-01-02]
```

- Input from the Moving Features JSON representation

  `tspatialFromMFJSON(text) → tspatial`

  In the above function, `tspatial` replaces any spatiotemporal type, for example, `tgeompoint`, `tgeometry`, or `tpose`

```
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"],"interpolation":"None"}'));
-- SRID=4326;POINT(50.81 4.38)@2019-01-01 17:00:00.15+01
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"],"interpolation":"None"}'));
-- SRID=4326;POINT(50.81 4.38)@2019-01-01 17:00:00.15+01
SELECT asEWKT(tgeographyFromMFJSON(text '{"type":"MovingGeometry",
  "crs":{"type":"Name","properties":{"name":"EPSG:7844"}},
  "values":[{"type":"LineString","coordinates":[[1,1],[1,2]]}],
  "datetimes":["2019-01-01T17:00:00.15+01"],"interpolation":"None"}'));
-- SRID=7844;LINESTRING(1 1,1 2)@2019-01-01 17:00:00.15+01
```

- Input from the Well-Known Binary (WKB), from the Extended Well-Known Binary (EWKB), or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `tspatialFromBinary(bytea) → tspatial`

  `tspatialFromEWKB(bytea) → tspatial`

  `tspatialFromHexEWKB(text) → tspatial`

  In the previous functions, `tspatial` replaces any spatiotemporal type, for example, `tgeompoint`, `tgeometry`, or `tpose`

```
SELECT asEWKT(tgeompointFromBinary(
  '\x012e001100000000000f03f0000000000000040000000000000840009c57d3c11c0000'));
-- POINT Z (1 2 3)@2001-01-01
SELECT asEWKT(tgeogpointFromEWKB(
  '\x012f0071a41e0000000000000000f03f0000000000000040000000000000000840009c57d3c11c0000'));
-- SRID=7844;POINT Z (1 1 1)@2001-01-01
SELECT asEWKT(tgeompointFromHexEWKB(
  '012E0051E40E0000000000000000F03F0000000000000040000000000000000840009C57D3C11C0000'));
-- SRID=3812;POINT(1 2 3)@2001-01-01
```

7.4 Conversions

- Convert a spatiotemporal value to a spatiotemporal box

  `tspatial::stbox`

  `stbox(tspatial)`

```
SELECT tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03]'::stbox;
-- STBOX XT(((1,1),(3,3)),[2001-01-01, 2001-01-03])
SELECT tgeography '[Point(1 1 1)@2001-01-01, Point(3 3 3)@2001-01-03]'::stbox;
-- SRID=4326;GEODSTBOX ZT(((1,1,1),(3,3,3)),[2001-01-01, 2001-01-03])
```

- Convert between a temporal geometry and a temporal geography

  `tgeometry::tgeography`

  `tgeography::tgeometry`

  `tgeompoint::tgeogpoint`

  `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)
SELECT asText((tgeography 'Linestring(0 0,1 1)@2001-01-01')::tgeometry);
-- LINESTRING(0 0,1 1)@2001-01-01
```

A common way to store temporal points in PostGIS is to represent them as geometries of type `LINESTRING M` and use 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.

- Convert between a temporal point and a PostGIS trajectory

  `tpoint::geo`

  `geo::tpoint`

```
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)) */
```

```
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]} */
```

7.5 Accessors

- Return the trajectory or the traversed area

  `trajectory(tpoint,unary_union=false)` → `geo`

  `traversedArea(tgeo,unary_union=false)` → `geo`

  This function is equivalent to getValues for temporal alphanumeric values. The last argument states whether the PostGIS function ST_UnaryUnion is applied to remove redundant geometries in the result. Notice that setting this argument to true is computationally expensive.

```
SELECT ST_AsText(trajectory(tgeompoint '[Point(0 0)@2001-01-01, Point(0 1)@2001-01-02,
  Point(0 0)@2001-01-03]'));
-- LINESTRING(0 0,0 1,0 0)
SELECT ST_AsText(trajectory(tgeompoint '[Point(0 0)@2001-01-01, Point(0 1)@2001-01-02,
```

```
  Point(0 0)@2001-01-03]', true));
-- LINESTRING(0 0,0 1)
SELECT ST_AsText(trajectory(tgeompoint '{[Point(0 0)@2001-01-01, Point(0 1)@2001-01-02),
  [Point(0 1)@2001-01-03, Point(0 0)@2001-01-04)}'));
-- LINESTRING(0 0,0 1)
SELECT ST_AsText(trajectory(tgeompoint 'Interp=Step;{[Point(0 0)@2001-01-01,
  Point(0 1)@2001-01-02], [Point(0 1)@2001-01-03, Point(1 1)@2001-01-04]}'));
-- MULTIPOINT((0 0),(1 1),(0 1))
```

```
SELECT ST_AsText(traversedArea(tgeometry '[Point(1 1)@2001-01-01,
  Linestring(1 1,2 2)@2001-01-02, Point(1 1)@2001-01-03]'));
-- GEOMETRYCOLLECTION(POINT(1 1),LINESTRING(1 1,2 2))
SELECT ST_AsText(traversedArea(tgeometry '[Point(1 1)@2001-01-01,
  Linestring(1 1,2 2)@2001-01-02, Point(1 1)@2001-01-03]', true));
-- LINESTRING(1 1,2 2)
```

- Return the centroid as a temporal point

  `centroid(tgeo) → tpoint`

```
SELECT asText(centroid(tgeometry '[Point(1 1)@2000-01-01, Linestring(1 1,3 3)@2000-01-02,
  Polygon((1 1,4 4,7 1,1 1))@2000-01-03]'));
-- Interp=Step;[POINT(1 1)@2000-01-01, POINT(2 2)@2000-01-02, POINT(4 2)@2000-01-03]
SELECT asText(centroid(tgeography '[MultiPoint(1 1,4 4,7 1)@2000-01-01,
  Polygon((1 1,4 4,7 1,1 1))@2000-01-02]'),6);
-- Interp=Step;[POINT(4 2.001727)@2000-01-01, POINT(4 2.001727)@2000-01-02]
```

- Return the X/Y/Z coordinate values as a temporal float

  `getX(tpoint) → tfloat`

  `getY(tpoint) → tfloat`

  `getZ(tpoint) → tfloat`

```
SELECT getX(tgeompoint '{Point(1 2)@2001-01-01, Point(3 4)@2001-01-02,
  Point(5 6)@2001-01-03}');
-- {1@2001-01-01, 3@2001-01-02, 5@2001-01-03}
SELECT getX(tgeogpoint 'Interp=Step;[Point(1 2 3)@2001-01-01, Point(4 5 6)@2001-01-02,
  Point(7 8 9)@2001-01-03]');
-- Interp=Step;[1@2001-01-01, 4@2001-01-02, 7@2001-01-03]
SELECT getY(tgeompoint '{Point(1 2)@2001-01-01, Point(3 4)@2001-01-02,
  Point(5 6)@2001-01-03}');
-- {2@2001-01-01, 4@2001-01-02, 6@2001-01-03}
SELECT getY(tgeogpoint 'Interp=Step;[Point(1 2 3)@2001-01-01, Point(4 5 6)@2001-01-02,
  Point(7 8 9)@2001-01-03]');
-- Interp=Step;[2@2001-01-01, 5@2001-01-02, 8@2001-01-03]
SELECT getZ(tgeompoint '{Point(1 2)@2001-01-01, Point(3 4)@2001-01-02,
  Point(5 6)@2001-01-03}');
-- The temporal point must have Z dimension
SELECT getZ(tgeogpoint 'Interp=Step;[Point(1 2 3)@2001-01-01, Point(4 5 6)@2001-01-02,
  Point(7 8 9)@2001-01-03]');
-- Interp=Step;[3@2001-01-01, 6@2001-01-02, 9@2001-01-03]
```

- Return 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)@2001-01-01, Point(1 1)@2001-01-02,
  Point(0 0)@2001-01-03]');
-- false
```

```
SELECT isSimple(tgeompoint '[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02,
  Point(2 0 2)@2001-01-03, Point(0 0 0)@2001-01-04]');
-- false
SELECT isSimple(tgeompoint '{[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02],
  [Point(1 1 1)@2001-01-03, Point(0 0 0)@2001-01-04]}');
-- true
```

- Return the length traversed by the temporal point

  `length(tpoint) → float`

```
SELECT length(tgeompoint '[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02]');
-- 1.73205080756888
SELECT length(tgeompoint '[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02,
  Point(0 0 0)@2001-01-03]');
-- 3.46410161513775
SELECT length(tgeompoint 'Interp=Step;[Point(0 0 0)@2001-01-01,
  Point(1 1 1)@2001-01-02, Point(0 0 0)@2001-01-03]');
-- 0
```

- Return the cumulative length traversed by the temporal point

  `cumulativeLength(tpoint) → tfloatSeq`

```
SELECT round(cumulativeLength(tgeompoint '{[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02,
  Point(1 0)@2001-01-03], [Point(1 0)@2001-01-04, Point(0 0)@2001-01-05]}'), 6);
-- {[0@2001-01-01, 1.414214@2001-01-02, 2.414214@2001-01-03],
  [2.414214@2001-01-04, 3.414214@2001-01-05]}
SELECT cumulativeLength(tgeompoint 'Interp=Step;[Point(0 0 0)@2001-01-01,
  Point(1 1 1)@2001-01-02, Point(0 0 0)@2001-01-03]');
-- Interp=Step;[0@2001-01-01, 0@2001-01-03]
```

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

  `speed(tpoint) → tfloatSeqSet`

  The temporal point must have linear interpolation

```
SELECT speed(tgeompoint '{[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02,
  Point(1 0)@2001-01-03], [Point(1 0)@2001-01-04, Point(0 0)@2001-01-05]}') * 3600 * 24;
/* Interp=Step;{[1.4142135623731@2001-01-01, 1@2001-01-02, 1@2001-01-03],
   [1@2001-01-04, 1@2001-01-05]} */
SELECT speed(tgeompoint 'Interp=Step;[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02,
  Point(1 0)@2001-01-03]');
-- ERROR:  The temporal value must have linear interpolation
```

- Return the time-weighted centroid

  `twCentroid(tgeompoint) → point`

```
SELECT ST_AsText(twCentroid(tgeompoint '{[Point(0 0 0)@2001-01-01,
  Point(0 1 1)@2001-01-02, Point(0 1 1)@2001-01-03, Point(0 0 0)@2001-01-04)}'));
-- POINT Z (0 0.666666666666667 0.666666666666667)
```

- Return the direction, that is, the azimuth between the start and end locations

  `direction(tpoint) → float`

  The result is expressed in radians. It is NULL if there is only one location or if the start and end locations are equal.

```
SELECT round(degrees(direction(tgeompoint '[Point(0 0)@2001-01-01,
  Point(-1 -1)@2001-01-02, Point(1 1)@2001-01-03]'))::numeric, 6);
-- 45.000000
```

```
SELECT direction(tgeompoint '{[Point(0 0 0)@2001-01-01,
  Point(0 1 1)@2001-01-02, Point(0 1 1)@2001-01-03, Point(0 0 0)@2001-01-04)}');
-- NULL
```

- Return the temporal azimuth

  `azimuth(tpoint) → tfloat`

  The result is expressed in radians. The azimut is undefined when two succesive locations are equal and in this case a temporal gap is added.

```
SELECT round(degrees(azimuth(tgeompoint '[Point(0 0 0)@2001-01-01,
  Point(1 1 1)@2001-01-02, Point(1 1 1)@2001-01-03, Point(0 0 0)@2001-01-04)')));
-- Interp=Step;{[45@2001-01-01, 45@2001-01-02], [225@2001-01-03, 225@2001-01-04)}
```

- Return the temporal angular difference

  `angularDifference(tpoint) → tfloat`

  The result is expressed in degrees.

```
SELECT round(angularDifference(tgeompoint '[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02,
  Point(1 1)@2001-01-03]'), 3);
-- {0@2001-01-01, 180@2001-01-02, 0@2001-01-03}
SELECT round(degrees(angularDifference(tgeompoint '{[Point(1 1)@2001-01-01,
  Point(2 2)@2001-01-02], [Point(2 2)@2001-01-03, Point(1 1)@2001-01-04]}')), 3);
-- {0@2001-01-01, 0@2001-01-02, 0@2001-01-03, 0@2001-01-04}
```

- Return the temporal bearing

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

  Notice that this function does not accept two temporal geographic points.

```
SELECT degrees(bearing(tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03]',
  geometry 'Point(2 2)'));
-- [45@2001-01-01, 0@2001-01-02, 225@2001-01-03]
SELECT round(degrees(bearing(tgeompoint '[Point(0 0)@2001-01-01, Point(2 0)@2001-01-03]',
  tgeompoint '[Point(2 1)@2001-01-01, Point(0 1)@2001-01-03]')), 3);
--  [63.435@2001-01-01, 0@2001-01-02, 296.565@2001-01-03]
SELECT round(degrees(bearing(tgeompoint '[Point(2 1)@2001-01-01, Point(0 1)@2001-01-03]',
  tgeompoint '[Point(0 0)@2001-01-01, Point(2 0)@2001-01-03]')), 3);
-- [243.435@2001-01-01, 116.565@2001-01-03]
```

7.6 Transformations

- Round the coordinate values to a number of decimal places

  `round(tspatial,integer=0) → tspatial`

```
SELECT asText(round(tgeompoint '{Point(1.12345 1.12345 1.12345)@2001-01-01,
  Point(2 2 2)@2001-01-02, Point(1.12345 1.12345 1.12345)@2001-01-03}', 2));
/* {POINT Z (1.12 1.12 1.12)@2001-01-01, POINT Z (2 2 2)@2001-01-02,
   POINT Z (1.12 1.12 1.12)@2001-01-03} */
SELECT asText(round(tgeography 'Linestring(1.12345 1.12345,2.12345 2.12345)@2001-01-01',
    2));
-- LINESTRING(1.12 1.12,2.12 2.12)@2001-01-01
```

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

  `makeSimple(tpoint) → tgeompoint[]`

```
SELECT asText(makeSimple(tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02,
  Point(0 0)@2001-01-03]'));
/* {"[POINT(0 0)@2001-01-01, POINT(1 1)@2001-01-02)",
   "[POINT(1 1)@2001-01-02, POINT(0 0)@2001-01-03]"} */
SELECT asText(makeSimple(tgeompoint '[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02,
  Point(2 0 2)@2001-01-03, Point(0 0 0)@2001-01-04]'));
/* {"[POINT Z (0 0 0)@2001-01-01, POINT Z (1 1 1)@2001-01-02, POINT Z (2 0 2)@2001-01-03,
   POINT Z (0 0 0)@2001-01-04]"} */
SELECT asText(makeSimple(tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02,
  Point(0 1)@2001-01-03, Point(1 0)@2001-01-04]'));
/* {"[POINT(0 0)@2001-01-01, POINT(1 1)@2001-01-02, POINT(0 1)@2001-01-03)",
    "[POINT(0 1)@2001-01-03, POINT(1 0)@2001-01-04]"} */
SELECT asText(makeSimple(tgeompoint '{[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-02],
  [Point(1 1 1)@2001-01-03, Point(0 0 0)@2001-01-04]}'));
/* {"{[POINT Z (0 0 0)@2001-01-01, POINT Z (1 1 1)@2001-01-02],
   [POINT Z (1 1 1)@2001-01-03, POINT Z (0 0 0)@2001-01-04]}"} */
```

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

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

  The last argument `segmentize` 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)@2001-01-01,
  Point(2 2 2)@2001-01-02}', '{5@2001-01-01, 5@2001-01-02}'));
-- MULTIPOINT ZM (1 1 1 5,2 2 2 5)
SELECT ST_AsText(geoMeasure(tgeogpoint '{[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02],
  [Point(1 1)@2001-01-03, Point(1 1)@2001-01-04]}',
  '{[5@2001-01-01, 5@2001-01-02],[7@2001-01-03, 7@2001-01-04]}'));
-- GEOMETRYCOLLECTION M (POINT M (1 1 7),LINESTRING M (1 1 5,2 2 5))
SELECT ST_AsText(geoMeasure(tgeompoint '[Point(1 1)@2001-01-01,
  Point(2 2)@2001-01-02, Point(1 1)@2001-01-03]',
  '[5@2001-01-01, 7@2001-01-02, 5@2001-01-03]', 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 7.3 shows the result of the query below using a color ramp in QGIS.

```
WITH Temp(t) AS (
  SELECT tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-05,
    Point(2 0)@2001-01-08, Point(3 1)@2001-01-10, Point(4 0)@2001-01-11]' )
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) )
```

- Return the 3D affine transform of a temporal geometry to translate, rotate, and/or scale it in a single step

  `affine(tgeo,float a,float b,float c,float d,float e,float f,float g,`
  `float h,float i,float xoff,float yoff,float zoff)` → `tgeo`

  `affine(tgeo,float a,float b,float d,float e,float xoff,float yoff)` → `tgeo`

Figure 7.3: Visualizing the speed of a moving object using a color ramp in QGIS.

```
-- Rotate a 3D temporal point 180 degrees about the z axis
SELECT asEWKT(affine(temp, cos(pi()), -sin(pi()), 0, sin(pi()), cos(pi()), 0, 0, 0, 1,
  0, 0, 0))
FROM (SELECT tgeompoint '[POINT(1 2 3)@2001-01-01, POINT(1 4 3)@2001-01-02]' AS temp) t;
-- [POINT Z (-1 -2 3)@2001-01-01, POINT Z (-1 -4 3)@2001-01-02]
SELECT asEWKT(rotate(temp, pi()))
FROM (SELECT tgeompoint '[POINT(1 2 3)@2001-01-01, POINT(1 4 3)@2001-01-02]' AS temp) t;
-- [POINT Z (-1 -2 3)@2001-01-01, POINT Z (-1 -4 3)@2001-01-02]
-- Rotate a 3D temporal point 180 degrees in both the x and z axis
SELECT asEWKT(affine(temp, cos(pi()), -sin(pi()), 0, sin(pi()), cos(pi()), -sin(pi()),
   0, sin(pi()), cos(pi()), 0, 0, 0))
FROM (SELECT tgeometry '[Point(1 1)@2001-01-01,
  Linestring(1 1,2 2)@2001-01-02]' AS temp) t;
-- [POINT(-1 -1)@2001-01-01, LINESTRING(-1 -1,-2 -2)@2001-01-02]
```

- Return the temporal point rotated counter-clockwise about the origin point

  `rotate(tgeo,float radians)` → `tgeo`

  `rotate(tgeo,float radians,float x0,float y0)` → `tgeo`

  `rotate(tgeo,float radians,geometry origin)` → `tgeo`

```
-- Rotate a temporal point 180 degrees
SELECT asEWKT(rotate(tgeompoint '[Point(5 10)@2001-01-01, Point(5 5)@2001-01-02,
  Point(10 5)@2001-01-03]', pi()), 6);
-- [POINT(-5 -10)@2001-01-01, POINT(-5 -5)@2001-01-02, POINT(-10 -5)@2001-01-03]
-- Rotate 30 degrees counter-clockwise at x=5, y=10
SELECT asEWKT(rotate(tgeompoint '[Point(5 10)@2001-01-01, Point(5 5)@2001-01-02,
  Point(10 5)@2001-01-03]', pi()/6, 5, 10), 6);
-- [POINT(5 10)@2001-01-01, POINT(7.5 5.67)@2001-01-02, POINT(11.83 8.17)@2001-01-03]
-- Rotate 60 degrees clockwise from centroid
SELECT asEWKT(rotate(temp, -pi()/3, ST_Centroid(traversedArea(temp))), 2)
FROM (SELECT tgeometry '[Point(5 10)@2001-01-01, Point(5 5)@2001-01-02,
  Linestring(5 5,10 5)@2001-01-03]' AS temp) AS t;
/* [POINT(10.58 9.67)@2001-01-01, POINT(6.25 7.17)@2001-01-02,
    LINESTRING(6.25 7.17,8.75 2.83)@2001-01-03] */
```

- Return a temporal point scaled by given factors

  `scale(tgeo,float Xfactor,float Yfactor,float Zfactor)` → `tgeo`

  `scale(tgeo,float Xfactor,float Yfactor)` → `tgeo`

  `scale(tgeo,geometry factor)` → `tgeo`

  `scale(tgeo,geometry factor,geometry origin)` → `tgeo`

```
SELECT asEWKT(scale(tgeompoint '[Point(1 2 3)@2001-01-01, Point(1 1 1)@2001-01-02]',
  0.5, 0.75, 0.8));
--  [POINT Z (0.5 1.5 2.4)@2001-01-01, POINT Z (0.5 0.75 0.8)@2001-01-02]
SELECT asEWKT(scale(tgeompoint '[Point(1 2 3)@2001-01-01, Point(1 1 1)@2001-01-02]',
  0.5, 0.75));
-- [POINT Z (0.5 1.5 3)@2001-01-01, POINT Z (0.5 0.75 1)@2001-01-02]
SELECT asEWKT(scale(tgeompoint '[Point(1 2 3)@2001-01-01, Point(1 1 1)@2001-01-02]',
```

```
  geometry 'Point(0.5 0.75 0.8)'));
-- [POINT Z (0.5 1.5 2.4)@2001-01-01, POINT Z (0.5 0.75 0.8)@2001-01-02]
SELECT asEWKT(scale(tgeometry '[Point(1 1)@2001-01-01, Linestring(1 1,2 2)@2001-01-02]',
  geometry 'Point(2 2)', geometry 'Point(1 1)'));
-- [POINT(1 1)@2001-01-01, LINESTRING(1 1,3 3)@2001-01-02]
```

- Transform a temporal geometric point into the coordinate space of a Mapbox Vector Tile

  `asMVTGeom(tpoint,bounds,extent=4096,buffer=256,clip=true)` → `(geom,times)`

  The result is a couple composed of a `geometry` value and an array of associated timestamp values encoded as Unix epoch. 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)@2001-01-01, Point(100 100)@2001-01-02]',
  stbox 'STBOX X((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)@2001-01-01, Point(100 100)@2001-01-02]',
  stbox 'STBOX X((40,40),(60,60))', clip:=false) AS mvt ) AS t;
-- LINESTRING(-8192 12288,12288 -8192) | {946681200,946767600}
```

- Extract from a temporal point with linear interpolation the subsequences where the point stays within an area with a specified maximum size for at least the given duration

  `stops(tpoint,maxDist=0.0,minDuration='0 minutes')` → `tpoint`

  The size of the area is computed as the diagonal of the minimum rotated rectangle of the points in the subsequence. If `maxDist` is not given it is assumed 0.0 and thus, the function extracts the constant segments of the given temporal point. The distance is computed in the units of the coordinate system. Note that even though the function accepts 3D geometries, the computation is always performed in 2D.

```
SELECT asText(stops(tgeompoint '[Point(1 1)@2001-01-01, Point(1 1)@2001-01-02,
  Point(2 2)@2001-01-03, Point(2 2)@2001-01-04]'));
/* {[POINT(1 1)@2001-01-01, POINT(1 1)@2001-01-02), [POINT(2 2)@2001-01-03,
   POINT(2 2)@2001-01-04]} */
SELECT asText(stops(tgeompoint '[Point(1 1 1)@2001-01-01, Point(1 1 1)@2001-01-02,
  Point(2 2 2)@2001-01-03, Point(2 2 2)@2001-01-04]', 1.75));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (1 1 1)@2001-01-02, POINT Z (2 2 2)@2001-01-03,
   POINT Z (2 2 2)@2001-01-04]} */
```

Chapter 8

Temporal Geometry Types (Part 2)

8.1 Restrictions

- Restrict to (the complement of) a geometry, a Z span, and/or a period

  `atGeometry(tgeom,geometry[,zspan]) → tgeom`

  `minusGeometry(tgeom,geometry[,zspan]) → tgeom`

  The geometry must be in 2D and the computation with respect to it is done in 2D. The result preserves the Z dimension of the temporal point, if it exists.

```
SELECT asText(atGeometry(tgeompoint '[Point(0 0)@2001-01-01, Point(3 3)@2001-01-04)',
  geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
-- {"[POINT(1 1)@2001-01-02, POINT(2 2)@2001-01-03]"}
SELECT astext(atGeometry(tgeompoint '[Point(0 0 0)@2001-01-01, Point(4 4 4)@2001-01-05]',
  geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
-- {[POINT Z (1 1 1)@2001-01-02, POINT Z (2 2 2)@2001-01-03]}
SELECT asText(atGeometry(tgeompoint '[Point(1 1 1)@2001-01-01, Point(3 1 1)@2001-01-03,
  Point(3 1 3)@2001-01-05]', 'Polygon((2 0,2 2,2 4,4 0,2 0))', '[0,2]'));
-- {[POINT Z (2 1 1)@2001-01-02, POINT Z (3 1 1)@2001-01-03, POINT Z (3 1 2)@2001-01-04]}
SELECT asText(atGeometry(tgeometry 'Linestring(1 1,10 1)@2001-01-01',
  'Polygon((0 0,0 5,5 5,5 0,0 0))'));
-- LINESTRING(1 1,5 1)@2001-01-01
```

```
SELECT asText(minusGeometry(tgeompoint '[Point(0 0)@2001-01-01, Point(3 3)@2001-01-04)',
  geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
/* {[POINT(0 0)@2001-01-01, POINT(1 1)@2001-01-02), (POINT(2 2)@2001-01-03,
   POINT(3 3)@2001-01-04)} */
SELECT astext(minusGeometry(tgeompoint '[Point(0 0 0)@2001-01-01,
  Point(4 4 4)@2001-01-05]', geometry 'Polygon((1 1,1 2,2 2,2 1,1 1))'));
/* {[POINT Z (0 0 0)@2001-01-01, POINT Z (1 1 1)@2001-01-02),
   (POINT Z (2 2 2)@2001-01-03, POINT Z (4 4 4)@2001-01-05]} */
SELECT asText(minusGeometry(tgeompoint '[Point(1 1 1)@2001-01-01, Point(3 1 1)@2001-01-03,
  Point(3 1 3)@2001-01-05]', 'Polygon((2 0,2 2,2 4,4 0,2 0))', '[0,2]'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 1 1)@2001-01-02),
    (POINT Z (3 1 2)@2001-01-04, POINT Z (3 1 3)@2001-01-05]} */
SELECT asText(minusGeometry(tgeometry 'Linestring(1 1,10 1)@2001-01-01',
  'Polygon((0 0,0 5,5 5,5 0,0 0))'));
-- LINESTRING(5 1,10 1)@2001-01-01
```

- Restrict to (the complement of) an `stbox`

  `atStbox(tgeom,stbox,borderInc bool=true) → tgeompoint`

```
minusStbox(tgeom,stbox,borderInc bool=true) → tgeompoint
```

The third optional argument is used for multidimensional tiling (see Section 9.5) to exclude the upper border of the tiles when a temporal value is split in multiple tiles, so that all fragments of the temporal geometry are exclusive.

```
SELECT asText(atStbox(tgeompoint '[Point(0 0)@2001-01-01, Point(3 3)@2001-01-04)',
  stbox 'STBOX XT(((0,0),(2,2)),[2001-01-02, 2001-01-04])'));
-- {[POINT(1 1)@2001-01-02, POINT(2 2)@2001-01-03]}
SELECT asText(atStbox(tgeompoint '[Point(1 1 1)@2001-01-01, Point(3 3 3)@2001-01-03,
  Point(3 3 2)@2001-01-04, Point(3 3 7)@2001-01-09]', stbox 'STBOX Z((2,2,2),(3,3,3))'));
/* {[POINT Z (2 2 2)@2001-01-02, POINT Z (3 3 3)@2001-01-03, POINT Z (3 3 2)@2001-01-04,
  POINT Z (3 3 3)@2001-01-05]} */
SELECT asText(atStbox(tgeometry '[Point(1 1)@2001-01-01, Linestring(1 1,3 3)@2001-01-03,
  Point(2 2)@2001-01-04, Linestring(3 3,4 4)@2001-01-09]', stbox 'STBOX X((2,2),(3,3))'));
-- {[LINESTRING(2 2,3 3)@2001-01-03, POINT(2 2)@2001-01-04, POINT(3 3)@2001-01-09]}
```

```
SELECT asText(minusStbox(tgeompoint '[Point(1 1)@2001-01-01, Point(4 4)@2001-01-04)',
  stbox 'STBOX XT(((1,1),(2,2)),[2001-01-03,2001-01-04])'));
-- {(POINT(2 2)@2001-01-02, POINT(3 3)@2001-01-03)}
SELECT asText(minusStbox(tgeompoint '[Point(1 1 1)@2001-01-01, Point(3 3 3)@2001-01-03,
  Point(3 3 2)@2001-01-04, Point(3 3 7)@2001-01-09]', stbox 'STBOX Z((2,2,2),(3,3,3))'));
/* {[POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02),
    (POINT Z (3 3 3)@2001-01-05, POINT Z (3 3 7)@2001-01-09]} */
SELECT asText(minusStbox(tgeometry '[Point(1 1)@2001-01-01,
  Linestring(1 1,3 3)@2001-01-03, Point(2 2)@2001-01-04,
  Linestring(1 1,4 4)@2001-01-09]', stbox 'STBOX X((2,2),(3,3))'));
/* {[POINT(1 1)@2001-01-01, LINESTRING(1 1,2 2)@2001-01-03,
   LINESTRING(1 1,2 2)@2001-01-04),[MULTILINESTRING((1 1,2 2),(3 3,4 4))@2001-01-09]} */
```

8.2 Spatial Reference System

- Return or set the spatial reference identifier

```
SRID(tspatial) → integer
setSRID(tspatial) → tspatial
```

```
SELECT SRID(tgeompoint 'Point(0 0)@2001-01-01');
-- 0
SELECT asEWKT(setSRID(tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-02)', 4326));
-- SRID=4326;[POINT(0 0)@2001-01-01, POINT(1 1)@2001-01-02)
```

- Transform to a spatial reference identifier

```
transform(tspatial,integer) → tspatial
transformPipeline(tspatial,pipeline text,to_srid integer,is_forward bool=true) → tspatia
```

The `transform` function specifies the transformation with a target SRID. An error is raised when the input temporal point has an unknown SRID (represented by 0).

The `transformPipeline` function specifies the transformation with a defined coordinate transformation pipeline represented with the following string format:

```
urn:ogc:def:coordinateOperation:AUTHORITY::CODE
```

The SRID of the input temporal point is ignored, and the SRID of the output temporal point will be set to zero unless a value is provided via the optional `to_srid` parameter. As stated by the last parameter, the pipeline is executed by default in a forward direction; by setting the parameter to false, the pipeline is executed in the inverse direction.

```
SELECT asEWKT(transform(tgeompoint 'SRID=4326;Point(4.35 50.85)@2001-01-01', 3812));
-- SRID=3812;POINT(648679.018035303 671067.055638114)@2001-01-01
```

```
WITH test(tgeo, pipeline) AS (
  SELECT tgeogpoint 'SRID=4326;{Point(4.3525 50.846667 100.0)@2001-01-01,
    Point(-0.1275 51.507222 100.0)@2001-01-02}',
    text 'urn:ogc:def:coordinateOperation:EPSG::16031' )
SELECT asEWKT(transformPipeline(transformPipeline(tgeo, pipeline, 4326),
  pipeline, 4326, false), 6)
FROM test;
/* SRID=4326;{POINT Z (4.3525 50.846667 100)@2001-01-01,
   POINT Z (-0.1275 51.507222 100)@2001-01-02} */
```

8.3 Bounding Box Operations

- Return the spatiotemporal bounding box expanded in the spatial dimension by a float value

  `expandSpace({spatial,tspatial},float) → stbox`

```
SELECT expandSpace(geography 'Linestring(0 0,1 1)', 2);
-- SRID=4326;GEODSTBOX X((-2,-2),(3,3))
SELECT expandSpace(tgeompoint 'Point(0 0)@2001-01-01', 2);
-- STBOX XT(((-2,-2),(2,2)),[2001-01-01,2001-01-01])
```

8.4 Distance Operations

- Return the smallest distance ever

  `{geo,tgeo} |=| {geo,tgeo} → float`

```
SELECT tgeompoint '[Point(0 0)@2001-01-02, Point(1 1)@2001-01-04, Point(0 0)@2001-01-06)'
  |=| geometry 'Linestring(2 2,2 1,3 1)';
-- 1
SELECT tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-03, Point(0 0)@2001-01-05)'
  |=| tgeompoint '[Point(2 0)@2001-01-02, Point(1 1)@2001-01-04, Point(2 2)@2001-01-06)';
-- 0.5
SELECT tgeompoint '[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-03,
  Point(0 0 0)@2001-01-05)' |=| tgeompoint '[Point(2 0 0)@2001-01-02,
  Point(1 1 1)@2001-01-04, Point(2 2 2)@2001-01-06)';
-- 0.5
SELECT tgeometry '(Point(1 1)@2001-01-01, Linestring(3 1,1 1)@2001-01-03]' |=|
  geometry 'Linestring(1 3,2 2,3 3)';
-- 1
```

The operator `|=|` can be used for doing nearest neighbor searches using a GiST or an SP-GiST index (see Section 10.2). 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)@2001-01-01, Point(1 1 1)@2001-01-03,
    Point(0 0 0)@2001-01-05)'::geometry,
  tgeompoint '[Point(2 0 0)@2001-01-02, Point(1 1 1)@2001-01-04,
    Point(2 2 2)@2001-01-06)'::geometry);
-- 0.5
```

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

  `nearestApproachInstant({geo,tgeo},{geo,tgeo}) → tgeo`

  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)@2001-01-01,
  Point(3 1)@2001-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
-- POINT(2 1)@2001-01-02
SELECT asText(nearestApproachInstant(tgeompoint 'Interp=Step;(Point(1 1)@2001-01-01,
  Point(3 1)@2001-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
-- POINT(1 1)@2001-01-01
SELECT asText(nearestApproachInstant(tgeompoint '(Point(1 1)@2001-01-01,
  Point(2 2)@2001-01-03]', tgeompoint '(Point(1 1)@2001-01-01, Point(4 1)@2001-01-03]'));
-- POINT(1 1)@2001-01-01
SELECT asText(nearestApproachInstant(tgeometry
  '[Linestring(0 0 0,1 1 1)@2001-01-01, Point(0 0 0)@2001-01-03]', tgeometry
  '[Point(2 0 0)@2001-01-02, Point(1 1 1)@2001-01-04, Point(2 2 2)@2001-01-06]'));
-- LINESTRING Z (0 0 0,1 1 1)@2001-01-02
```

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)@2001-01-01, Point(1 1 1)@2001-01-03,
    Point(0 0 0)@2001-01-05)'::geometry,
  tgeompoint '[Point(2 0 0)@2001-01-02, Point(1 1 1)@2001-01-04,
    Point(2 2 2)@2001-01-06)'::geometry));
-- 2001-01-03 12:00:00+00
```

- Return the line connecting the nearest approach point

  `shortestLine({geo,tgeo},{geo,tgeo}) → 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)@2001-01-01,
  Point(3 1)@2001-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
-- LINESTRING(2 1,2 2)
SELECT ST_AsText(shortestLine(tgeompoint 'Interp=Step;(Point(1 1)@2001-01-01,
  Point(3 1)@2001-01-03]', geometry 'Linestring(1 3,2 2,3 3)'));
-- LINESTRING(1 1,2 2)
SELECT ST_AsText(shortestLine(tgeometry
  '[Linestring(0 0 0,1 1 1)@2001-01-01, Point(0 0 0)@2001-01-03]', tgeometry
  '[Point(2 0 0)@2001-01-02, Point(1 1 1)@2001-01-04, Point(2 2 2)@2001-01-06]'));
-- LINESTRING Z (0 0 0,2 0 0)
```

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)@2001-01-01, Point(1 1 1)@2001-01-03,
    Point(0 0 0)@2001-01-05)',
  tgeompoint '[Point(2 0 0)@2001-01-02, Point(1 1 1)@2001-01-04,
    Point(2 2 2)@2001-01-06)')) <= 0.5;
-- true
SELECT ST_CPAWithin(
  tgeompoint '[Point(0 0 0)@2001-01-01, Point(1 1 1)@2001-01-03,
    Point(0 0 0)@2001-01-05)'::geometry,
  tgeompoint '[Point(2 0 0)@2001-01-02, Point(1 1 1)@2001-01-04,
    Point(2 2 2)@2001-01-06)'::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.

- Return the temporal distance

`{geo,tgeo} <-> {geo,tgeo} → tfloat`

```
SELECT tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-03)' <->
  geometry 'Point(0 1)';
-- [1@2001-01-01, 0.707106781186548@2001-01-02, 1@2001-01-03)
SELECT tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-03)' <->
  tgeompoint '[Point(0 1)@2001-01-01, Point(1 0)@2001-01-03)';
-- [1@2001-01-01, 0@2001-01-02, 1@2001-01-03)
SELECT tgeompoint '[Point(0 1)@2001-01-01, Point(0 0)@2001-01-03)' <->
  tgeompoint '[Point(0 0)@2001-01-01, Point(1 0)@2001-01-03)';
-- [1@2001-01-01, 0.707106781186548@2001-01-02, 1@2001-01-03)
SELECT tgeometry '[Point(0 0)@2001-01-01, Linestring(0 0,1 1)@2001-01-02]' <->
  tgeometry '[Point(0 1)@2001-01-01, Point(1 0)@2001-01-02)';
-- Interp=Step;[1@2001-01-01, 1@2001-01-02]
```

8.5 Spatial Relationships

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

There are three versions of the relationships:

- The *ever* relationships determine whether the topological or distance relationship is ever satisfied (see Section 5.4.2) and returns a `boolean`. Examples are the `eIntersects` and `eDwithin` functions.
- The *always* relationships determine whether the topological or distance relationship is always satisfied (see Section 5.4.2) and returns a `boolean`. Examples are the `aIntersects` and `aDwithin` 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 eIntersects(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  tgeompoint '[Point(0 2)@2001-01-01, Point(4 2)@2001-01-05)');
-- 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)@2001-01-01, Point(4 2)@2001-01-05)');
-- {[f@2001-01-01, t@2001-01-02, t@2001-01-04], (f@2001-01-04, f@2001-01-05)}
```

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

```
SELECT eDwithin(tgeompoint '[Point(3 1)@2001-01-01, Point(5 1)@2001-01-03)',
  tgeompoint '[Point(3 1)@2001-01-01, Point(1 1)@2001-01-03)', 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)@2001-01-01, Point(5 1)@2001-01-03)',
  tgeompoint '[Point(3 1)@2001-01-01, Point(1 1)@2001-01-03)', 2);
-- {[t@2001-01-01, t@2001-01-02], (f@2001-01-02, f@2001-01-03)}
```

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

The ever or always 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 eIntersects(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 `eIntersects` 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 been generalized for temporal geometries, only those derived from the following functions: `ST_Contains`, `ST_Covers`, `ST_Disjoint`, `ST_Intersects`, `ST_Touches`, and `ST_DWithin`. These functions only suport 2D geometries and only the functions `ST_Covers`, `ST_Intersects`, and `ST_DWithin` suport geographies. Consequently, the same applies for the MobilityDB functions derived from them, excepted that they support 3D for temporal points, that is, `tgeompoint`, and `tgeogpoint`. As stated above, each of the above PostGIS functions, such as `ST_Contains`, has three generalized versions in MobilityDB, namely `eContains`, `aContains`, and `tContains`. Furthermore, not all combinations of parameters are meaningful for the generalized functions. For example, `tContains(tpoint, geometry)` is meaningful only when the geometry is a single point, and `tContains(tpoint, tpoint)` is equivalent to `tintersects(tpoint, geometry)`.

8.5.1 Ever and Always Relationships

We present next the ever and always relationships. These relationships automatically include a bounding box comparison that makes use of any spatial indexes that are available on the arguments.

- Ever or always contains

  `eContains({geometry,tgeom},{geometry,tgeom}) → boolean`

  `aContains({geometry,tgeom},{geometry,tgeom}) → boolean`

  This function returns true if the temporal geometry and the geometry ever or always intersect at their interior. Recall that a geometry does not contain things in its boundary and thus, polygons and lines do not contain lines and points lying in their boundary. Please refer to the documentation of the ST_Contains function in PostGIS.

```
SELECT eContains(geometry 'Linestring(1 1,3 3)',
  tgeompoint '[Point(4 2)@2001-01-01, Point(2 4)@2001-01-02]');
-- false
SELECT eContains(geometry 'Linestring(1 1,3 3,1 1)',
  tgeompoint '[Point(4 2)@2001-01-01, Point(2 4)@2001-01-03]');
-- true
SELECT eContains(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
```

```
 tgeompoint '[Point(0 1)@2001-01-01, Point(4 1)@2001-01-02]');
-- false
SELECT eContains(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  tgeometry '[Linestring(1 1,4 4)@2001-01-01, Point(3 3)@2001-01-04]');
-- true
```

- Ever or always covers

  `eCovers({geometry,tgeom},{geometry,tgeom}) → boolean`

  `aCovers({geometry,tgeom},{geometry,tgeom}) → boolean`

  Please refer to the documentation of the ST_Contains and the ST_Covers function in PostGIS for detailed explanations about the difference between the two functions.

```
SELECT eCovers(geometry 'Linestring(1 1,3 3)',
  tgeompoint '[Point(4 2)@2001-01-01, Point(2 4)@2001-01-02]');
-- false
SELECT eCovers(geometry 'Linestring(1 1,3 3,1 1)',
  tgeompoint '[Point(4 2)@2001-01-01, Point(2 4)@2001-01-03]');
-- true
SELECT eCovers(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
 tgeompoint '[Point(0 1)@2001-01-01, Point(4 1)@2001-01-02]');
-- false
SELECT eCovers(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  tgeometry '[Linestring(1 1,4 4)@2001-01-01, Point(3 3)@2001-01-04]');
-- true
```

- Is ever or always disjoint

  `eDisjoint({geo,tgeo},{geo,tgeo}) → boolean`

  `aDisjoint({geo,tgeo},{geo,tgeo}) → boolean`

```
SELECT eDisjoint(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
  tgeompoint '[Point(0 0)@2001-01-01, Point(1 1)@2001-01-03)');
-- false
SELECT eDisjoint(geometry 'Polygon((0 0 0,0 1 1,1 1 1,1 0 0,0 0 0))',
  tgeometry '[Linestring(1 1 1,2 2 2)@2001-01-01, Point(2 2 2)@2001-01-03]');
-- true
```

- Is ever or always at distance within

  `eDwithin({geo,tgeo},{geo,tgeo},float) → boolean`

  `aDwithin({geometry,tgeom},{geometry,tgeom},float) → boolean`

```
SELECT eDwithin(geometry 'Point(1 1 1)',
  tgeompoint '[Point(0 0 0)@2001-01-01, Point(1 1 0)@2001-01-02]', 1);
-- true
SELECT eDwithin(geometry 'Polygon((0 0 0,0 1 1,1 1 1,1 0 0,0 0 0))',
  tgeompoint '[Point(0 2 2)@2001-01-01,Point(2 2 2)@2001-01-02]', 1);
-- false
```

- Ever or always intersects

  `eIntersects({geo,tgeo},{geo,tgeo}) → boolean`

  `aIntersects({geometry,tgeom},{geometry,tgeom}) → boolean`

```
SELECT eIntersects(geometry 'Polygon((0 0 0,0 1 0,1 1 0,1 0 0,0 0 0))',
  tgeompoint '[Point(0 0 1)@2001-01-01, Point(1 1 1)@2001-01-03)');
-- false
SELECT eIntersects(geometry 'Polygon((0 0 0,0 1 1,1 1 1,1 0 0,0 0 0))',
  tgeompoint '[Point(0 0 1)@2001-01-01, Point(1 1 1)@2001-01-03)');
-- true
```

- Ever or always touches

  `eTouches({geometry,tgeom},{geometry,tgeom}) → boolean`

  `aTouches({geometry,tgeom},{geometry,tgeom}) → boolean`

```
SELECT eTouches(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))',
  tgeompoint '[Point(0 0)@2001-01-01, Point(0 1)@2001-01-03)');
-- true
```

8.5.2 Spatiotemporal Relationships

We present next the spatiotemporal relationships. A common requirement regarding them is to restrict the result of the relationship to the instants when the value of the result is true or false. As an example, the following query computes for each trip the time spent traveling in the Brussels municipality.

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

To simplify query writing, the spatiotemporal relationships have an optional last parameter, which if given applies the `atValue` function (see Section 5.2) 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,tgeom,atValue boolean=NULL) → tbool`

```
SELECT tContains(geometry 'Linestring(1 1,3 3)',
  tgeompoint '[Point(4 2)@2001-01-01, Point(2 4)@2001-01-02]');
-- {[f@2001-01-01, f@2001-01-02]}
SELECT tContains(geometry 'Linestring(1 1,3 3,1 1)',
  tgeompoint '[Point(4 2)@2001-01-01, Point(2 4)@2001-01-03]');
-- {[f@2001-01-01, t@2001-01-02], (f@2001-01-02, f@2001-01-03]}
SELECT tContains(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  tgeompoint '[Point(0 1)@2001-01-01, Point(4 1)@2001-01-02]');
-- {[f@2001-01-01, f@2001-01-02]}
SELECT tContains(geometry 'Polygon((1 1,1 3,3 3,3 1,1 1))',
  tgeompoint '[Point(1 4)@2001-01-01, Point(4 1)@2001-01-04]');
-- {[f@2001-01-01, f@2001-01-02], (t@2001-01-02, f@2001-01-03, f@2001-01-04]}
```

- Temporal disjoint

  `tDisjoint({geo,tgeo},{geo,tgeo},atValue boolean=NULL) → 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)@2001-01-01, Point(3 3)@2001-01-04)');
-- {[t@2001-01-01, f@2001-01-02, f@2001-01-03], (t@2001-01-03, t@2001-01-04]}
SELECT tDisjoint(tgeompoint '[Point(0 3)@2001-01-01, Point(3 0)@2001-01-05)',
  tgeompoint '[Point(0 0)@2001-01-01, Point(3 3)@2001-01-05)');
-- {[t@2001-01-01, f@2001-01-03], (t@2001-01-03, t@2001-01-05)}
```

- Temporal distance within

  `tDwithin({geo,tgeo},{geo,tgeo},float,atValue boolean=NULL) → tbool`

```
SELECT tDwithin(geometry 'Point(1 1)',
  tgeompoint '[Point(0 0)@2001-01-01, Point(2 2)@2001-01-03)', sqrt(2));
--  {[t@2001-01-01, t@2001-01-03)}
SELECT tDwithin(tgeompoint '[Point(1 0)@2001-01-01, Point(1 4)@2001-01-05]',
  tgeompoint 'Interp=Step;[Point(1 2)@2001-01-01, Point(1 3)@2001-01-05]', 1);
-- {[f@2001-01-01, t@2001-01-02, t@2001-01-04], (f@2001-01-04, t@2001-01-05]}
```

- Temporal intersects

  `tIntersects({geo,tgeo},{geo,tgeo},atValue boolean=NULL) → tbool`

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

```
SELECT tIntersects(geometry 'MultiPoint(1 1,2 2)',
  tgeompoint '[Point(0 0)@2001-01-01, Point(3 3)@2001-01-04)');
/* {[f@2001-01-01, t@2001-01-02], (f@2001-01-02, t@2001-01-03],
   (f@2001-01-03, f@2001-01-04]} */
SELECT tIntersects(tgeompoint '[Point(0 3)@2001-01-01, Point(3 0)@2001-01-05)',
  tgeompoint '[Point(0 0)@2001-01-01, Point(3 3)@2001-01-05)');
-- {[f@2001-01-01, t@2001-01-03], (f@2001-01-03, f@2001-01-05)}
```

- Temporal touches

  `tTouches({geometry,tgeom},{geometry,tgeom},atValue boolean=NULL) → tbool`

```
SELECT tTouches(geometry 'Polygon((1 0,1 2,2 2,2 0,1 0))',
  tgeompoint '[Point(0 0)@2001-01-01, Point(3 0)@2001-01-04)');
-- {[f@2001-01-01, t@2001-01-02, t@2001-01-03], (f@2001-01-03, f@2001-01-04]}
```

Chapter 9

Temporal Types: Analytics Operations

9.1 Simplification

- Return a temporal float or a temporal point simplified ensuring that consecutive values are at least a certain distance or time interval apart

  `minDistSimplify({tfloat,tpoint},mindist float) → {tfloat,tpoint}`

  `minTimeDeltaSimplify({tfloat,tpoint},mint interval) → {tfloat,tpoint}`

  In the case of temporal points, the distance is specified in the units of the coordinate system. Notice that simplification applies only to temporal sequences or sequence sets with linear interpolation. In all other cases, a copy of the given temporal value is returned.

```
SELECT minDistSimplify(tfloat '[1@2001-01-01,2@2001-01-02,3@2001-01-04,4@2001-01-05]', 1);
-- [1@2001-01-01, 3@2001-01-04, 4@2001-01-05]
SELECT asText(minDistSimplify(tgeompoint '[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02, Point(3 3 3)@2001-01-04, Point(5 5 5)@2001-01-05)', sqrt(3)));
-- [POINT Z (1 1 1)@2001-01-01, POINT Z (3 3 3)@2001-01-04, POINT Z (5 5 5)@2001-01-05)
SELECT asText(minDistSimplify(tgeompoint '[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02, Point(3 3 3)@2001-01-04, Point(4 4 4)@2001-01-05)', sqrt(3)));
-- [POINT Z (1 1 1)@2001-01-01, POINT Z (3 3 3)@2001-01-04, POINT Z (4 4 4)@2001-01-05]
```

```
SELECT minTimeDeltaSimplify(tfloat '[1@2001-01-01, 2@2001-01-02, 3@2001-01-04,
  4@2001-01-05]', '1 day');
-- [1@2001-01-01, 3@2001-01-04, 4@2001-01-05]
SELECT asText(minTimeDeltaSimplify(tgeogpoint '[Point(1 1 1)@2001-01-01,
  Point(2 2 2)@2001-01-02, Point(3 3 3)@2001-01-04, Point(5 5 5)@2001-01-05)', '1 day'));
-- [POINT Z (1 1 1)@2001-01-01, POINT Z (3 3 3)@2001-01-04, POINT Z (5 5 5)@2001-01-05]
```

- Return a temporal float or a temporal point simplified using the Douglas-Peucker algorithm

  `maxDistSimplify({tfloat,tgeompoint},maxdist float,syncdist=true) →`
  `{tfloat,tgeompoint}`

  `douglasPeuckerSimplify({tfloat,tgeompoint},maxdist float,syncdist=true) →`
  `{tfloat,tgeompoint}`

  The difference between the two functions is that `maxDistSimplify` uses a single-pass version of the algorithm whereas `douglasPeuckerSimplify` uses the standard recursive algorithm.

  The function removes values or points that are less than or equal to the distance passed as second argument. In the case of temporal points, the distance is specified in the units of the coordinate system. The third argument applies only for temporal points and specifies whether the spatial or the synchronized distance is used. Notice that simplification applies only to temporal sequences or sequence sets with linear interpolation. In all other cases, a copy of the given temporal value is returned.

```
-- Only synchronous distance for temporal floats
SELECT maxDistSimplify(tfloat '[1@2001-01-01, 2@2001-01-02, 1@2001-01-03, 3@2001-01-04,
  1@2001-01-05]', 1, false);
-- [1@2001-01-01, 1@2001-01-03, 3@2001-01-04, 1@2001-01-05]
-- Synchronous distance by default for temporal points
SELECT asText(maxDistSimplify(tgeompoint '[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02,
  Point(3 1)@2001-01-03, Point(3 3)@2001-01-05, Point(5 1)@2001-01-06]', 2));
-- [POINT(1 1)@2001-01-01, POINT(3 3)@2001-01-05, POINT(5 1)@2001-01-06]
-- Spatial distance
SELECT asText(maxDistSimplify(tgeompoint '[Point(1 1)@2001-01-01, Point(2 2)@2001-01-02,
  Point(3 1)@2001-01-03, Point(3 3)@2001-01-05, Point(5 1)@2001-01-06]', 2, false));
-- [POINT(1 1)@2001-01-01, POINT(5 1)@2001-01-06]
```

```
-- Spatial vs synchronized Euclidean distance
SELECT asText(douglasPeuckerSimplify(tgeompoint '[Point(1 1)@2001-01-01,
  Point(6 1)@2001-01-06, Point(7 4)@2001-01-07]', 2.3, false));
-- [POINT(1 1)@2001-01-01, POINT(7 4)@2001-01-07]
SELECT asText(douglasPeuckerSimplify(tgeompoint '[Point(1 1)@2001-01-01,
  Point(6 1)@2001-01-06, Point(7 4)@2001-01-07]', 2.3, true));
-- [POINT(1 1)@2001-01-01, POINT(6 1)@2001-01-06, POINT(7 4)@2001-01-07]
```

The difference between the spatial and the synchronized distance is illustrated in the last two examples above and in Figure 9.1. In the first example, which uses the spatial distance, the second instant is removed since the perpendicular distance between `POINT(6 1)` and the line defined by `POINT(1 1)` and `POINT(7 4)` is equal to 2.23. On the contrary, in the second example the second instant is kept since the projection of `Point(6 2)` at timestamp `2001-01-06` over the temporal line segment results in `Point(6 3.5)` and the distance between the original point and its projection is 2.5.

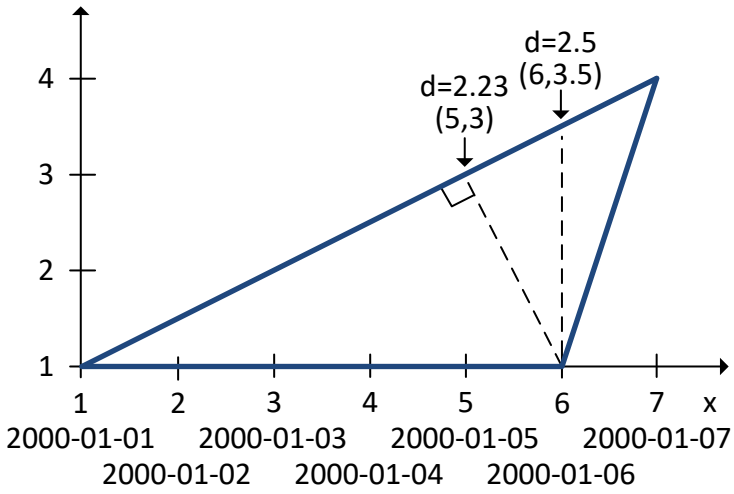


Figure 9.1: Difference between the spatial and the synchronous distance.

A typical use for the `douglasPeuckerSimplify` function is to reduce the size of a dataset, in particular for visualization purposes. If the visualization is static, then the spatial distance should be preferred, if the visualization is dynamic or animated, the synchronized distance should be preferred.

9.2 Reduction

- Sample a temporal value with respect to an interval

  `tsample(temporal,duration interval,torigin timestamptz='2000-01-03',`
  `interp='discrete') → temporal`

If the origin is not specified, it is set by default to Monday, January 3, 2000. The given interval must be strictly greater than zero.

```
SELECT tsample(tbool '[true@2001-01-01,false@2001-01-02]', '12 hours', '2001-01-01', 'step ↩
    ');
-- [t@2001-01-01, f@2001-01-02]
SELECT tsample(tint '{1@2001-01-01,5@2001-01-05}', '3 days', '2001-01-01');
-- {1@2001-01-01}
SELECT tsample(tfloat '[1@2001-01-01,5@2001-01-05]', '1 day', '2001-01-01');
-- {1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 4@2001-01-04, 5@2001-01-05}
SELECT tsample(tfloat '[1@2001-01-01,5@2001-01-05]', '3 days', '2001-01-01');
-- {1@2001-01-01, 4@2001-01-04}
SELECT tsample(tfloat '[1@2001-01-01,5@2001-01-05]', '3 days', '2001-01-01', 'linear');
-- [1@2001-01-01, 4@2001-01-04]
```

```
SELECT asText(tsample(tgeompoint '[Point(1 1)@2001-01-01, Point(5 5)@2001-01-05]',
  '2 days', '2001-01-01'));
-- {POINT(1 1)@2001-01-01, POINT(3 3)@2001-01-03, POINT(5 5)@2001-01-05}
SELECT asText(tsample(tgeompoint '{[Point(1 1)@2001-01-01, Point(5 5)@2001-01-05],
  [Point(1 1)@2001-01-06, Point(5 5)@2001-01-08]}', '3 days', '2001-01-01'));
-- {POINT(1 1)@2001-01-01, POINT(4 4)@2001-01-04, POINT(3 3)@2001-01-07}
SELECT asText(tsample(tgeompoint '{[Point(1 1)@2001-01-01, Point(5 5)@2001-01-05],
  [Point(1 1)@2001-01-06, Point(5 5)@2001-01-08]}', '3 days', '2001-01-01', 'step'));
-- Interp=Step;[POINT(1 1)@2001-01-01, POINT(4 4)@2001-01-04, POINT(3 3)@2001-01-07]
```

Figure 9.2 illustrates the sampling of temporal floats with various interpolations. As shown in the figure, the sampling operation is best suited for temporal values with continuous interpolation.

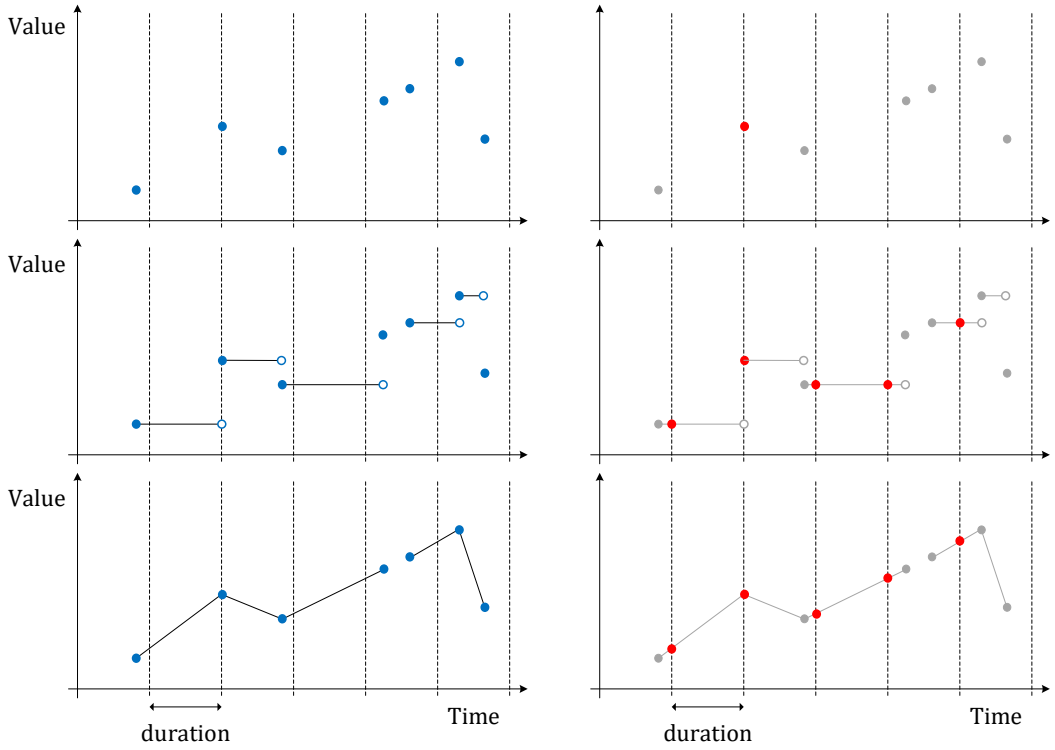


Figure 9.2: Sampling of temporal floats with discrete, step, and linear interpolation.

- Reduce the temporal precision of a temporal value with respect to an interval computing the time-weighted average/centroid in each time bin

  `tprecision({tnumber,tgeompoint},duration interval,torigin timestamptz='2000-01-03')`

→ {tnumber,tpoint}

If the origin is not specified, it is set by default to Monday, January 3, 2000. The given interval must be strictly greater than zero.

```
SELECT tprecision(tint '[1@2001-01-01,5@2001-01-05,1@2001-01-09]','1 day',
  '2001-01-01');
-- Interp=Step;[1@2001-01-01, 5@2001-01-05, 1@2001-01-09]
SELECT tprecision(tfloat '[1@2001-01-01,5@2001-01-05,1@2001-01-09)','1 day',
  '2001-01-01');
-- [1.5@2001-01-01, 4.5@2001-01-04, 4.5@2001-01-05, 1.5@2001-01-08]
SELECT tprecision(tfloat '[1@2001-01-01,5@2001-01-05,1@2001-01-09]','1 day',
  '2001-01-01');
-- [1.5@2001-01-01, 4.5@2001-01-04, 4.5@2001-01-05, 1.5@2001-01-08, 1@2001-01-09]
SELECT tprecision(tfloat '[1@2001-01-01,5@2001-01-05,1@2001-01-09)','2 days',
  '2001-01-01');
-- [2@2001-01-01, 4@2001-01-03, 4@2001-01-05, 2@2001-01-07]
```

```
SELECT asText(tprecision(tgeompoint '[Point(1 1)@2001-01-01, Point(5 5)@2001-01-05,
  Point(1 1)@2001-01-09)', '1 day', '2001-01-01'));
/* [POINT(1.5 1.5)@2001-01-01, POINT(4.5 4.5)@2001-01-04, POINT(4.5 4.5)@2001-01-05,
   POINT(1.5 1.5)@2001-01-08] */
SELECT asText(tprecision(tgeompoint '[Point(1 1)@2001-01-01, Point(5 5)@2001-01-05,
  Point(1 1)@2001-01-09)', '2 days', '2001-01-01'));
/* [POINT(2 2)@2001-01-01, POINT(4 4)@2001-01-03, POINT(4 4)@2001-01-05,
   POINT(2 2)@2001-01-07] */
SELECT asText(tprecision(tgeompoint '[Point(1 1)@2001-01-01, Point(5 5)@2001-01-05,
  Point(1 1)@2001-01-09)', '4 days', '2001-01-01'));
-- [POINT(3 3)@2001-01-01, POINT(3 3)@2001-01-05]
```

Changing the precision of a temporal value is akin to changing its *temporal granularity*, for example, from timestamps to hours or days, although the precision can be set to an arbitrary interval, such as 2 hours and 15 minutes. Figure 9.3 illustrates a change of temporal precision for temporal floats with various interpolations.

9.3 Similarity

- Return the discrete Hausdorff distance, the discrete Fréchet distance, or the Dynamic Time Warp (DTW) distance between two temporal values

  `hausdorffDistance({tnumber, tgeo}, {tnumber, tgeo}) → float`

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

  `dynTimeWarpDistance({tnumber, tgeo}, {tnumber, tgeo}) → float`

  These functions have a linear space complexity since only two rows of the distance matrix are allocated in memory. Nevertheless, their time complexity is quadratic in the number of instants of the temporal values. Therefore, the functions require considerable time for temporal values with large number of instants.

```
SELECT hausdorffDistance(tfloat '[1@2001-01-01, 3@2001-01-03, 1@2001-01-06]',
 tfloat '[1@2001-01-01, 1.5@2001-01-02, 2.5@2001-01-03, 1.5@2001-01-04, 1.5@2001-01-05]');
-- 0.5
SELECT round(hausdorffDistance(tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03,
  Point(1 1)@2001-01-05]', tgeompoint '[Point(1.1 1.1)@2001-01-01,
  Point(2.5 2.5)@2001-01-02, Point(4 4)@2001-01-03, Point(3 3)@2001-01-04,
  Point(1.5 2)@2001-01-05]')::numeric, 6);
-- 1.414214
```

```
SELECT frechetDistance(tfloat '[1@2001-01-01, 3@2001-01-03, 1@2001-01-06]',
 tfloat '[1@2001-01-01, 1.5@2001-01-02, 2.5@2001-01-03, 1.5@2001-01-04, 1.5@2001-01-05]');
-- 0.5
```

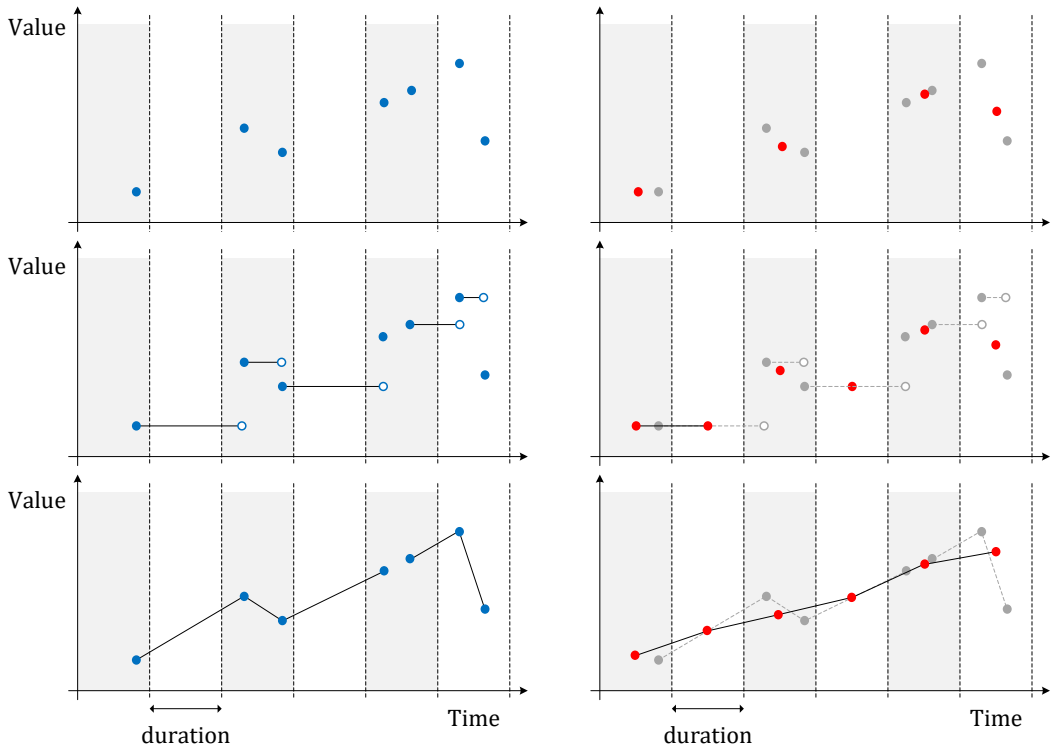


Figure 9.3: Changing the precision of temporal floats with discrete, step, and linear interpolation.

```
SELECT round(frechetDistance(tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03,
  Point(1 1)@2001-01-05]', tgeompoint '[Point(1.1 1.1)@2001-01-01,
  Point(2.5 2.5)@2001-01-02, Point(4 4)@2001-01-03, Point(3 3)@2001-01-04,
  Point(1.5 2)@2001-01-05]')::numeric, 6);
-- 1.414214
```

```
SELECT dynTimeWarpDistance(tfloat '[1@2001-01-01, 3@2001-01-03, 1@2001-01-06]',
 tfloat '[1@2001-01-01, 1.5@2001-01-02, 2.5@2001-01-03, 1.5@2001-01-04, 1.5@2001-01-05]');
-- 2
SELECT round(dynTimeWarpDistance(tgeompoint '[Point(1 1)@2001-01-01,
  Point(3 3)@2001-01-03, Point(1 1)@2001-01-05]',
  tgeompoint '[Point(1.1 1.1)@2001-01-01, Point(2.5 2.5)@2001-01-02,
  Point(4 4)@2001-01-03, Point(3 3)@2001-01-04, Point(1.5 2)@2001-01-05]')::numeric, 6);
-- 3.380776
```

- Return the correspondence pairs between two temporal values with respect to the discrete Fréchet distance or the Dynamic Time Warp Distance {}

  `frechetDistancePath({tnumber, tgeo}, {tnumber, tgeo}) → {(i,j)}`

  `dynTimeWarpPath({tnumber, tgeo}, {tnumber, tgeo}) → {(i,j)}`

  The result is a set of pairs `(i,j)`. 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@2001-01-01, 3@2001-01-03, 1@2001-01-06]',
 tfloat '[1@2001-01-01, 1.5@2001-01-02, 2.5@2001-01-03, 1.5@2001-01-04, 1.5@2001-01-05]');
-- (0,0)
-- (1,0)
-- (2,1)
-- (3,2)
```

```
-- (4,2)
SELECT frechetDistancePath(tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03,
  Point(1 1)@2001-01-05]', tgeompoint '[Point(1.1 1.1)@2001-01-01,
  Point(2.5 2.5)@2001-01-02, Point(4 4)@2001-01-03, Point(3 3)@2001-01-04,
  Point(1.5 2)@2001-01-05]');
-- (0,0)
-- (1,1)
-- (2,1)
-- (3,1)
-- (4,2)
```

```
SELECT dynTimeWarpPath(tfloat '[1@2001-01-01, 3@2001-01-03, 1@2001-01-06]',
 tfloat '[1@2001-01-01, 1.5@2001-01-02, 2.5@2001-01-03, 1.5@2001-01-04, 1.5@2001-01-05]');
-- (0,0)
-- (1,0)
-- (2,1)
-- (3,2)
-- (4,2)
SELECT dynTimeWarpPath(tgeompoint '[Point(1 1)@2001-01-01, Point(3 3)@2001-01-03,
  Point(1 1)@2001-01-05]', tgeompoint '[Point(1.1 1.1)@2001-01-01,
  Point(2.5 2.5)@2001-01-02, Point(4 4)@2001-01-03, Point(3 3)@2001-01-04,
  Point(1.5 2)@2001-01-05]');
-- (0,0)
-- (1,1)
-- (2,1)
-- (3,1)
-- (4,2)
```

9.4 Splitting Operations

When creating indexes for temporal types, what is stored in the index is not the actual value but instead, a bounding box that *represents* the value. In this case, the index will provide a list of candidate values that *may* satisfy the query predicate, and a second step is needed to filter out candidate values by computing the query predicate on the actual values.

However, when the bounding boxes have a large empty space not covered by the actual values, the index will generate many candidate values that do not satisfy the query predicate, which reduces the efficiency of the index. In these situations, it may be better to represent a value not with a *single* bounding box, but instead with *multiple* bounding boxes. This increases considerably the efficiency of the index, provided that the index is able to manage multiple bounding boxes per value. The following functions are used for generating multiple bounding boxes for a single temporal value.

- Return an array of N time spans obtained by merging the instants or segments of a temporal value 

  `splitNSpans(temp, integer) → tstzspan[]`

  The choice between instants or segments depends on whether the interpolation is discrete or continuous. The last argument specifies the number of output spans. If the number of instants or segments is less than or equal to the given number, the resulting array will have one span per instant or segment of the temporal value. Otherwise, the given number of spans will be obtained by merging consecutive instants or segments.

```
SELECT splitNSpans(ttext '{A@2000-01-01, B@2000-01-02, A@2000-01-03, B@2000-01-04,
  A@2000-01-05}', 1);
-- {"[2000-01-01, 2000-01-05]"}
SELECT splitNSpans(tfloat '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 2@2000-01-04,
  1@2000-01-05}', 2);
-- {"[2000-01-01, 2000-01-03]","[2000-01-04, 2000-01-05]"}
SELECT splitNSpans(tgeompoint '[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(1 1)@2000-01-03, Point(2 2)@2000-01-04, Point(1 1)@2000-01-05]', 2);
-- {"[2000-01-01, 2000-01-03]","[2000-01-03, 2000-01-05]"}
SELECT splitNSpans(tgeogpoint '{[Point(1 1 1)@2000-01-01, Point(2 2 2)@2000-01-02],
```

```
  [Point(1 1 1)@2000-01-03, Point(2 2 2)@2000-01-04], [Point(1 1 1)@2000-01-05]}', 2);
-- {"[2000-01-01, 2000-01-04])","[2000-01-05, 2000-01-05])"}
SELECT splitNSpans(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 2@2000-01-04,
  2@2000-01-05}', 6);
/* {"[2000-01-01, 2000-01-01]","[2000-01-02, 2000-01-02]","[2000-01-03, 2000-01-03]",
    "[2000-01-04, 2000-01-04]","[2000-01-05, 2000-01-05]"} */
SELECT splitNSpans(ttext '[A@2000-01-01, B@2000-01-02, A@2000-01-03, B@2000-01-04,
  A@2000-01-05]', 6);
/* {"[2000-01-01, 2000-01-02]","[2000-01-02, 2000-01-03]",
    "[2000-01-03, 2000-01-04]","[2000-01-04, 2000-01-05]"} */
```

- Return an array of time spans obtained by merging N consecutive instants or segments of a temporal value

  `splitEachNSpans(temp, integer) → tstzspan[]`

  The choice between instants or segments depends on whether the interpolation is discrete or continuous. The last argument specifies the number of input instants or segments that are merged to produce an output span. If the number of input elements is less than or equal to the given number, the resulting array will have a single span per sequence. Otherwise, the given number of consecutive instants or segments will be merged into each output span. Notice that, contrary to the splitNSpans function, the number of spans in the result depends on the number of input instants or segments.

```
SELECT splitEachNSpans(ttext '{A@2000-01-01, B@2000-01-02, A@2000-01-03, B@2000-01-04,
  A@2000-01-05}', 1);
/* {"[2000-01-01, 2000-01-01]","[2000-01-02, 2000-01-02]","[2000-01-03, 2000-01-03]",
     "[2000-01-04, 2000-01-04]","[2000-01-05, 2000-01-05]} */
SELECT splitEachNSpans(tfloat '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 2@2000-01-04,
  1@2000-01-05]', 2);
--  {"[2000-01-01, 2000-01-03]","[2000-01-03, 2000-01-05]"}
SELECT splitEachNSpans(tgeompoint '{[Point(1 1 1)@2000-01-01, Point(2 2 2)@2000-01-02],
  [Point(1 1 1)@2000-01-03, Point(2 2 2)@2000-01-04], [Point(1 1 1)@2000-01-05]}', 2);
--  {"[2000-01-01, 2000-01-02]","[2000-01-03, 2000-01-04]","[2000-01-05, 2000-01-05]"}
SELECT splitEachNSpans(tgeogpoint '[Point(1 1 1)@2000-01-01, Point(2 2 2)@2000-01-02,
  Point(1 1 1)@2000-01-03, Point(2 2 2)@2000-01-04, Point(1 1 1)@2000-01-05]', 6);
-- {"[2000-01-01, 2000-01-05])"}
SELECT splitEachNSpans(tgeogpoint '{[Point(1 1 1)@2000-01-01, Point(2 2 2)@2000-01-02],
  [Point(1 1 1)@2000-01-03, Point(2 2 2)@2000-01-04], [Point(1 1 1)@2000-01-05]}', 6);
-- {"[2000-01-01, 2000-01-02]","[2000-01-03, 2000-01-04]","[2000-01-05, 2000-01-05]"}
```

- Return an array of N temporal boxes obtained by merging the instants or segments of a temporal number

  `splitNTboxes(tnumber, integer) → tbox[]`

  The choice between instants or segments depends on whether the interpolation is discrete or continuous. The last argument specifies the number of output boxes. If the number of input instants or segments is less than or equal to the given number, the resulting array will have one box per instant or segment of the temporal number. Otherwise, the specified number of boxes will be obtained by merging consecutive instants or segments.

```
SELECT splitNTboxes(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05}', 1);
-- {"TBOXINT XT([1, 5),[2000-01-01, 2000-01-05])"}
SELECT splitNTboxes(tfloat '{[1@2000-01-01, 2@2000-01-02], [1@2000-01-03, 4@2000-01-04],
  [1@2000-01-05]}', 2);
/* {"TBOXFLOAT XT([1, 4],[2000-01-01, 2000-01-04])",
    "TBOXFLOAT XT([1, 1],[2000-01-05, 2000-01-05])"} */
SELECT splitNTboxes(tfloat '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05]', 3);
/* {"TBOXFLOAT XT([1, 2],[2000-01-01, 2000-01-03])",
    "TBOXFLOAT XT([1, 4],[2000-01-03, 2000-01-04])",
    "TBOXFLOAT XT([1, 4],[2000-01-04, 2000-01-05])"} */
SELECT splitNTboxes(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05}', 6);
/* {"TBOXINT XT([1, 2),[2000-01-01, 2000-01-01])",
    "TBOXINT XT([2, 3),[2000-01-02, 2000-01-02])",
```

```
    "TBOXINT XT([1, 2),[2000-01-03, 2000-01-03])",
    "TBOXINT XT([4, 5),[2000-01-04, 2000-01-04])",
    "TBOXINT XT([1, 2),[2000-01-05, 2000-01-05])"} */
SELECT splitNTboxes(tint '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05]', 6);
/* {"TBOXINT XT([1, 3),[2000-01-01, 2000-01-02])",
    "TBOXINT XT([1, 3),[2000-01-02, 2000-01-03])",
    "TBOXINT XT([1, 5),[2000-01-03, 2000-01-04])",
    "TBOXINT XT([1, 5),[2000-01-04, 2000-01-05])"} */
```

- Return an array of temporal boxes obtained by merging N consecutive instants or segments of a temporal number

  `splitEachNTboxes(tnumber, integer) → tbox[]`

  The choice between instants or segments depends on whether the interpolation is discrete or continuous. The last argument specifies the number of input instants or segments that are merged to produce an output box. If the number of input instants or segments is less than or equal to the given number, the resulting array will have a single box per sequence. Otherwise, the specified number of consecutive instants or segments will be merged into each output box. Notice that, contrary to the splitNTboxes function, the number of boxes in the result depends on the number of input instants or segments.

```
SELECT splitEachNTboxes(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05}', 1);
/* {"TBOXINT XT([1, 2),[2000-01-01, 2000-01-01])",
    "TBOXINT XT([2, 3),[2000-01-02, 2000-01-02])",
    "TBOXINT XT([1, 2),[2000-01-03, 2000-01-03])",
    "TBOXINT XT([4, 5),[2000-01-04, 2000-01-04])"} */
SELECT splitEachNTboxes(tint '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05]', 1);
/* {"TBOXINT XT([1, 3),[2000-01-01, 2000-01-02])",
    "TBOXINT XT([1, 3),[2000-01-02, 2000-01-03])",
    "TBOXINT XT([1, 5),[2000-01-03, 2000-01-04])",
    "TBOXINT XT([1, 5),[2000-01-04, 2000-01-05])"} */
SELECT splitEachNTboxes(tfloat '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05]', 3);
/* {"TBOXFLOAT XT([1, 4],[2000-01-01, 2000-01-04])",
    "TBOXFLOAT XT([1, 4],[2000-01-04, 2000-01-05])"} */
SELECT splitEachNTboxes(tfloat '{[1@2000-01-01, 2@2000-01-02], [1@2000-01-03, 4@2000 ↩
    -01-04],
  [1@2000-01-05]}', 6);
/* {"TBOXFLOAT XT([1, 2],[2000-01-01, 2000-01-02])",
    "TBOXFLOAT XT([1, 4],[2000-01-03, 2000-01-04])",
    "TBOXFLOAT XT([1, 1],[2000-01-05, 2000-01-05])"} */
```

- Return either an array of N spatial boxes obtained by merging the segments of a (multi)line or an array of N spatiotemporal boxes obtained by merging the instants or segments of a temporal point

  `splitNStboxes(lines, integer) → stbox[]`

  `splitNStboxes(tpoint, integer) → stbox[]`

  For temporal points, the choice between instants or segments depends on whether the interpolation is discrete or continuous. The last argument specifies the number of output boxes. If the number of instants or segments is less than or equal to the given number, the resulting array will have either one box per segment of the (multi)line or one box per instant or segment of the temporal point. Otherwise, the specified number of boxes will be obtained by merging consecutive instants or segments.

```
SELECT splitNStboxes(geometry 'Linestring(1 1,2 2,3 1,4 2,5 1)', 1);
-- {"STBOX X((1,1),(5,2))"}
SELECT splitNStboxes(geometry 'Linestring(1 1,2 2,3 1,4 2,5 1)', 2);
-- {"STBOX X((1,1),(3,2))","STBOX X((3,1),(5,2))"}
SELECT splitNStboxes(geography 'Linestring(1 1 1,2 2 1,3 1 1,4 2 1,5 1 1)', 6);
/* {"SRID=4326;GEODSTBOX Z((1,1,1),(2,2,1))",
    "SRID=4326;GEODSTBOX Z((2,1,1),(3,2,1))",
    "SRID=4326;GEODSTBOX Z((3,1,1),(4,2,1))",
```

```
    "SRID=4326;GEODSTBOX Z((4,1,1),(5,2,1))"} */
SELECT splitNStboxes(geometry 'MultiLinestring((1 1,2 2),(3 1,4 2),(5 1,6 2))', 2);
-- {"STBOX X((1,1),(4,2))","STBOX X((5,1),(6,2))"}
```

```
SELECT splitNStboxes(tgeompoint '{Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(3 1)@2000-01-03, Point(4 2)@2000-01-04, Point(5 1)@2000-01-05}', 1);
-- {"STBOX XT(((1,1),(5,2)),[2000-01-01, 2000-01-05])"}
SELECT splitNStboxes(tgeompoint '[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(3 1)@2000-01-03, Point(4 2)@2000-01-04, Point(5 1)@2000-01-05]');
/* {"STBOX XT(((1,1),(2,2)),[2000-01-01, 2000-01-02])",
    "STBOX XT(((2,1),(3,2)),[2000-01-02, 2000-01-03])",
    "STBOX XT(((3,1),(4,2)),[2000-01-03, 2000-01-04])",
    "STBOX XT(((4,1),(5,2)),[2000-01-04, 2000-01-05])"} */
SELECT splitNStboxes(tgeompoint '{[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02],
  [Point(3 1)@2000-01-03, Point(4 2)@2000-01-04], [Point(5 1)@2000-01-05]}', 2);
/* {"STBOX XT(((1,1),(4,2)),[2000-01-01, 2000-01-04])",
    "STBOX XT(((5,1),(5,1)),[2000-01-05, 2000-01-05])"} */
SELECT splitNStboxes(tgeogpoint '{Point(1 1 1)@2000-01-01, Point(2 2 1)@2000-01-02,
  Point(3 1 1)@2000-01-03, Point(4 2 1)@2000-01-04, Point(5 1 1)@2000-01-05}', 6);
/* {"SRID=4326;GEODSTBOX ZT(((1,1,1),(1,1,1)),[2000-01-01, 2000-01-01])",
    "SRID=4326;GEODSTBOX ZT(((2,2,1),(2,2,1)),[2000-01-02, 2000-01-02])",
    "SRID=4326;GEODSTBOX ZT(((3,1,1),(3,1,1)),[2000-01-03, 2000-01-03])",
    "SRID=4326;GEODSTBOX ZT(((4,2,1),(4,2,1)),[2000-01-04, 2000-01-04])",
    "SRID=4326;GEODSTBOX ZT(((5,1,1),(5,1,1)),[2000-01-05, 2000-01-05])"} */
SELECT splitNStboxes(tgeompoint '[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(3 1)@2000-01-03, Point(4 2)@2000-01-04, Point(5 1)@2000-01-05]', 6);
/* {"STBOX XT(((1,1),(2,2)),[2000-01-01, 2000-01-02])",
    "STBOX XT(((2,1),(3,2)),[2000-01-02, 2000-01-03])",
    "STBOX XT(((3,1),(4,2)),[2000-01-03, 2000-01-04])",
    "STBOX XT(((4,1),(5,2)),[2000-01-04, 2000-01-05])"} */
```

- Return either an array of spatial boxes obtained by merging N consecutive segments of a (multi)line or an array of spatiotemporal boxes obtained by merging N consecutive instants or segments of a temporal point

  `splitEachNStboxes(lines, integer) → stbox[]`

  `splitEachNStboxes(tpoint, integer) → stbox[]`

  For temporal points, the choice between instants or segments depends on whether the interpolation is discrete or continuous. The last argument specifies the number of input instants or segments that are merged to produce an output box. If the number of instants or segments is less than or equal to the given number, the resulting array will have a single box per sequence. Otherwise, the specified number of consecutive instants or segments will be merged into each output box. Notice that, contrary to the splitNStboxes function, the number of boxes in the result depends on the number of input instants or segments.

```
SELECT splitEachNStboxes(geometry 'Linestring(1 1,2 2,3 1,4 2,5 1)', 1);
-- {"STBOX X((1,1),(5,2))"}
SELECT splitEachNStboxes(geometry 'Linestring(1 1,2 2,3 1,4 2,5 1)', 2);
-- {"STBOX X((1,1),(3,2))","STBOX X((3,1),(5,2))"}
SELECT splitEachNStboxes(geography 'Linestring(1 1 1,2 2 1,3 1 1,4 2 1,5 1 1)', 6);
/* {"SRID=4326;GEODSTBOX Z((1,1,1),(2,2,1))",
    "SRID=4326;GEODSTBOX Z((2,1,1),(3,2,1))",
    "SRID=4326;GEODSTBOX Z((3,1,1),(4,2,1))",
    "SRID=4326;GEODSTBOX Z((4,1,1),(5,2,1))"} */
SELECT splitEachNStboxes(geometry 'MultiLinestring((1 1,2 2),(3 1,4 2),(5 1,6 2))', 2);
-- {"STBOX X((1,1),(4,2))","STBOX X((5,1),(6,2))"}
```

```
SELECT splitEachNStboxes(tgeompoint '{Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(3 1)@2000-01-03, Point(4 2)@2000-01-04, Point(5 1)@2000-01-05}', 1);
/* {"STBOX XT(((1,1),(1,1)),[2000-01-01, 2000-01-01])",
    "STBOX XT(((2,2),(2,2)),[2000-01-02, 2000-01-02])",
    "STBOX XT(((3,1),(3,1)),[2000-01-03, 2000-01-03])",
    "STBOX XT(((4,2),(4,2)),[2000-01-04, 2000-01-04])",
```

```
    "STBOX XT(((5,1),(5,1)),[2000-01-05, 2000-01-05])"} */
SELECT splitEachNStboxes(tgeompoint '[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(3 1)@2000-01-03, Point(4 2)@2000-01-04, Point(5 1)@2000-01-05]', 1);
/* {"STBOX XT(((1,1),(2,2)),[2000-01-01, 2000-01-02])",
    "STBOX XT(((2,1),(3,2)),[2000-01-02, 2000-01-03])",
    "STBOX XT(((3,1),(4,2)),[2000-01-03, 2000-01-04])",
    "STBOX XT(((4,1),(5,2)),[2000-01-04, 2000-01-05])"} */
SELECT splitEachNStboxes(tgeogpoint '{[Point(1 1)@2000-01-01, Point(2 2)@2000-01-02],
  [Point(3 1)@2000-01-03, Point(4 2)@2000-01-04], [Point(5 1)@2000-01-05]}', 2);
/* {"SRID=4326;GEODSTBOX XT(((1,1),(2,2)),[2000-01-01, 2000-01-02])",
    "SRID=4326;GEODSTBOX XT(((3,1),(4,2)),[2000-01-03, 2000-01-04])",
    "SRID=4326;GEODSTBOX XT(((5,1),(5,1)),[2000-01-05, 2000-01-05])"} */
```

9.5 Multidimensional Tiling

Multidimensional tiling is the mechanism used to partition the domain of temporal values in bins 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 bins 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.

Multidimensional tiling can be used for various purposes. 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 also be used for indexing purposes, where the bounding box of a temporal value can be fragmented into multiple boxes in order to improve the efficiency of the index. Finally, multidimensional tiling can be used for fragmenting temporal values according to a multidimensional grid defined over the underlying domain. This enables the distribution of 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 value/space and time, unlike the traditional hash-based partition mechanisms.

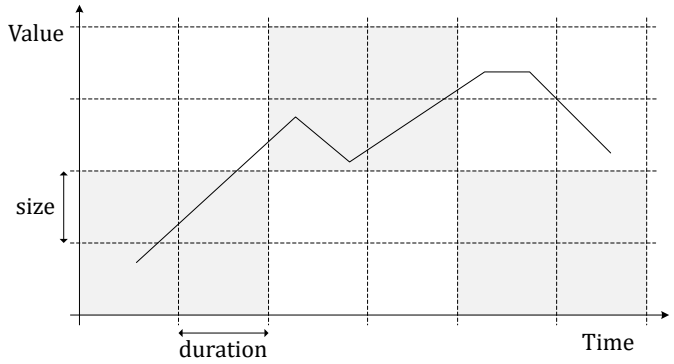


Figure 9.4: Multidimensional tiling for temporal floats.

Figure 9.4 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.

9.5.1 Bin Operations

- Return an array of bins that cover a value or time span with bins of the same size or duration

  `bins(numspans,size number,origin number=0) → span[]`

  `bins(datespans,duration interval,origin date='2000-01-03') → span[]`

  `bins(tstzspans,duration interval,origin timestamptz='2000-01-03') → span[]`

  If the origin is not specified, it is set by default to 0 for value spans and Monday, January 3, 2000 for time spans.

```
SELECT bins(floatspan '[-10, -1]', 2.5, -7);
-- {"[-12, -9.5)","[-9.5, -7)","[-7, -4.5)","[-4.5, -2)","[-2, 0.5)"}
SELECT bins(intspan '[15, 25]', 2);
-- {"[14, 16)","[16, 18)","[18, 20)","[20, 22)","[22, 24)","[24, 26)","[26, 28)"}
SELECT index, span
FROM unnest(bins(datespan '[2001-01-15, 2001-01-25]', '2 days'))
  WITH ORDINALITY t(span, index);
-- 1 | [2001-01-15, 2001-01-17)
-- 2 | [2001-01-17, 2001-01-19)
-- 3 | [2001-01-19, 2001-01-21)
-- ...
SELECT index, span
FROM unnest(bins(tstzspanset '{[2001-01-15, 2001-01-17],[2001-01-22, 2001-01-25]}',
  '2 days', '2001-01-02')) WITH ORDINALITY t(span, index);
-- 1 | [2001-01-15, 2001-01-16)
-- 2 | [2001-01-16, 2001-01-17]
-- 3 | [2001-01-22, 2001-01-24)
-- 4 | [2001-01-24, 2001-01-25]
```

- Return the bin that contains a number or a timestamp

  `getBin(number,size number,origin number=0) → span`

  `getBin(date,duration interval,origin date='2000-01-03') →`

  `getBin(timestamptz,duration interval,origin timestamptz='2000-01-03') → timestamptz`

  If the origin is not specified, it is set by default to 0 for value bins and to Monday, January 3, 2000 for time bins.

```
SELECT getBin(2, 2);
-- [2, 4)
SELECT getBin(2, 2.5, 1.5);
-- [1.5, 4)
SELECT getBin('2001-01-04', interval '1 week');
-- [2001-01-03, 2001-01-10)
SELECT getBin('2001-01-04', interval '1 week', '2001-01-07');
-- [2000-12-31, 2001-01-07)
```

9.5.2 Tile Operations

- Return the set of tiles that covers a temporal box with tiles of the same size and/or duration {}

  `valueTiles(tbox,vsize float,vorigin float=0) → {(index,tile)}`

  `timeTiles(tbox,duration interval,torigin timestamptz='2000-01-03') → {(index,tile)}`

  `valueTimeTiles(tbox,vsize float,duration interval,vorigin float=0, torigin timestamptz='2000-01-03') → {(index,tile)}`

  The result is a set of pairs `(index,tile)`. If the origin of the value and/or time dimension is not specified, it is set by default to 0 and to Monday, January 3, 2000, respectively.

```
SELECT (gr).index, (gr).tile
FROM (SELECT valueTiles(tfloat '[15@2001-01-15, 25@2001-01-25]'::tbox, 2.0) AS gr) t;
-- 1 | TBOXFLOAT X([14, 16))
-- 2 | TBOXFLOAT X([16, 18))
-- 3 | TBOXFLOAT X([18, 20))
-- ...
SELECT (gr).index, (gr).tile
FROM (SELECT timeTiles(tfloat '[15@2001-01-15, 25@2001-01-25]'::tbox, '2 days') AS gr) t;
-- 1 | TBOX T([2001-01-15, 2001-01-17))
-- 2 | TBOX T([2001-01-17, 2001-01-19))
-- 3 | TBOX T([2001-01-19, 2001-01-21))
-- ...
SELECT (gr).index, (gr).tile
FROM (SELECT valueTimeTiles(tfloat '[15@2001-01-15, 25@2001-01-25]'::tbox, 2.0, '2 days')
  AS gr) t;
-- 1 | TBOX XT([14,16),[2001-01-15,2001-01-17))
-- 2 | TBOX XT([16,18),[2001-01-15,2001-01-17))
-- 3 | TBOX XT([18,20),[2001-01-15,2001-01-17))
-- ...
SELECT valueTimeTiles(tfloat '[15@2001-01-15,25@2001-01-25]'::tbox, 2.0, '2 days', 11.5);
-- (1,"TBOX XT([13.5,15.5),[2001-01-15,2001-01-17))")
-- (2,"TBOX XT([15.5,17.5),[2001-01-15,2001-01-17))")
-- (3,"TBOX XT([17.5,19.5),[2001-01-15,2001-01-17))")
-- ...
```

- Return the set of tiles that covers a spatiotemporal box with tiles of the same size and/or duration {}

```
spaceTiles(stbox,xsize float,[ysize float,zsize float,]
 sorigin geompoint='Point(0 0 0)',borderInc bool=true) → {(index,tile)}
timeTiles(stbox,duration interval,torigin timestamptz='2000-01-03',
 borderInc bool=true) → {(index,tile)}
spaceTimeTiles(stbox,xsize float,[ysize float,zsize float,]duration interval,
 sorigin geompoint='Point(0 0 0)',torigin timestamptz='2000-01-03',
 borderInc bool=true) → {(index,tile)}
```

The result is a set of pairs `(index,tile)`. If the origin of the space and/or time dimension is not specified, it is set by default to `'Point(0 0 0)'` and to Monday, January 3, 2000, respectively. The optional argument `borderInc` states whether the upper border of the extent is included and thus, extra tiles containing the border are generated.

In the case of a spatiotemporal grid, `ysize` and `zsize` are optional, the size for the missing dimensions is assumed to be equal to `xsize`. The SRID of the tile coordinates is determined by the input box and the sizes are 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 spaceTiles(tgeompoint '[Point(3 3)@2001-01-15,
  Point(15 15)@2001-01-25]'::stbox, 2.0);
-- (1,"STBOX X((2,2),(4,4))")
-- (2,"STBOX X((4,2),(6,4))")
-- (3,"STBOX X((6,2),(8,4))")
-- ...
SELECT timeTiles(tgeompoint '[Point(3 3)@2001-01-15,
  Point(15 15)@2001-01-25]'::stbox, '2 days');
-- (1,"STBOX T([2001-01-15, 2001-01-17))")
-- (2,"STBOX T([2001-01-17, 2001-01-19))")
-- (3,"STBOX T([2001-01-19, 2001-01-21))")
-- ...
SELECT spaceTiles(tgeompoint 'SRID=3812;[Point(3 3)@2001-01-15,
  Point(15 15)@2001-01-25]'::stbox, 2.0, geometry 'Point(3 3)');
-- (1,"SRID=3812;STBOX X((3,3),(5,5))")
```

```
-- (2,"SRID=3812;STBOX X((5,3),(7,5))")
-- (3,"SRID=3812;STBOX X((7,3),(9,5))")
-- ...
SELECT spaceTiles(tgeompoint '[Point(3 3 3)@2001-01-15,
  Point(15 15 15)@2001-01-25]'::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 spaceTimeTiles(tgeompoint '[Point(3 3)@2001-01-15,
  Point(15 15)@2001-01-25]'::stbox, 2.0, interval '2 days');
-- (1,"STBOX XT(((2,2),(4,4)),[2001-01-15,2001-01-17))")
-- (2,"STBOX XT(((4,2),(6,4)),[2001-01-15,2001-01-17))")
-- (3,"STBOX XT(((6,2),(8,4)),[2001-01-15,2001-01-17))")
-- ...
SELECT spaceTimeTiles(tgeompoint '[Point(3 3 3)@2001-01-15,
  Point(15 15 15)@2001-01-25]'::stbox, 2.0, interval '2 days',
  'Point(3 3 3)', '2001-01-15');
-- (1,"STBOX ZT(((3,3,3),(5,5,5)),[2001-01-15,2001-01-17))")
-- (2,"STBOX ZT(((5,3,3),(7,5,5))),[2001-01-15,2001-01-17)")
-- (3,"STBOX ZT(((7,3,3),(9,5,5))),[2001-01-15,2001-01-17)")
-- ...
```

- Return the temporal tile that covers a value and/or a timestamp

```
getValueTile(value float,vsize float,vorigin float=0.0,) → tbox
getTboxTimeTile(time timestamptz,duration interval,
 torigin timestamptz='2000-01-03') → tbox
getValueTimeTile(value float,time timestamptz,size float,duration interval,
 vorigin float=0.0,torigin timestamptz='2000-01-03') → tbox
```

If the origin of the value and/or time dimension is not specified, it is set by default to 0 and to Monday, January 3, 2000, respectively.

```
SELECT getValueTile(15, 2);
-- TBOX ([14,16))
SELECT getTboxTimeTile('2001-01-15', interval '2 days');
-- TBOX ([2001-01-15,2001-01-17))
SELECT getValueTimeTile(15, '2001-01-15', 2, interval '2 days');
-- TBOX XT([14,16),[2001-01-15,2001-01-17))
SELECT getValueTimeTile(15, '2001-01-15', 2, interval '2 days', 1, '2001-01-02');
-- TBOX XT([15,17),[2001-01-14,2001-01-16))
```

- Return the spatiotemporal tile that covers a point and/or a timestamp

```
getSpaceTile(point geometry,xsize float,[ysize float,zsize float],
 sorigin geompoint='Point(0 0 0)') → stbox
getStboxTimeTile(time timestamptz,duration interval,
 torigin timestamptz='2000-01-03') → stbox
getSpaceTimeTile(point geometry,time timestamptz,xsize float,
 [ysize float,zsize float,]duration,interval,sorigin geompoint='Point(0 0 0)',
 torigin timestamptz='2000-01-03') → stbox
```

If the origin of the space and/or time dimension is not specified, it is set by default to `Point(0 0 0)` and to Monday, January 3, 2000, respectively.

In the case of a spatiotemporal grid, `ysize` and `zsize` are optional, the size for the missing dimensions is assumed to be equal to `xsize`. The SRID of the tile coordinates is determined by the input point and the sizes are 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 getSpaceTile(geometry 'Point(1 1 1)', 2.0);
-- STBOX Z((0,0,0),(2,2,2))
SELECT getStboxTimeTile(timestamptz '2001-01-01', interval '2 days');
-- STBOX T([2001-01-01,2001-01-03))
SELECT getSpaceTimeTile(geometry 'Point(1 1)', '2001-01-01', 2.0, interval '2 days');
-- STBOX XT(((0,0),(2,2)),[2001-01-01, 2001-01-03))
SELECT getSpaceTimeTile(geometry 'Point(1 1)', '2001-01-01', 2.0, interval '2 days',
  'Point(1 1)', '2001-01-02');
-- STBOX XT(((1,1),(3,3)),[2000-12-31, 2001-01-02))
```

9.5.3 Bounding Box Operations

These functions fragment the bounding box of a temporal value with respect to a multidimensional grid. They provide an alternative to the functions in Section 9.4 to split the bounding boxes by specifying the maximum size of the boxes in the various dimensions.

- Return an array of value or time spans obtained from the instants or segments of a temporal number with respect to a value or time grid

  `valueBins(tnumber,vsize number,vorigin number=0) → numspan[]`

  `timeBins(temp,duration interval,torigin timestamptz='2000-01-03') → tstzspan[]`

  The choice between instants or segments depends on whether the interpolation is discrete or continuous. If the origin of values or time is not specified, it is set by default to 0 for value spans or to Monday, January 3, 2000 for time spans.

```
SELECT valueBins(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05}', 3);
-- {"[1, 3)","[4, 5)"}
SELECT valueBins(tfloat '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05]', 3, 1);
-- {"[1, 4)","[4, 4]"}
```

```
SELECT timeBins(ttext '{AAA@2000-01-01, BBB@2000-01-02, AAA@2000-01-03, CCC@2000-01-04,
  AAA@2000-01-05}', '3 days');
-- {"[2000-01-01, 2000-01-02]","[2000-01-03, 2000-01-05]"}
SELECT timeBins(tfloat '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05]', '3 days', '2000-01-01');
-- {"[2000-01-01, 2000-01-04)","[2000-01-04, 2000-01-05)"}
```

- Return an array of temporal boxes obtained from the instants or segments of a temporal number with respect to a value and/or time grid

  `valueBoxes(tnumber,size number,vorigin number=0) → tbox[]`

  `timeBoxes(tnumber,duration interval,torigin timestamptz='2000-01-03') → tbox[]`

  `valueTimeBoxes(tnumber,size number,duration interval,vorigin number=0,`
  `torigin timestamptz='2000-01-03') → tbox[]`

  The choice between instants or segments depends on whether the interpolation is discrete or continuous. If the origin of value and/or time dimension is not specified, it is set by default to 0 and to Monday, January 3, 2000, respectively.

```
SELECT valueBoxes(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
1@2000-01-05}', 3);
/* {"TBOXINT XT([1, 3),[2000-01-01, 2000-01-05])",
    "TBOXINT XT([4, 5),[2000-01-04, 2000-01-04])"} */
SELECT timeBoxes(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
1@2000-01-05}', '3 days');
/* {"TBOXINT XT([1, 3),[2000-01-01, 2000-01-02])",
    "TBOXINT XT([1, 5),[2000-01-03, 2000-01-05])"} */
```

```
SELECT valueTimeBoxes(tint '{1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05}', 3, '3 days');
/* {"TBOXINT XT([1, 3),[2000-01-01, 2000-01-02])",
    "TBOXINT XT([1, 2),[2000-01-03, 2000-01-05])",
    "TBOXINT XT([4, 5),[2000-01-04, 2000-01-04])"} */
SELECT valueTimeBoxes(tfloat '[1@2000-01-01, 2@2000-01-02, 1@2000-01-03, 4@2000-01-04,
  1@2000-01-05]', 3, '3 days', 1, '2000-01-01');
/* {"TBOXFLOAT XT([1, 4),[2000-01-01, 2000-01-04))",
    "TBOXFLOAT XT([1, 4),(2000-01-04, 2000-01-05])",
    "TBOXFLOAT XT([4, 4],[2000-01-04, 2000-01-04])"} */
```

- Return an array of spatiotemporal boxes obtained from the instants or segments of a temporal point with respect to a space and/or time grid

```
spaceBoxes(tgeompoint,xsize float,[ysize float,zsize float,]
 sorigin geompoint='Point(0 0 0)',borderInc bool=true) → stbox[]
timeBoxes(tgeompoint,duration interval,torigin timestamptz='2000-01-03',
 borderInc bool=true) → stbox[]
spaceTimeBoxes(tgeompoint,xsize float,[ysize float,zsize float,]duration interval,
 sorigin geompoint='Point(0 0 0)',torigin timestamptz='2000-01-03',
 borderInc bool=true) → stbox[]
```

The choice between instants or segments depends on whether the interpolation is discrete or continuous. The arguments `ysize` and `zsize` are optional, the size for the missing dimensions is assumed to be equal to `xsize`. The SRID of the tile coordinates is determined by the temporal point and the sizes are 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 temporal point, otherwise an error is raised. If the origin of space and/or time dimension is not specified, it is set by default to `'Point(0 0 0)'` and to Monday, January 3, 2000, respectively. The optional argument `borderInc` states whether the upper border of the extent is included and thus, extra tiles containing the border are generated.

```
SELECT spaceBoxes(tgeompoint '{Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(1 1)@2000-01-03, Point(4 4)@2000-01-04, Point(1 1)@2000-01-05}', 3);
/* {"STBOX XT(((1,1),(2,2)),[2000-01-01, 2000-01-05])",
   "STBOX XT(((4,4),(4,4)),[2000-01-04, 2000-01-04])"} */
SELECT timeBoxes(tgeompoint '{Point(1 1 1)@2000-01-01, Point(2 2 2)@2000-01-02,
  Point(1 1 1)@2000-01-03, Point(4 4 4)@2000-01-04, Point(1 1 1)@2000-01-05}',
  interval '2 days', '2000-01-01');
/* {"STBOX ZT(((1,1,1),(2,2,2)),[2000-01-01, 2000-01-02])",
    "STBOX ZT(((1,1,1),(4,4,4)),[2000-01-03, 2000-01-04])",
    "STBOX ZT(((1,1,1),(1,1,1)),[2000-01-05, 2000-01-05])"} */
SELECT spaceTimeBoxes(tgeompoint '{Point(1 1)@2000-01-01, Point(2 2)@2000-01-02,
  Point(1 1)@2000-01-03, Point(4 4)@2000-01-04, Point(1 1)@2000-01-05}', 3, '3 days');
/* {"STBOX XT(((1,1),(2,2)),[2000-01-01, 2000-01-02])",
    "STBOX XT(((1,1),(1,1)),[2000-01-03, 2000-01-05])",
    "STBOX XT(((4,4),(4,4)),[2000-01-04, 2000-01-04])"} */
SELECT spaceTimeBoxes(tgeompoint '[Point(1 1 1)@2000-01-01, Point(2 2 2)@2000-01-02,
  Point(1 1 1)@2000-01-03, Point(4 4 4)@2000-01-04, Point(1 1 1)@2000-01-05]',
  3, interval '3 days', 'Point(1 1 1)', '2000-01-01');
/* {"STBOX ZT(((1,1,1),(4,4,4)),[2000-01-01, 2000-01-04))",
    "STBOX ZT(((1,1,1),(4,4,4)),(2000-01-04, 2000-01-05])",
    "STBOX ZT(((4,4,4),(4,4,4)),[2000-01-04, 2000-01-04])"} */
```

9.5.4 Splitting Operations

These functions split a temporal value with respect to a sequence of bins (see Section 9.5.1) or tiles (see Section 9.5.2).

- Split a temporal number with respect to value and/or time bins {}

  `valueSplit(tnumber,size number,origin number=0) → {(number,tnumber)}`

  `timeSplit(ttype,duration interval,origin timestamptz='2000-01-03') → {(time,temp)}`

  `valueTimeSplit(tnumber,size number,duration interval,vorigin number=0, torigin timestamptz='2000-01-03') → {(number,time,tnumber}`

  The result is a set of pairs or a set of triples. If the origin of the value and/or the time dimension is not specified, they are set by default to 0 and to Monday, January 3, 2000, respectively.

```
SELECT (sp).number, (sp).tnumber
FROM (SELECT valueSplit(tint '[1@2001-01-01, 2@2001-01-02, 5@2001-01-05, 10@2001-01-10]',
  2) AS sp) t;
--  0 | {[1@2001-01-01, 1@2001-01-02)}
--  2 | {[2@2001-01-02, 2@2001-01-05)}
--  4 | {[5@2001-01-05, 5@2001-01-10)}
-- 10 | {[10@2001-01-10]}
SELECT valueSplit(tfloat '[1@2001-01-01, 10@2001-01-10)', 2.0, 1.0);
-- (1,"{[1@2001-01-01, 3@2001-01-03)}")
-- (3,"{[3@2001-01-03, 5@2001-01-05)}")
-- (5,"{[5@2001-01-05, 7@2001-01-07)}")
-- (7,"{[7@2001-01-07, 9@2001-01-09)}")
-- (9,"{[9@2001-01-09, 10@2001-01-10 00:00:00+01)}")
```

```
SELECT (ts).time, (ts).temp
FROM (SELECT timeSplit(tfloat '[1@2001-02-01, 10@2001-02-10)', '2 days') AS ts) t;
-- 2001-01-31 | [1@2001-02-01, 2@2001-02-02)
-- 2001-02-02 | [2@2001-02-02, 4@2001-02-04)
-- 2001-02-04 | [4@2001-02-04, 6@2001-02-06)
-- ...
SELECT (ts).time, astext((ts).temp) AS temp
FROM (SELECT timeSplit(tgeompoint '[Point(1 1)@2001-02-01, Point(10 10)@2001-02-10]',
  '2 days', '2001-02-01') AS ts) AS t;
-- 2001-02-01 | [POINT(1 1)@2001-02-01, POINT(3 3)@2001-02-03)
-- 2001-02-03 | [POINT(3 3)@2001-02-03, POINT(5 5)@2001-02-05)
-- 2001-02-05 | [POINT(5 5)@2001-02-05, POINT(7 7)@2001-02-07)
-- ...
```

Notice that we can split a temporal value in cyclic (instead of linear) time bins. The following two examples show how to split 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@2001-01-01, 10@2001-01-03]', '1 hour') AS ts) t
GROUP BY hour ORDER BY hour;
/* 00:00:00 | {[1@2001-01-01 00:00:00+01, 1.1875@2001-01-01 01:00:00+01),
               [5.5@2001-01-02 00:00:00+01, 5.6875@2001-01-02 01:00:00+01)} */
/* 01:00:00 | {[1.1875@2001-01-01 01:00:00+01, 1.375@2001-01-01 02:00:00+01),
               [5.6875@2001-01-02 01:00:00+01, 5.875@2001-01-02 02:00:00+01)} */
/* 02:00:00 | {[1.375@2001-01-01 02:00:00+01, 1.5625@2001-01-01 03:00:00+01),
               [5.875@2001-01-02 02:00:00+01, 6.0625@2001-01-02 03:00:00+01)} */
/* 03:00:00 | {[1.5625@2001-01-01 03:00:00+01, 1.75@2001-01-01 04:00:00+01),
               [6.0625@2001-01-02 03:00:00+01, 6.25@2001-01-02 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)@2001-01-01, Point(10 10)@2001-01-14)',
  '1 hour') AS ts) t
GROUP BY dow, dow_no ORDER BY dow_no;
/* 0 | Sun | {[POINT(1 1)@2001-01-01, POINT(1.69 1.69)@2001-01-02),
             [POINT(5.85 5.85)@2001-01-08, POINT(6.54 6.54)@2001-01-09)} */
```

```
/* 1 | Mon | {[POINT(1.69 1.69)@2001-01-02, POINT(2.38 2.38)@2001-01-03),
             [POINT(6.54 6.54)@2001-01-09, POINT(7.23 7.23)@2001-01-10)} */
/* 2 | Tue | {[POINT(2.38 2.38)@2001-01-03, POINT(3.08 3.08)@2001-01-04),
             [POINT(7.23 7.23)@2001-01-10, POINT(7.92 7.92)@2001-01-11)} */
/* ... */
```

```
SELECT (sp).number, (sp).time, (sp).tnumber
FROM (SELECT valueTimeSplit(tint '[1@2001-02-01, 2@2001-02-02, 5@2001-02-05,
  10@2001-02-10]', 5, '5 days') AS sp) t;
--  0 | 2001-02-01 | {[1@2001-02-01, 2@2001-02-02, 2@2001-02-05)}
--  5 | 2001-02-01 | {[5@2001-02-05, 5@2001-02-06)}
--  5 | 2001-02-06 | {[5@2001-02-06, 5@2001-02-10)}
-- 10 | 2001-02-06 | {[10@2001-02-10]}
SELECT (sp).number, (sp).time, (sp).tnumber
FROM (SELECT valueTimeSplit(tfloat '[1@2001-02-01, 10@2001-02-10)', 5.0, '5 days', 1.0,
  '2001-02-01') AS sp) t;
-- 1 | 2001-01-01 | [1@2001-01-01, 6@2001-01-06)
-- 6 | 2001-01-06 | [6@2001-01-06, 10@2001-01-10)
```

- Split a temporal point with respect to a spatial grid 🧊{}

```
spaceSplit(tgeompoint,xsize float,[ysize float,zsize float,]
 origin geompoint='Point(0 0 0)',bitmatrix boolean=true,borderInc bool=true) →
 {(point,tpoint)}
timeSplit(tpoint,duration interval,torigin timestamptz='2000-01-03') → {(time,tpoint)}
spaceTimeSplit(tgeompoint,xsize float,[ysize float,zsize float,]
 duration interval,sorigin geompoint='Point(0 0 0)',
 torigin timestamptz='2000-01-03',bitmatrix boolean=true,borderInc boolean=true) →
 {(point,time,tpoint)}
```

The result is a set of pairs or a set of triples. If the origin of the space and/or time dimension is not specified, it is set by default to `'Point(0 0 0)'` and to Monday, January 3, 2000, respectively. The arguments `ysize` and `zsize` are optional, the size for the missing dimensions is assumed to be equal to `xsize`. If the argument `bitmatrix` is not specified, then the computation will use a bit matrix to speed up the process. The optional argument `borderInc` states whether the upper border of the extent is included and thus extra tiles containing the border are generated.

```
SELECT ST_AsText((sp).point) AS point, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceSplit(tgeompoint '[Point(1 1)@2001-03-01, Point(10 10)@2001-03-10]',
  2.0) AS sp) t;
-- POINT(0 0) | {[POINT(1 1)@2001-03-01, POINT(2 2)@2001-03-02)}
-- POINT(2 2) | {[POINT(2 2)@2001-03-02, POINT(4 4)@2001-03-04)}
-- POINT(4 4) | {[POINT(4 4)@2001-03-04, POINT(6 6)@2001-03-06)}
-- ...
SELECT ST_AsText((sp).point) AS point, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceSplit(tgeompoint '[Point(1 1 1)@2001-03-01,
  Point(10 10 10)@2001-03-10]', 2.0, geometry 'Point(1 1 1)') AS sp) t;
-- POINT Z(1 1 1) | {[POINT Z (1 1 1)@2001-03-01, POINT Z (3 3 3)@2001-03-03)}
-- POINT Z(3 3 3) | {[POINT Z (3 3 3)@2001-03-03, POINT Z (5 5 5)@2001-03-05)}
-- POINT Z(5 5 5) | {[POINT Z (5 5 5)@2001-03-05, POINT Z (7 7 7)@2001-03-07)}
-- ...
```

```
SELECT (sp).time, astext((sp).temp) AS tpoint
FROM (SELECT timeSplit(tgeompoint '[Point(1 1)@2001-02-01, Point(10 10)@2001-02-10]',
  interval '2 days') AS sp) t;
-- 2001-01-31 | [POINT(1 1)@2001-02-01, POINT(2 2)@2001-02-02)
-- 2001-02-02 | [POINT(2 2)@2001-02-02, POINT(4 4)@2001-02-04)
-- 2001-02-04 | [POINT(4 4)@2001-02-04, POINT(6 6)@2001-02-06)
-- ...
```

```
SELECT ST_AsText((sp).point) AS point, (sp).time, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceTimeSplit(tgeompoint '[Point(1 1)@2001-02-01, Point(10 10)@2001-02-10]',
  2.0, interval '2 days') AS sp) t;
-- POINT(0 0) | 2001-01-31 | {[POINT(1 1)@2001-02-01, POINT(2 2)@2001-02-02)}
-- POINT(2 2) | 2001-01-31 | {[POINT(2 2)@2001-02-02]}
-- POINT(2 2) | 2001-02-02 | {[POINT(2 2)@2001-02-02, POINT(4 4)@2001-02-04)}
-- ...
SELECT ST_AsText((sp).point) AS point, (sp).time, astext((sp).tpoint) AS tpoint
FROM (SELECT spaceTimeSplit(tgeompoint '[Point(1 1 1)@2001-02-01,
  Point(10 10 10)@2001-02-10]', 2.0, interval '2 days', 'Point(1 1 1)',
  '2001-03-01') AS sp) t;
-- POINT Z(1 1 1) | 2001-02-01 | {[POINT Z(1 1 1)@2001-02-01, POINT Z(3 3 3)@2001-02-03)}
-- POINT Z(3 3 3) | 2001-02-01 | {[POINT Z(3 3 3)@2001-02-03]}
-- POINT Z(3 3 3) | 2001-02-03 | {[POINT Z(3 3 3)@2001-02-03, POINT Z (5 5 5)@2001-02-05)}
-- ...
```

Chapter 10

Temporal Types: Aggregation and Indexing

10.1 Aggregation

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 `tstzspan`, 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 4.2.

- Temporal count

  `tCount(ttype) → {tintSeq,tintSeqSet}`

```
SELECT tCount(NoEmps) FROM Department;
-- {[1@2001-01-01, 2@2001-02-01, 1@2001-08-01, 1@2001-10-01)}
```

- Bounding box extent

  `extent(temp) → {tstzspan,tbox,stbox}`

```
SELECT extent(noEmps) FROM Department;
-- TBOX XT((4,12),[2001-01-01,2001-10-01])
SELECT extent(Trip) FROM Trips;
-- STBOX XT(((0,0),(3,3)),[2001-01-01 08:00:00+01, 2001-01-01 08:20:00+01))
```

- Temporal and, temporal or

  `tOr(tbool) → tbool`

  `tAnd(tbool) → tbool`

```
SELECT tAnd(NoEmps #> 6) FROM Department;
-- {[t@2001-01-01, f@2001-04-01, f@2001-10-01)}
SELECT tOr(NoEmps #> 6) FROM Department;
-- {[t@2001-01-01, f@2001-08-01, f@2001-10-01)}
```

- Temporal minimum, maximum, sum, and average

  `tMin(ttype) → ttype`

  `tMax(ttype) → ttype`

  `tSum(tnumber) → {tnumberSeq,tnumberSeqSet}`

  `tAvg(tnumber) → {tfloatSeq,tfloatSeqSet}`

```
SELECT tMin(NoEmps) FROM Department;
-- {[10@2001-01-01, 4@2001-02-01, 6@2001-06-01, 6@2001-10-01)}
SELECT tMax(NoEmps) FROM Department;
-- {[10@2001-01-01, 12@2001-04-01, 6@2001-08-01, 6@2001-10-01)}
SELECT tSum(NoEmps) FROM Department;
/* {[10@2001-01-01, 14@2001-02-01, 16@2001-04-01, 18@2001-06-01, 6@2001-08-01,
   6@2001-10-01)} */
SELECT tAvg(NoEmps) FROM Department;
/* {[10@2001-01-01, 10@2001-02-01), [7@2001-02-01, 7@2001-04-01),
   [8@2001-04-01, 8@2001-06-01), [9@2001-06-01, 9@2001-08-01),
   [6@2001-08-01, 6@2001-10-01) */
```

- Window count

  `wCount(tnumber,interval) → {tintSeq,tintSeqSet}`

```
SELECT wCount(NoEmps, interval '2 days') FROM Department;
/* {[1@2001-01-01, 2@2001-02-01, 3@2001-04-01, 2@2001-04-03, 3@2001-06-01, 2@2001-06-03,
   1@2001-08-03, 1@2001-10-03)} */
```

- Window minimum, maximum, sum, and average

  `wMin(tnumber,interval) → {tnumberSeq,tnumberSeqSet}`

  `wMax(tnumber,interval) → {tnumberDiscSeq,tnumberSeqSet}`

  `wSum(tint,interval) → {tintSeq,tintSeqSet}`

  `wAvg(tint,interval) → {tfloatSeq,tfloatSeqSet}`

```
SELECT wMin(NoEmps, interval '2 days') FROM Department;
-- {[10@2001-01-01, 4@2001-04-01, 6@2001-06-03, 6@2001-10-03)}
SELECT wMax(NoEmps, interval '2 days') FROM Department;
-- {[10@2001-01-01, 12@2001-04-01, 6@2001-08-03, 6@2001-10-03)}
SELECT wSum(NoEmps, interval '2 days') FROM Department;
/* {[10@2001-01-01, 14@2001-02-01, 26@2001-04-01, 16@2001-04-03, 22@2001-06-01,
   18@2001-06-03, 6@2001-08-03, 6@2001-10-03)} */
SELECT round(wAvg(NoEmps, interval '2 days'), 3) FROM Department;
/*  Interp=Step;{[10@2001-01-01, 7@2001-02-01, 8.667@2001-04-01, 8@2001-04-03,
   7.333@2001-06-01, 9@2001-06-03, 6@2001-08-03, 6@2001-10-03)} */
```

- Temporal centroid

  `tCentroid(tgeompoint) → tgeompoint`

```
SELECT tCentroid(Trip) FROM Trips;
/* {[POINT(0 0)@2001-01-01 08:00:00+00, POINT(1 0)@2001-01-01 08:05:00+00),
   [POINT(0.5 0)@2001-01-01 08:05:00+00, POINT(1.5 0.5)@2001-01-01 08:10:00+00,
   POINT(2 1.5)@2001-01-01 08:15:00+00),
   [POINT(2 2)@2001-01-01 08:15:00+00, POINT(3 3)@2001-01-01 08:20:00+00)} */
```

10.2 Indexing

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 4, these are

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

A GiST or SP-GiST index can accelerate queries involving the following operators (see Section 5.3 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, `tstzspan`, `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 span, 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/span 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 && intspan '[1, 5)';
SELECT * FROM Department WHERE NoEmps && tstzspan '[2001-04-01, 2001-05-01)';
SELECT * FROM Department WHERE NoEmps &&
  tbox(intspan '[1, 5)', tstzspan '[2001-04-01, 2001-05-01)');
SELECT * FROM Department WHERE NoEmps &&
  tfloat '{[1@2001-01-01, 1@2001-02-01), [5@2001-04-01, 5@2001-05-01)}';
```

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 && tstzspan '[2001-01-01, 2001-01-05)';
SELECT * FROM Trips WHERE Trip &&
  stbox(geometry 'Polygon((0 0,0 1,1 1,1 0,0 0))', tstzspan '[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 for temporal types, 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 atTime(T.Trip, tstzspan '[2001-01-01, 2001-01-02)')
FROM Trips T
-- Bouding box index filtering
WHERE T.Trip && tstzspan '[2001-01-01, 2001-01-02)';
```

In the case of temporal points, all spatial relationships with the ever and always semantics (see Section 8.5) 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 eIntersects(T.Trip, R.Geom);
```

10.3 Statistics and Selectivity

10.3.1 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 span 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 span types, like `tstzspan`, three additional histograms are collected:

7. length histogram of non-empty spans,
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 time and span types `tstzset`, `tstzspan`, `tstzspanset`, `intspan`, and `floatspan` replicate those collected by PostgreSQL for the `tstzrange`. This is clear for the span types in MobilityDB, which are more efficient versions of the range types in PostgreSQL. For the `tstzset` and the `tstzspanset` types, a value is converted into its bounding period, then the statistics for the `tstzspan` 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 step 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 spans.
- For columns of temporal point types (that is, `tgeompoint` and `tgeogpoint`) the statistics (9)–(10) are collected for the points.

10.3.2 Selectivity Estimation

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 `tstzspan`, `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 both restriction and join selectivity for time, span, and temporal types.

Chapter 11

Temporal Poses

The `pose` type is used to represent the *location* and *orientation* of geometric objects within coordinate systems anchored to the earth's surface or within other astronomical coordinate systems. The location is represented by a 2D or 3D point. For 2D poses, the orientation is defined by a rotation angle in (-π, π] expressed in radians. For 3D poses, the orientation is defined by four float values `W`, `X`, `Y`, `Z`, representing a unit quaternion $Q = \langle W, X, Y, Z \rangle$ where $\| Q^2 \| = \sqrt{W^2 + X^2 + Y^2 + Z^2} = 1$.

The GeoPose Standards Working Group (SWG), working under the auspices of the Open Geospatial Consortium, has defined a standard for exchanging pose information across different users, devices, and platforms. More information about the standard can be found in the GitHub repository of the GeoPose SWG.

The `pose` type serves as base type for defining the temporal pose type `tpose`. The `tpose` type has similar functionality as the temporal point type `tgeompoint`. Thus, most functions and operators described before for the `tgeompoint` type are also applicable for the `tpose` type. In addition, there are specific functions defined for the `tpose` type. We cover these functions in this chapter.

The `tpose` type is used for defining the type `trgeometry` (that is, temporal rigid geometry) defined in the next chapter. The implementation of the these types in MobilityDB has been studied in the following PhD thesis.

- Maxime Schoemans, *Managing Rigid Temporal Geometries in Moving Object Databases*, PhD thesis, Université libre de Bruxelles, 2025.

11.1 Static Poses

A 2D `pose` is a couple of the form `(point2D,radius)` where `point2D` is a 2D geometric point and `radius` is a `float` value representing a rotation angle in (-π, π] expressed in radians. A 3D `pose` is a tuple of the form `(point3D,W,Y,X,Z)` where `point3D` is a 3D geometric point, and `W`, and `X`, `Y`, and `Z` are four `floats` representing a *unit* quaternion. Examples of input of pose values are as follows:

```
SELECT pose 'Pose(Point(1 1), 0.5)';
SELECT pose 'Pose(Point Z(1 1 1), 0.5, 0.5, 0.5, 0.5)';
```

An SRID can be specified for a pose either at the begining of the pose literal or before the point literal as shown below.

```
SELECT pose 'SRID=3812;Pose(Point(1 1), 0.5)';
SELECT pose 'Pose(SRID=5676;Point Z(1 1 1), 0.5, 0.5, 0.5, 0.5)';
```

Values of the pose type must satisfy several constraints so that they are well defined. Examples of incorrect pose type values are as follows.

```
-- Empty point
select pose 'Pose(Point empty, 0.5)';
-- Incorrect point value
SELECT pose 'Pose(Linestring(1 1,2 2), 1.0)';
```

```
-- Incorrect radius value
SELECT pose 'Pose(Point(1 1), -10.0)';
-- Incorrect 3D point
SELECT pose 'Pose(Point Z(1 1), 1.0)';
-- Incomplete 3D orientation
SELECT pose 'Pose(Point Z(1 1 1), 1.0)';
```

We give next the functions and operators for the pose type.

11.1.1 Input and Output

- Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation

  `asText({pose,pose[]}) → {text,text[]}`

  `asEWKT({pose,pose[]}) → {text,text[]}`

```
SELECT asText(pose 'SRID=4326;Pose(Point(0 0),1)');
-- Pose(POINT(0 0),1)
SELECT asText(ARRAY[pose 'Pose(Point(0 0),1)', 'Pose(Point(1 1),2)']);
-- {"Pose(POINT(0 0),1)","Pose(POINT(1 1),2)"}
SELECT asEWKT(pose 'SRID=4326;Pose(Point(0 0),1)');
-- SRID=4326;Pose(Point(0 0),1)
SELECT asEWKT(ARRAY[pose 'Pose(SRID=5676;Point(0 0),1)', 'Pose(SRID=5676;Point(1 1),2)']);
-- {"Pose(SRID=5676;POINT(0 0),1)","Pose(SRID=5676;POINT(1 1),2))"}
```

- Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB), or the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `asBinary(pose,endian text='') → bytea`

  `asEWKB(pose,endian text='') → bytea`

  `asHexEWKB(pose,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 the encoding of the machine is used.

```
SELECT asBinary(pose 'Pose(Point(1 2),1)');
-- \x0101000000000000f03f0000000000000040000000000000f03f
SELECT asEWKB(pose 'SRID=7844;Pose(Point(1 2),1)');
-- \x0141a41e0000000000000000f03f0000000000000040000000000000f03f
SELECT asHexEWKB(pose 'SRID=3812;Pose(Point(1 2),1)');
-- 0141E40E0000000000000000F03F0000000000000040000000000000F03F
```

- Input from the Well-Known Text (WKT) or from the Extended Well-Known Text (EWKT) representation

  `poseFromText(text) → pose`

  `poseFromEWKT(text) → pose`

```
SELECT asEWKT(poseFromText(text 'Pose(Point(1 2),1)'));
-- Pose(POINT(1 2),1)
SELECT asEWKT(poseFromEWKT(text 'SRID=3812;Pose(Point(1 2),1)'));
-- SRID=3812;Pose(Point(1 2),1)
```

- Input from the Well-Known Binary (WKB), from the Extended Well-Known Binary (EWKB), or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `poseFromBinary(bytea) → pose`

  `poseFromEWKB(bytea) → pose`

  `poseFromHexEWKB(text) → pose`

```
SELECT asEWKT(poseFromBinary(
  '\x0101000000000000f03f0000000000000040000000000000f03f'));
-- Pose(POINT(1 2),1)
SELECT asEWKT(poseFromEWKB(
  '\x0141a41e0000000000000000f03f0000000000000040000000000000f03f'));
-- SRID=7844;Pose(Point(1 2),1)
SELECT asEWKT(poseFromHexEWKB(
  '0141E40E0000000000000000F03F0000000000000040000000000000F03F'));
-- SRID=3812;Pose(POINT(1 2),1)
```

11.1.2 Constructors

- Constructor for poses

  `pose(geompoint2D,float)` → `pose`

  `pose(geompoint3D,float,float,float,float)` → `pose`

```
SELECT asText(pose(ST_Point(1,1), radians(45)), 6);
-- Pose(POINT(1 1),0.785398)
SELECT asEWKT(pose(ST_Point(1,1,3812), radians(45)), 6);
-- SRID=3812;Pose(POINT(1 1),0.785398)
SELECT asText(pose(ST_PointZ(1,1,1), 1, 0, 0, 0));
-- Pose(POINT Z (1 1 1),1,0,0,0)
```

11.1.3 Conversions

Values of the `pose` type can be converted to the `geometry` point type using an explicit `CAST` or using the `::` notation as shown below.

- Convert a pose and, optionally, a timestamp or a period, into a spatiotemporal box

  `pose::stbox`

  `stbox(pose)` → `stbox`

  `stbox(pose,{timestamptz,tstzspan})` → `stbox`

```
SELECT stbox(pose 'SRID=5676;Pose(Point(1 1),0.3)');
-- SRID=5676;STBOX X((1,1),(1,1))
SELECT stbox(pose 'Pose(Point(1 1),0.3)', timestamptz '2001-01-01');
-- STBOX XT(((1,1),(1.3,1.3)),[2001-01-01, 2001-01-01])
SELECT stbox(pose 'Pose(Point(1 1),0.3)', tstzspan '[2001-01-01,2001-01-02]');
-- STBOX XT(((1,1),(1.3,1.3)),[2001-01-01, 2001-01-02])
```

- Convert a pose into geometry point

  `pose::geompoint`

```
SELECT ST_AsText(pose(ST_Point(1, 1), 1)::geometry);
-- Point(1 1)
SELECT ST_AsEWKT(pose(ST_PointZ(1, 1, 1, 5676), 1, 0, 0, 0)::geometry);
-- SRID=5676;POINT(1 1 1)
```

11.1.4 Accessors

- Return the point

  `point(pose)` → `geompoint`

```
SELECT ST_AsText(point(pose 'Pose(Point(1 1), 0.3)'));
-- Point(1 1)
```

- Return the rotation

  `rotation(pose2D) → float`

```
SELECT rotation(pose 'Pose(Point(1 1), 0.3)');
-- 0.3
```

- Return the orientation

  `orientation(pose3D) → (X,W,Z,T)`

```
SELECT orientation(pose 'Pose(Point Z(1 1 1), 0, 0, 0, 1)');
-- (0, 0, 0, 1)
```

11.1.5 Transformations

- Round the point and the orientation of the pose to the number of decimal places

  `round(pose,integer=0) → pose`

```
SELECT asText(round(pose(ST_Point(1.123456789,1.123456789), 0.123456789), 6));
-- Pose(POINT(1.123457 1.123457),0.123457)
```

11.1.6 Spatial Reference System

- Return or set the spatial reference identifier

  `srid(pose) → integer`

  `setSRID(pose) → pose`

```
SELECT SRID(pose 'Pose(SRID=5676;Point(1 1), 0.3)');
-- 5676
SELECT asEWKT(setSRID(pose 'Pose(Point(0 0),1)', 4326));
-- SRID=4326;Pose(POINT(0 0),1)
```

- Transform to a spatial reference identifier

  `transform(pose,integer) → pose`

  `transformPipeline(pose,pipeline text,to_srid integer,is_forward bool=true) → pose`

  The `transform` function specifies the transformation with a target SRID. An error is raised when the input pose has an unknown SRID (represented by 0).

  The `transformPipeline` function specifies the transformation with a defined coordinate transformation pipeline represented with the following string format:

  `urn:ogc:def:coordinateOperation:AUTHORITY::CODE`

  The SRID of the input pose is ignored, and the SRID of the output pose will be set to zero unless a value is provided via the optional `to_srid` parameter. As stated by the last parameter, the pipeline is executed by default in a forward direction; by setting the parameter to false, the pipeline is executed in the inverse direction.

```
SELECT asEWKT(transform(pose 'SRID=4326;Pose(Point(4.35 50.85),1)', 3812), 6);
-- SRID=4326;Pose(POINT(648679.018035 671067.055638),1)
```

```
WITH test(pose, pipeline) AS (
  SELECT pose 'Pose(SRID=4326;Point(4.3525 50.846667),1)',
    text 'urn:ogc:def:coordinateOperation:EPSG::16031' )
SELECT asEWKT(transformPipeline(transformPipeline(pose, pipeline, 4326), pipeline,
  4326, false), 6)
FROM test;
-- SRID=4326;Pose(POINT(4.3525 50.846667),1)
```

11.1.7 Comparisons

The comparison operators (=, <, and so on) are available for poses. 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 poses.

- Traditional comparisons

  `pose {=, <>, <, >, <=, >=} pose`

```
SELECT pose 'Pose(Point(3 3), 0.5)' = pose 'Pose(Point(3 3), 0.5)';
-- true
SELECT pose 'Pose(Point(3 3), 0.5)' <> pose 'Pose(Point(3 3), 0.6)';
-- true
SELECT pose 'Pose(Point(3 3), 0.5)' < pose 'Pose(Point(3 3), 0.6)';
-- true
SELECT pose 'Pose(Point(3 3), 0.6)' > pose 'Pose(Point(2 2), 0.6)';
-- true
SELECT pose 'Pose(Point Z(1 1 1), 0.5, 0.5, 0.5, 0.5)' <= pose 'Pose(Point Z(2 2 2), 0.5,
    0.5, 0.5, 0.5)';
-- true
SELECT pose 'Pose(Point(1 1), 0.6)' >= pose 'Pose(Point(1 1), 0.5)';
-- true
```

- Are the poses approximately equal with respect to an epsilon value?

  `pose ~= pose → boolean`

```
SELECT pose 'Pose(SRID=5676;Point(1 1), 0.3)' ~=
  pose 'Pose(SRID=5676;Point(1 1.0000001), 0.30000001)';
-- true
```

11.2 Temporal Poses

The temporal pose type `tpose` allows to represent the evolution in time of the position of objects and their orientation. As all temporal types, it comes in three subtypes, namely, instant, sequence, and sequence set. Examples of `tpose` values in these subtypes are given next.

```
SELECT tpose 'Pose(Point(1 1), 0.5)@2001-01-01';
SELECT tpose '{Pose(Point(1 1), 0.3)@2001-01-01, Pose(Point(1 1), 0.5)@2001-01-02,
  Pose(Point(1 1), 0.5)@2001-01-03}';
SELECT tpose '[Pose(Point(1 1), 0.2)@2001-01-01, Pose(Point(1 1), 0.4)@2001-01-02,
  Pose(Point(1 1), 0.5)@2001-01-03]';
SELECT tpose '{[Pose(Point(1 1), 1)@2001-01-01, Pose(Point(2 2), 1)@2001-01-02],
  [Pose(Point(2 2), 2)@2001-01-04, Pose(Point(2 2), 3)@2001-01-05]}';
```

As for temporal point types, the temporal pose type accepts type modifiers (or `typmod` in PostgreSQL terminology) to specify the subtype, the dimensionality of the point, and/or the spatial reference identifier (SRID). The possible values for the subtype are `Instant`, `Sequence`, and `SequenceSet` and the possible values for the geometry type are either `Point` or `PointZ`. 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.

```
SELECT asEWKT(tpose(Sequence,5676) 'SRID=5676;[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-03]');
-- SRID=5676;[Pose(POINT(1 1),0.2)@2001-01-01, Pose(POINT(1 1),0.5)@2001-01-03]
SELECT tpose(Sequence) 'Pose(Point(1 1), 0.2)@2001-01-01';
-- ERROR: Temporal type (Instant) does not match column type (Sequence)
SELECT tpose(PointZ) 'Pose(Point(1 1), 0.2)@2001-01-01';
-- Column has Z dimension but the tpose value does not
SELECT tpose(5676) 'Pose(Point(1 1), 0.2)@2001-01-01';
-- ERROR:  Temporal pose SRID (0) does not match column SRID (5676)
```

Temporal pose 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 asText(tpose '[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(2 2), 0.4)@2001-01-02, Pose(Point(3 3), 0.6)@2001-01-03)');
-- [Pose(Point(1 1),0.2)@2001-01-01, Pose(Point(3 3),0.6)@2001-01-03)
SELECT asText(tpose '{[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(2 2), 0.3)@2001-01-02, Pose(Point(2 2), 0.5)@2001-01-03),
  [Pose(Point(2 2), 0.5)@2001-01-03, Pose(Point(2 2), 0.7)@2001-01-04)}');
/* {[Pose(Point(1 1),0.2)@2001-01-01, Pose(Point(2 2),0.3)@2001-01-02,
   Pose(Point(2 2),0.7)@2001-01-04)} */
```

11.3 Validity of Temporal Poses

Temporal pose values must satisfy the constraints specified in Section 4.3 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 tpose 'NULL@2001-01-01 08:05:00';
SELECT tpose 'Point(0 0)@NULL';
-- Base type is not a pose
SELECT tpose 'Point(0 0)@2001-01-01 08:05:00';
```

We give next the functions and operators for temporal poses. Most functions and operators for temporal types and spatiotemporal types described in the previous chapters can be applied for temporal pose types. Therefore, in the signatures of the functions, the type `pose` can be used whenever the types `base` and `spatial` are specified, and the type `tpose` can be used whenever the types `ttype` and `tpatial` are specified. To avoid redundancy, we only present some examples of these functions and operators for temporal poses and we refer to previous chapters for detailed explanations about them.

11.4 Input and Output

- Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation

  `asText({tpose,tpose[]}) → {text,text[]}`

  `asEWKT({tpose,tpose[]}) → {text,text[]}`

```
SELECT asText(tpose 'SRID=4326;[Pose(Point(0 0),1)@2001-01-01,
  Pose(Point(1 1),2)@2001-01-02)');
-- [Pose(Point(0 0),1)@2001-01-01, Pose(Point(1 1),2)@2001-01-02)
SELECT asText(ARRAY[pose 'Pose(Point(0 0),1)', 'Pose(Point(1 1),2)']);
-- {"Pose(POINT(0 0),1)","Pose(POINT(1 1),2)"}
SELECT asEWKT(tpose 'SRID=4326;[Pose(Point(0 0),1)@2001-01-01,
  Pose(Point(1 1),2)@2001-01-02)');
-- SRID=4326;[Pose(Point(0 0),1)@2001-01-01, Pose(Point(1 1),2)@2001-01-02)
```

```
SELECT asEWKT(ARRAY[pose 'Pose(SRID=5676;Point(0 0),1)',
  'Pose(SRID=5676;Point(1 1),2)']);
-- {"Pose(SRID=5676;POINT(0 0),1)","Pose(SRID=5676;POINT(1 1),2))"}
```

- Return the Moving Features JSON (MF-JSON) representation

  `asMFJSON(tpose) → text`

```
SELECT asMFJSON(tpose 'Pose(Point(1 2),0.5)@2001-01-01', 3);
/* {"type":"MovingPose","bbox":[[1,2],[1,2]],"period":{"begin":"2001-01-01T00:00:00+01",
  "end":"2001-01-01T00:00:00+01","lower_inc":true,"upper_inc":true},
  "values":[{"position":{"lat":2,"lon":1},"rotation":0.5}],
  "datetimes":["2001-01-01T00:00:00+01"],"interpolation":"None"} */
SELECT asMFJSON(tpose 'Pose(Point Z(1 2 3),1,0,0,0)@2001-01-01', 3);
/* {"type":"MovingPose","bbox":[[1,2,3],[1,2,3]],
  "period":{"begin":"2001-01-01T00:00:00+01","end":"2001-01-01T00:00:00+01",
  "lower_inc":true,"upper_inc":true},
  "values":[{"position":{"lat":2,"lon":1,"h":3},"quaternion":{"x":0,"y":0,"z":0,"w":1}}],
  "datetimes":["2001-01-01T00:00:00+01"],"interpolation":"None"} */
```

- Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB) representation, or the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `asBinary(tpose,endian text='') → bytea`

  `asEWKB(tpose,endian text='') → bytea`

  `asHexEWKB(tpose,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 the encoding of the machine is used.

```
SELECT asBinary(tpose 'Pose(Point(1 2),1)@2001-01-01');
-- \x013a0001000000000000f03f000000000000004000000000000000f03f009c57d3c11c0000
SELECT asEWKB(tpose 'SRID=7844;Pose(Point(1 2),1)@2001-01-01');
-- \x013a0001000000000000f03f000000000000004000000000000000f03f009c57d3c11c0000
SELECT asHexEWKB(tpose 'SRID=3812;Pose(Point(1 2),1)@2001-01-01');
-- 013A0001000000000000F03F000000000000004000000000000000F03F009C57D3C11C0000
```

- Input from the Well-Known Text (WKT) representation or from the Extended Well-Known Text (EWKT) representation

  `tposeFromText(text) → tpose`

  `tposeFromEWKT(text) → tpose`

```
SELECT asEWKT(tposeFromText(text '[Pose(Point(1 2),1)@2001-01-01,
  Pose(Point(3 4),2)@2001-01-02]'));
-- [Pose(POINT(1 2),1)@2001-01-01, Pose(POINT(3 4),2)@2001-01-02]
SELECT asEWKT(tposeFromEWKT(text 'SRID=3812;[Pose(Point(1 2),1)@2001-01-01,
  Pose(Point(3 4),2)@2001-01-02]'));
-- SRID=3812;[Pose(Point(1 2),1)@2001-01-01, Pose(Point(3 4),2)@2001-01-02]
```

- Input from the Moving Features JSON (MF-JSON) representation

  `tposeFromMFJSON(bytea) → tpose`

```
SELECT asEWKT(tposeFromMFJSON(asMFJSON(tpose
  'SRID=3812;Pose(Point(1 2),0.5)@2001-01-01', 3)));
-- SRID=3812;Pose(POINT(1 2),0.5)@2001-01-01
SELECT asEWKT(tposeFromMFJSON(asMFJSON(tpose
  'SRID=5676;Pose(Point Z(1 2 3),1,0,0,0)@2001-01-01', 3)));
-- SRID=5676;Pose(POINT Z(1 2 3),1,0,0,0)@2001-01-01
```

- Input from the Well-Known Binary (WKB) representation, from the Extended Well-Known Binary (EWKB) representation, or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `tposeFromBinary(bytea) → tpose`

  `tposeFromEWKB(bytea) → tpose`

  `tposeFromHexEWKB(text) → tpose`

```
SELECT asEWKT(tposeFromBinary(
  '\x013a0001000000000000f03f000000000000000400000000000000f03f009c57d3c11c0000'));
-- Pose(POINT(1 2),1)@2001-01-01
SELECT asEWKT(tposeFromEWKB(
  '\x013a0001000000000000f03f000000000000000400000000000000f03f009c57d3c11c0000'));
-- SRID=7844;Pose(Point(1 1),2)@2001-01-01
SELECT asEWKT(tposeFromHexEWKB(
  '013A0041E40E0000E40E000000000000000F03F...40000000000000F03F009C57D3C11C0000'));
-- SRID=3812;Pose(POINT(1 2),1)@2001-01-01
```

11.5 Constructors

- Constructor for temporal poses having a constant value

  `tpose(pose,timestamptz) → tposeInst`

  `tpose(pose,tstzset) → tposeDiscSeq`

  `tpose(pose,tstzspan,interp='linear') → tposeContSeq`

  `tpose(pose,tstzspanset,interp='linear') → tposeSeqSet`

```
SELECT asText(tpose('Pose(Point(1 1), 0.5)', timestamptz '2001-01-01'));
-- Pose(Point(1 1),0.5)@2001-01-01
SELECT asText(tpose('Pose(Point Z(1 1 1), 0.5, 0.5, 0.5, 0.5)',
  tstzset '{2001-01-01, 2001-01-05}'));
/* {Pose(POINT Z (1 1 1),0.5,0.5,0.5,0.5)@2001-01-01,
  Pose(POINT Z (1 1 1),0.5,0.5,0.5,0.5)@2001-01-05} */
SELECT asText(tpose('Pose(Point(1 1), 0.5)',
  tstzspan '[2001-01-01, 2001-01-02]'));
-- [Pose(Point(1 1),0.5)@2001-01-01, Pose(Point(1 1),0.5)@2001-01-02]
SELECT asText(tpose('Pose(Point(1 1), 0.2)',
  tstzspanset '{[2001-01-01, 2001-01-03]}', 'step'));
-- Interp=Step;{[Pose(Point(1 1),0.2)@2001-01-01, Pose(Point(1 1),0.2)@2001-01-03]}
```

- Constructor for temporal poses of sequence subtype

  `tposeSeq(tposeInst[],interp={'step','linear'},leftInc bool=true,`
  `rightInc bool=true) → tposeSeq`

```
SELECT asText(tposeSeq(ARRAY[tpose 'Pose(Point(1 1), 0.3)@2001-01-01',
  'Pose(Point(2 2), 0.5)@2001-01-02', 'Pose(Point(1 1), 0.5)@2001-01-03']));
/* {Pose(Point(1 1),0.3)@2001-01-01, Pose(Point(2 2),0.5)@2001-01-02,
   Pose(Point(1 1),0.5)@2001-01-03} */
SELECT asText(tposeSeq(ARRAY[tpose 'Pose(Point(1 1), 0.2)@2001-01-01',
  'Pose(Point(1 1), 0.4)@2001-01-02', 'Pose(Point(1 1), 0.5)@2001-01-03']));
/* [Pose(Point(1 1),0.2)@2001-01-01, Pose(Point(1 1),0.4)@2001-01-02,
   Pose(Point(1 1),0.5)@2001-01-03] */
```

- Constructor for temporal poses of sequence set subtype

  `tposeSeqSet(tpose[]) → tposeSeqSet`

  `tposeSeqSetGaps(tposeInst[],maxt=NULL,maxdist=NULL,interp='linear') → tposeSeqSet`

```
SELECT asText(tposeSeqSet(ARRAY[tpose
  '[Pose(Point(1 1),0.2)@2001-01-01, Pose(Point(2 2),0.4)@2001-01-02]',
  '[Pose(Point(2 2),0.6)@2001-01-03, Pose(Point(2 2),0.8)@2001-01-04]']));
/* {[Pose(Point(1 1),0.2)@2001-01-01, Pose(Point(2 2),0.4)@2001-01-02],
   [Pose(Point(2 2),0.6)@2001-01-03, Pose(Point(2 2),0.6)@2001-01-04]} */
SELECT asText(tposeSeqSetGaps(ARRAY[tpose 'Pose(Point(1 1),0.1)@2001-01-01',
  'Pose(Point(1 1),0.3)@2001-01-03', 'Pose(Point(1 1),0.5)@2001-01-05'], '1 day'));
/* {[Pose(Point(1 1),0.1)@2001-01-01], [Pose(Point(1 1),0.3)@2001-01-03],
   [Pose(Point(1 1),0.5)@2001-01-05]} */
```

- Construct a 2D temporal pose from a temporal geometry point and a temporal float

  `tpose(tgeompoint, tfloat) → tpose`

  The time frame of the result is the intersection of the time frames of the temporal point and the temporal float. If they do not intersect, a null value is returned

```
SELECT asText(tpose(tgeompoint '[POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02)',
  tfloat '[1@2001-01-01, 2@2001-01-02)'));
-- [Pose(Point(1 1),1)@2001-01-01, Pose(Point(1 1),2)@2001-01-02)
SELECT asText(tpose(tgeompoint '[POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02)',
  tfloat '[2@2001-01-03, 3@2001-01-04)'));
-- NULL
```

11.6 Conversions

A temporal pose 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 `pose` value.

- Convert a temporal pose into a temporal geometry point

  `tpose::tgeompoint`

```
SELECT asText((tpose '[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(1 1), 0.3)@2001-01-02)')::tgeompoint);
-- [POINT(1 1)@2001-01-01, POINT(1 1)@2001-01-02)
SELECT asEWKT((tpose '[Pose(Point Z(1 1 1), 0.5, 0.5, 0.5, 0.5)@2001-01-01,
  Pose(Point Z(2 2 2), 1, 0, 0, 0)@2001-01-02)')::tgeompoint);
-- [POINT Z (1 1 1)@2001-01-01, POINT Z (2 2 2)@2001-01-02)
```

11.7 Accessors

- Return the values

  `getValues(tpose) → poseset`

```
SELECT asText(getValues(tpose '{[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-02)}'));
-- {"Pose(Point(1 1),0.3)","Pose(Point(1 1),0.5)"}
SELECT asEWKT(getValues(tpose 'SRID=5676;{[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.3)@2001-01-02)}'));
-- SRID=5676;{"Pose(Point(1 1),0.3)"}
```

- Return the points

  `points(tpose) → geomset`

```
SELECT asEWKT(points(tpose 'SRID=5676;{Pose(Point(3 3), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-02}'));
-- SRID=5676;{Point(1 1), Point(3 3)}
```

- Return the rotation

  `rotation(tpose2d) → tfloat`

```
SELECT rotation(tpose 'SRID=5676;{[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.3)@2001-01-02)}');
-- {[0.3@2001-01-01, 0.3@2001-01-02)}
```

- Return the value at a timestamp

  `valueAtTimestamp(tpose,timestamptz) → pose`

```
SELECT asText(valueAtTimestamp(tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(3 3), 0.5)@2001-01-03)', '2001-01-02'));
-- Pose(Point(2 2),0.4)
SELECT asText(valueAtTimestamp(tpose
  '[Pose(Point Z(1 1 1), 0.5, 0.5, 0.5, 0.5)@2001-01-01,
    Pose(Point Z(3 3 3), 1, 0, 0, 0)@2001-01-03)', timestamptz '2001-01-02'), 6);
-- Pose(POINT Z (2 2 2),0.866025,0.288675,0.288675,0.288675)
```

11.8 Transformations

- Transform to another subtype

  `tposeInst(tpose) → tposeInst`

  `tposeSeq(tpose) → tposeSeq`

  `tposeSeqSet(tpose) → tposeSeqSet`

```
SELECT asText(tposeSeq(tpose 'Pose(Point(1 1), 0.5)@2001-01-01', 'discrete'));
-- {Pose(Point(1 1),0.5)@2001-01-01}
SELECT asText(tposeSeq(tpose 'Pose(Point(1 1), 0.5)@2001-01-01'));
-- [Pose(Point(1 1),0.5)@2001-01-01]
SELECT asText(tposeSeqSet(tpose 'Pose(PointZ(1 1 1), 0.5, 0.5, 0.5, 0.5)@2001-01-01'));
-- {[Pose(POINT Z (1 1 1),0.5,0.5,0.5,0.5)@2001-01-01]}
```

- Transform to another interpolation

  `setInterp(tpose, interp) → tpose`

```
SELECT asText(setInterp(tpose 'Pose(Point(1 1),0.2)@2001-01-01','linear'));
-- [Pose(Point(1 1),0.2)@2001-01-01]
SELECT asText(setInterp(tpose '{[Pose(Point Z(1 1 1),1,0,0,0)@2001-01-01],
  [Pose(Point Z(2 2 2),0,0,0,1)@2001-01-02]}', 'discrete'));
-- {Pose(POINT Z (1 1 1),1,0,0,0)@2001-01-01, Pose(POINT Z (2 2 2),0,0,0,1)@2001-01-02}
```

- Round the points and the orientations of the temporal pose to the number of decimal places

  `round(tpose,integer=0) → tpose`

```
SELECT asText(round(tpose '{[Pose(Point(1 1.123456789),0.123456789)@2001-01-01,
  Pose(Point(1 1),0.5)@2001-01-02)}', 3));
-- {[Pose(POINT(1 1.123),0.123)@2001-01-01, Pose(POINT(1 1),0.5)@2001-01-02)}
```

11.9 Modifications

- Append a temporal instant or a temporal sequence

  `appendInstant(tpose,tposeInst) → tpose`

  `appendSequence(tpose,tposeSeq) → tpose`

```
SELECT asText(appendInstant(tpose 'Pose(Point(1 1), 0.5)@2001-01-01', 'Pose(Point(2 2), 1) ↩
    @2001-01-02'));
-- [Pose(POINT(1 1),0.5)@2001-01-01, Pose(POINT(2 2),1)@2001-01-02]
SELECT asText(appendSequence(tpose '[Pose(Point(1 1), 0.5)@2001-01-01]', '[Pose(Point(2 2) ↩
    , 1)@2001-01-02]'));
-- {[Pose(POINT(1 1),0.5)@2001-01-01], [Pose(POINT(2 2),1)@2001-01-02]}
```

- Merge the temporal poses

  `merge(tpose,tpose) → tpose`

  `merge(tpose[]) → tpose`

  `merge(tpose) → tpose`

```
SELECT asText(merge(tpose
  '[Pose(Point(1 1 1),1,0,0,0)@2001-01-01, Pose(Point(2 2 2),0,1,0,0)@2001-01-02]',
  '[Pose(Point(2 2 2),0,1,0,0)@2001-01-02, Pose(Point(3 3 3),0,0,0,1)@2001-01-03]'));
/* [Pose(POINT Z (1 1 1),1,0,0,0)@2001-01-01, Pose(POINT Z (2 2 2),0,1,0,0)@2001-01-02,
    Pose(POINT Z (3 3 3),0,0,0,1)@2001-01-03] */
SELECT asText(merge(ARRAY[tpose
  '{[Pose(Point(1 1),0.1)@2001-01-01, Pose(Point(2 2),0.2)@2001-01-02],
    [Pose(Point(3 3),0.3)@2001-01-03, Pose(Point(4 4),0.4)@2001-01-04]}',
  '[Pose(Point(4 4),0.4)@2001-01-04, Pose(Point(5 5),0.5)@2001-01-05]']));
/* {[Pose(POINT(1 1),0.1)@2001-01-01, Pose(POINT(2 2),0.2)@2001-01-02],
    [Pose(POINT(3 3),0.3)@2001-01-03, Pose(POINT(5 5),0.5)@2001-01-05]} */
WITH temp(inst) AS (
  SELECT tpose 'Pose(Point(1 1),0.1)@2001-01-01' UNION
  SELECT tpose 'Pose(Point(2 2),0.2)@2001-01-02' UNION
  SELECT tpose 'Pose(Point(3 3),0.3)@2001-01-03' )
SELECT asText(merge(inst)) FROM temp;
/* {Pose(POINT(1 1),0.1)@2001-01-01, Pose(POINT(2 2),0.2)@2001-01-02,
    Pose(POINT(3 3),0.3)@2001-01-03} */
```

11.10 Restrictions

- TODO Restrict to (the complement of) a set of values

  `atValues(tpose,values) → tpose`

  `minusValues(tpose,values) → tpose`

```
SELECT atValues(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-03]', 'Pose(Point(2 2), 0.5)');
-- {[Pose(Point(2 2),0.5)@2001-01-02]}
SELECT minusValues(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-03]', 'Pose(Point(2 2), 0.5)');
/* {[Pose(Point(2 2),0.3)@2001-01-01, Pose(Point(2 2),0.5)@2001-01-02),
   (Pose(Point(2 2),0.5)@2001-01-02, Pose(Point(2 2),0.7)@2001-01-03]} */
```

- TODO Restrict to (the complement of) a geometry

  `atGeometry(tpose,geometry) → tpose`

  `minusGeometry(tpose,geometry) → tpose`

```
SELECT atGeometry(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-03]',
  'Polygon((40 40,40 50,50 50,50 40,40 40))');
SELECT minusGeometry(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-03]',
  'Polygon((40 40,40 50,50 50,50 40,40 40))');
/* {(Pose(Point(2 2),0.342593)@2001-01-01 05:06:40.364673+01,
   Pose(Point(2 2),0.7)@2001-01-03]} */
```

11.11 Spatial Reference System

- Return or set the spatial reference identifier

  `SRID(tpose) → integer`

  `setSRID(tpose) → tpose`

```
SELECT SRID(tpose 'SRID=5676;[Pose(Point(0 0),1)@2001-01-01,
  Pose(Point(1 1),2)@2001-01-02)'));
-- 5676
SELECT asEWKT(setSRID(tpose '[Pose(Point(0 0),1)@2001-01-01,
  Pose(Point(1 1),2)@2001-01-02)', 5676));
-- SRID=5676;[Pose(POINT(0 0),1)@2001-01-01, Pose(POINT(1 1),2)@2001-01-02)
```

- Transform to a spatial reference identifier

  `transform(tpose,integer) → tpose`

  `transformPipeline(tpose,pipeline text,to_srid integer,is_forward bool=true) → tpose`

```
SELECT asEWKT(transform(tpose 'SRID=4326;Pose(Point(4.35 50.85),1)@2001-01-01',
  3812));
-- SRID=3812;Pose(POINT(648679.0180353033 671067.0556381135),1)@2001-01-01
```

```
WITH test(tpose, pipeline) AS (
  SELECT tpose 'SRID=4326;{Pose(Point(4.3525 50.846667),1)@2001-01-01,
    Pose(Point(-0.1275 51.507222),2)@2001-01-02}',
    text 'urn:ogc:def:coordinateOperation:EPSG::16031' )
SELECT asEWKT(transformPipeline(transformPipeline(tpose, pipeline, 4326),  pipeline,
  4326, false), 6)
FROM test;
/* SRID=4326;{Pose(POINT(4.3525 50.846667),1)@2001-01-01,
   Pose(POINT(-0.1275 51.507222),2)@2001-01-02} */
```

11.12 Bounding Box Operations

- Topological operators

  `{tstzspan,stbox,tpose} {&&, <@, @>, ~=, -|-} {tstzspan,stbox,tpose} → boolean`

```
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-03]' && tstzspan '[2001-01-02,2001-01-04]';
-- true
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-02]' @> stbox(pose 'Pose(Point(1 1), 0.5)');
-- true
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-03]' ~= tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.35)@2001-01-02, Pose(Point(1 1), 0.5)@2001-01-03]';
-- true
```

- Position operators

  `{stbox,tpose} {<<, &<, >>, &>} {stbox,tpose} → boolean`

  `{stbox,tpose} {<<|, &<|, |>>, |&>} {stbox,tpose} → boolean`

  `{stbox,tpose} {<</, &</, />>, /&>} {stbox,tpose} → boolean`

  `{tstzspan,stbox,tpose} {<<#, &<#, #>>, #&>} {tstzspan,stbox,tpose} → boolean`

```
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-02]' << stbox(pose 'Pose(Point(2 2), 0.5)');
-- true
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-02]' <<| stbox(pose 'Pose(Point(2 2), 0.5)');
-- true
SELECT tpose '[Pose(Point(1 1 1), 1,0,0,0)@2001-01-01,
  Pose(Point(1 1 1), 1,0,0,0)@2001-01-02]' <</ stbox(pose 'Pose(Point(2 2 2), 1,0,0,0)');
-- true
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-02]' <<# tstzspan '[2001-01-03, 2001-01-04]';
-- true
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-03,
  Pose(Point(1 1), 0.5)@2001-01-05]' #>>
  tpose '[Pose(Point(1 1), 0.3)@2001-01-01, Pose(Point(1 1), 0.5)@2001-01-02]';
-- true
```

11.13 Distance Operations

We present next the functions that compute distance operations for temporal poses. Notice that for these operations only the point component is considered, not the orientation.

- Return the smallest distance ever

  `{geo,pose,tpose} |=| {geo,pose,tpose} → float`

```
SELECT tpose '[Pose(Point(2 2), 0.3)@2001-01-01, Pose(Point(2 2), 0.7)@2001-01-02]' |=|
  geometry 'Linestring(50 50,55 55)';
-- 67.88225099390856
SELECT tpose '[Pose(Point(2 2), 0.3)@2001-01-01, Pose(Point(2 2), 0.7)@2001-01-02]' |=|
  pose 'Pose(Point(1 1), 0.5)';
-- 1.4142135623730951
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-03]' |=| pose 'Pose(Point(2 2), 0.2)';
-- 1.4142135623730951
SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-03]' |=| geometry 'Linestring(2 2,2 1,3 1)';
-- 1
```

  The operator `|=|` can be used for doing nearest neighbor searches using GiST or SP-GiST indexes (see Section 10.2).

- Return the instant of the first temporal pose at which the two arguments are at the nearest distance

  `nearestApproachInstant({geo,pose,tpose},{geo,pose,tpose}) → tpose`

```
SELECT asText(nearestApproachInstant(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-02]', geometry 'Linestring(50 50,55 55)'));
-- Pose(POINT(2 2),0.3)@2001-01-01
SELECT asText(nearestApproachInstant(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-02]', pose 'Pose(Point(1 1), 0.5)'));
-- Pose(POINT(2 2),0.3)@2001-01-01
```

- Return the line connecting the nearest approach point

  `shortestLine({geo,pose,tpose},{geo,pose,tpose}) → geometry`

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

```
SELECT ST_AsText(shortestLine(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-02]', geometry 'Linestring(50 50,55 55)'));
-- LINESTRING(2 2,50 50)
SELECT ST_AsText(shortestLine(tpose '[Pose(Point(2 2), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.7)@2001-01-02]', pose 'Pose(Point(1 1), 0.5)'));
-- LINESTRING(2 2,1 1)
```

- Return the temporal distance

  `tpose <-> tpose → tfloat`

```
SELECT round(tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-03]' <-> pose 'Pose(Point(2 2), 0.2)', 6);
-- [1.414214@2001-01-01, 1.414214@2001-01-03]
SELECT round(tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(1 1), 0.5)@2001-01-03]' <-> geometry 'Point(50 50)', 6);
-- [69.296465@2001-01-01, 69.296465@2001-01-03]
SELECT round(tpose '[Pose(Point(1 1), 0.3)@2001-01-01,
  Pose(Point(2 2), 0.5)@2001-01-03]' <->
  tpose '[Pose(Point(1 2), 0.3)@2001-01-02, Pose(Point(2 1), 0.5)@2001-01-04]', 6);
-- [0.707107@2001-01-02, 0.5@2001-01-02 12:00:00+01, 0.707107@2001-01-03]
```

11.14 Spatial Relationships

The topological and distance relationships described in Section 8.5 such as `eIntersects`, `aDwithin`, or `tContains` only consider the location of the geometries, not their orientation. To be able to apply these spatial relationships to the temporal poses, they must be transformed to temporal geometry points. This can be easily performed using the functions `geometry` and `tgeompoint` or using an explicit casting `::` as shown in the examples below.

- Ever and always relationships

```
SELECT eContains(geometry 'Polygon((0 0,0 50,50 50,50 0,0 0))',
  tgeompoint(tpose '[Pose(Point(1 1),0.1)@2001-01-01, Pose(Point(1 1),0.3)@2001-01-03)'));
-- true
SELECT eDisjoint(geometry(pose 'Pose(Point(2 2), 0.0)'),
  tgeompoint(tpose '[Pose(Point(1 1),0.1)@2001-01-01, Pose(Point(1 1),0.3)@2001-01-03)'));
-- true
SELECT eIntersects(
  tpose '[Pose(Point(1 1),0.1)@2001-01-01, Pose(Point(1 1),0.3)@2001-01-03)'::tgeompoint,
  tpose '[Pose(Point(2 2),0.0)@2001-01-01, Pose(Point(2 2),1)@2001-01-03)'::tgeompoint);
-- false
```

- Spatiotemporal relationships

```
SELECT tContains(geometry 'Polygon((0 0,0 50,50 50,50 0,0 0))',
  tgeompoint(tpose '[Pose(Point(1 1),0.1)@2001-01-01, Pose(Point(1 1),0.3)@2001-01-03)'));
-- {[t@2001-01-01 00:00:00+01, t@2001-01-03 00:00:00+01)}
SELECT tDisjoint(geometry(pose 'Pose(Point(2 2), 0.0)'),
  tgeompoint(tpose '[Pose(Point(1 1),0.1)@2001-01-01, Pose(Point(1 1),0.3)@2001-01-03)'));
-- [t@2001-01-01, t@2001-01-03)
SELECT tDwithin(
 tpose '[Pose(Point(1 1),0.3)@2001-01-01,Pose(Point(1 1),0.5)@2001-01-03)'::tgeompoint,
 tpose '[Pose(Point(1 1),0.5)@2001-01-01,Pose(Point(3 3),0.3)@2001-01-03)'::tgeompoint,1);
/* {[t@2001-01-01 00:00:00+01, t@2001-01-01 16:58:14.025894+01],
  (f@2001-01-01 16:58:14.025894+01, f@2001-01-03 00:00:00+01)} */
```

11.15 Comparisons

- Traditional comparisons

`tpose {=, <>, <, >, <=, >=} tpose → boolean`

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

- Ever and always comparisons

`tpose {?=, %=} tpose → boolean`

```
SELECT tpose '[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(1 1), 0.4)@2001-01-04)' ?= Pose(Point(1 1), 0.3);
-- true
SELECT tpose '[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(1 1), 0.2)@2001-01-04)' &= Pose(Point(1 1), 0.2);
-- true
```

- Temporal comparisons

`tpose {#=, #<>} tpose → tbool`

```
SELECT tpose '[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(1 1), 0.4)@2001-01-03)' #= pose 'Pose(Point(1 1), 0.3)';
-- {[f@2001-01-01, t@2001-01-02], (f@2001-01-02, f@2001-01-03)}
SELECT tpose '[Pose(Point(1 1), 0.2)@2001-01-01,
  Pose(Point(1 1), 0.8)@2001-01-03)' #<>
  tpose '[Pose(Point(1 1), 0.3)@2001-01-01, Pose(Point(1 1), 0.7)@2001-01-03)';
-- {[t@2001-01-01, f@2001-01-02], (t@2001-01-02, t@2001-01-03)}
```

11.16 Aggregations

The three aggregate functions for temporal poses are illustrated next.

- Temporal count

`tCount(tpose) → {tintSeq,tintSeqSet}`

```
WITH Temp(temp) AS (
  SELECT tpose '[Pose(Point(1 1), 0.1)@2001-01-01,
    Pose(Point(1 1), 0.3)@2001-01-03)' UNION
  SELECT tpose '[Pose(Point(1 1), 0.2)@2001-01-02,
    Pose(Point(1 1), 0.4)@2001-01-04)' UNION
  SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-03,
    Pose(Point(1 1), 0.5)@2001-01-05)' )
SELECT tCount(Temp)
FROM Temp;
-- {[1@2001-01-01, 2@2001-01-02, 1@2001-01-04, 1@2001-01-05)}
```

- Window count

  `wCount(tpose) → {tintSeq,tintSeqSet}`

```
WITH Temp(temp) AS (
  SELECT tpose '[Pose(Point(1 1), 0.1)@2001-01-01,
    Pose(Point(1 1), 0.3)@2001-01-03)' UNION
  SELECT tpose '[Pose(Point(1 1), 0.2)@2001-01-02,
    Pose(Point(1 1), 0.4)@2001-01-04)' UNION
  SELECT tpose '[Pose(Point(1 1), 0.3)@2001-01-03,
    Pose(Point(1 1), 0.5)@2001-01-05)' )
SELECT wCount(Temp, '1 day')
FROM Temp;
/* {[1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 2@2001-01-04, 1@2001-01-05,
   1@2001-01-06)} */
```

11.17 Indexing

GiST and SP-GiST indexes can be created for table columns of temporal poses. 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 poses, which is an `stbox`, as all spatiotemporal types.

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

- `<<`, `&<`, `&>`, `>>`, `<<|`, `&<|`, `|&>`, `|>>`, which only consider the spatial dimension in temporal poses,
- `<<#`, `&<#`, `#&>`, `#>>`, which only consider the time dimension in temporal poses,
- `&&`, `@>`, `<@`, `~=`, `-|-`, 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.

Chapter 12

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.

12.1 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` $\leq$ `endPosition`. Semantically, a network segment represents a set of network points `(rid,position)` with `startPosition` $\leq$ `position` $\leq$ `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` $\leq$ `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.

12.1.1 Constructors

- Constructors for network points and network segments

  `npoint(bigint,float)` $\rightarrow$ `npoint`

  `nsegment(bigint,float,float)` $\rightarrow$ `nsegment`

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

12.1.2 Conversions

Values of the `npoint` and `nsegment` types can be converted to the `geometry` type using an explicit `CAST` or using the `::` notation as shown below. Similarly, `geometry` values of subtype `point` or `linestring` (restricted to two points) can be converted, respectively, to `npoint` and `nsegment` values. 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.

- Convert between a network point or segment and a geometry

  `{npoint,nsegment}::geometry`

  `geometry::{npoint,nsegment}`

```
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)
```

```
SELECT geometry 'SRID=5676;Point(279.269156511873 811.497076880187)'::npoint;
-- NPoint(3,0.781413)
SELECT geometry 'SRID=5676;LINESTRING(406.729536784738 702.58583437902,
  383.570801314823 845.137059419277)'::nsegment;
-- NSegment(3,0.6,0.9)
SELECT geometry 'SRID=5676;Point(279.3 811.5)'::npoint;
-- NULL
SELECT geometry 'SRID=5676;LINESTRING(406.7 702.6,383.6 845.1)'::nsegment;
-- NULL
```

- Construct a spatiotemporal box from a network point and, optionally, a timestamp or a period

  `stbox(npoint) → stbox`

  `stbox(npoint,{timestamptz,tstzspan}) → stbox`

```
SELECT stbox(npoint 'NPoint(1,0.3)');
-- STBOX X((48.711754,20.92568),(48.711758,20.925682))
SELECT stbox(npoint 'NPoint(1,0.3)', timestamptz '2001-01-01');
-- STBOX XT(((62.786633,80.143555),(62.786636,80.143562)),[2001-01-01,2001-01-01])
SELECT stbox(npoint 'NPoint(1,0.3)', tstzspan '[2001-01-01,2001-01-02]');
-- STBOX XT(((62.786633,80.143555),(62.786636,80.143562)),[2001-01-01,2001-01-02])
```

12.1.3 Accessors

- Return 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
```

- Return the position

  `getPosition(npoint) → float`

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

- Return the start/end position

  `startPosition(nsegment) → float`

  `endPosition(nsegment) → float`

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

12.1.4 Transformations

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

  `round({npoint,nsegment},integer=0) → {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)
```

12.1.5 Spatial Operations

- Return the spatial reference identifier

  `SRID({npoint,nsegment}) → integer`

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

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

- Spatial similarity

  `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
```

12.1.6 Comparisons

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.

- Traditional comparisons

  `npoint {=, <>, <, >, <=, >=} npoint`

  `nsegment {=, <>, <, >, <=, >=} 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, 0.5)';
-- false
SELECT nsegment 'Nsegment(3, 0.5, 0.5)' < nsegment 'Nsegment(3, 0.5, 0.6)';
-- true
SELECT nsegment 'Nsegment(3, 0.5, 0.5)' > nsegment 'Nsegment(2, 0.5, 0.5)';
-- true
SELECT npoint 'Npoint(1, 0.5)' <= npoint 'Npoint(2, 0.5)';
-- true
SELECT npoint 'Npoint(1, 0.6)' >= npoint 'Npoint(1, 0.5)';
-- true
```

12.2 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 three subtypes, namely, instant, sequence, and sequence set. Examples of `tnpoint` values in these subtypes are given next.

```
SELECT tnpoint 'Npoint(1, 0.5)@2001-01-01';
SELECT tnpoint '{Npoint(1, 0.3)@2001-01-01, Npoint(1, 0.5)@2001-01-02,
  Npoint(1, 0.5)@2001-01-03}';
SELECT tnpoint '[Npoint(1, 0.2)@2001-01-01, Npoint(1, 0.4)@2001-01-02,
  Npoint(1, 0.5)@2001-01-03]';
SELECT tnpoint '{[Npoint(1, 0.2)@2001-01-01, Npoint(1, 0.4)@2001-01-02,
  Npoint(1, 0.5)@2001-01-03], [Npoint(2, 0.6)@2001-01-04, Npoint(2, 0.6)@2001-01-05]}';
```

The temporal network point type accepts type modifiers (or `typmod` in PostgreSQL terminology). The possible values for the type modifier are `Instant`, `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)@2001-01-01, Npoint(1, 0.4)@2001-01-02,
  Npoint(1, 0.5)@2001-01-03]';
SELECT tnpoint(Sequence) 'Npoint(1, 0.2)@2001-01-01';
-- ERROR: Temporal type (Instant) does not match column type (Sequence)
```

Temporal network point values of sequence subtype and linear or step interpolation 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)}
```

12.3 Validity of Temporal Network Points

Temporal network point values must satisfy the constraints specified in Section 4.3 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 continuous sequence
SELECT tnpoint '[Npoint(1, 0.2)@2001-01-01 09:00:00, Npoint(2, 0.2)@2001-01-01 09:05:00)';
```

We present next the operation for temporal network point types. Most functions for temporal types described in the previous chapters 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.

12.4 Constructors

- Constructor for temporal network points having a constant value

  `tnpoint(npoint,timestamptz)` → `tnpointInst`

  `tnpoint(npoint,tstzset)` → `tnpointDiscSeq`

  `tnpoint(npoint,tstzspan,interp='linear')` → `tnpointContSeq`

  `tnpoint(npoint,tstzspanset,interp='linear')` → `tnpointSeqSet`

```
SELECT tnpoint('Npoint(1, 0.5)', timestamptz '2001-01-01');
-- NPoint(1,0.5)@2001-01-01
SELECT tnpointSeq('Npoint(1, 0.3)', tstzset '{2001-01-01, 2001-01-03, 2001-01-05}');
-- {NPoint(1,0.3)@2001-01-01, NPoint(1,0.3)@2001-01-03, NPoint(1,0.3)@2001-01-05}
SELECT tnpointSeq('Npoint(1, 0.5)', tstzspan '[2001-01-01, 2001-01-02]');
-- [NPoint(1,0.5)@2001-01-01, NPoint(1,0.5)@2001-01-02]
SELECT tnpointSeqSet('Npoint(1, 0.2)', tstzspanset '{[2001-01-01, 2001-01-03]}', 'step');
-- Interp=Step;{[NPoint(1,0.2)@2001-01-01, NPoint(1,0.2)@2001-01-03]}
```

- Constructor for temporal network points of sequence subtype

  `tnpointSeq(tnpointInst[],interp={'step','linear'},leftInc bool=true,`
  `rightInc bool=true)` →`tnpointSeq`

```
SELECT tnpointSeq(ARRAY[tnpoint 'Npoint(1, 0.3)@2001-01-01',
  'Npoint(1, 0.5)@2001-01-02', 'Npoint(1, 0.5)@2001-01-03']);
-- {NPoint(1,0.3)@2001-01-01, NPoint(1,0.5)@2001-01-02, NPoint(1,0.5)@2001-01-03}
SELECT tnpointSeq(ARRAY[tnpoint 'Npoint(1, 0.2)@2001-01-01',
  'Npoint(1, 0.4)@2001-01-02', 'Npoint(1, 0.5)@2001-01-03']);
-- [NPoint(1,0.2)@2001-01-01, NPoint(1,0.4)@2001-01-02, NPoint(1,0.5)@2001-01-03]
```

- Constructor for temporal network points of sequence set subtype

  `tnpointSeqset(tnpoint[])` → `tnpointSeqSet`

  `tnpointSeqSetGaps(tnpointInst[],maxt=NULL,maxdist=NULL,interp='linear')` →
  `tnpointSeqSet`

```
SELECT tnpointSeqSet(ARRAY[tnpoint '[Npoint(1,0.2)@2001-01-01, Npoint(1,0.4)@2001-01-02,
  Npoint(1,0.5)@2001-01-03]', '[Npoint(2,0.6)@2001-01-04, Npoint(2,0.6)@2001-01-05]']);
/* {[NPoint(1,0.2)@2001-01-01, NPoint(1,0.4)@2001-01-02, NPoint(1,0.5)@2001-01-03],
   [NPoint(2,0.6)@2001-01-04, NPoint(2,0.6)@2001-01-05]} */
SELECT tnpointSeqSetGaps(ARRAY[tnpoint 'NPoint(1,0.1)@2001-01-01',
  'NPoint(1,0.3)@2001-01-03', 'NPoint(1,0.5)@2001-01-05'], '1 day');
-- {[NPoint(1,0.1)@2001-01-01], [NPoint(1,0.3)@2001-01-03], [NPoint(1,0.5)@2001-01-05]}
```

12.5 Conversions

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.

- Convert between a temporal network point and a temporal geometry point

  `tnpoint::tgeompoint`

  `tgeompoint::tnpoint`

```
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) */
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
```

12.6 Accessors

- Return the values

  `getValues(tnpoint) → npointset`

```
SELECT getValues(tnpoint '{[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02)}');
-- {"NPoint(1,0.3)","NPoint(1,0.5)"}
SELECT getValues(tnpoint '{[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.3)@2001-01-02)}');
-- {"NPoint(1,0.3)"}
```

- Return the road identifiers

  `routes(tnpoint) → bigintset`

```
SELECT routes(tnpoint '{NPoint(3, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02}');
-- {1, 3}
```

- Return the value at a timestamp

  `valueAtTimestamp(tnpoint,timestamptz) → npoint`

```
SELECT valueAtTimestamp(tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-03)',
  '2001-01-02');
-- NPoint(1,0.4)
```

- Return the length traversed by the temporal network point

  `length(tnpoint) → float`

```
SELECT length(tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02]');
-- 54.3757408468784
```

- Return the cumulative length traversed by the temporal network point

  `cumulativeLength(tnpoint) → tfloat`

```
SELECT cumulativeLength(tnpoint '{[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02,
  NPoint(1, 0.5)@2001-01-03], [NPoint(1, 0.6)@2001-01-04, NPoint(1, 0.7)@2001-01-05]}');
/* {[0@2001-01-01, 54.3757408468784@2001-01-02, 54.3757408468784@2001-01-03],
   [54.3757408468784@2001-01-04, 81.56361127031770@2001-01-05]} */
```

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

  `speed({tnpointSeq, tnpointSeqSet}) → tfloatSeqSet`

```
SELECT speed(tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.4)@2001-01-02,
  NPoint(1, 0.6)@2001-01-03]') * 3600 * 24;
/* Interp=Step;[21.4016800272077@2001-01-01, 14.2677866848051@2001-01-02,
   14.2677866848051@2001-01-03] */
```

12.7 Transformations

- Transform a temporal network point to another subtype

  `tnpointInst(tnpoint) → tnpointInst`

  `tnpointSeq(tnpoint) → tnpointSeq`

  `tnpointSeqSet(tnpoint) → tnpointSeqSet`

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

- Transform a temporal network point to another interpolation

  `setInterp(tnpoint, interp) → tnpoint`

```
SELECT setInterp(tnpoint 'NPoint(1,0.2)@2001-01-01','linear');
-- [NPoint(1,0.2)@2001-01-01]
SELECT setInterp(tnpoint '{[NPoint(1,0.1)@2001-01-01], [NPoint(1,0.2)@2001-01-02]}',
  'discrete');
-- {NPoint(1,0.1)@2001-01-01, NPoint(1,0.2)@2001-01-02}
```

- 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)@2001-01-01, NPoint(1,0.5)@2001-01-02)}', 6);
-- {[NPoint(1,0.123457)@2001-01-01 00:00:00+01, NPoint(1,0.5)@2001-01-02 00:00:00+01)}
```

12.8 Restrictions

- Restrict to (the complement of) a set of values

  `atValues(tnpoint,values) → tnpoint`

  `minusValues(tnpoint,values) → tnpoint`

```
SELECT atValues(tnpoint '[NPoint(2, 0.3)@2001-01-01, NPoint(2, 0.7)@2001-01-03]',
  'NPoint(2, 0.5)');
-- {[NPoint(2,0.5)@2001-01-02]}
SELECT minusValues(tnpoint '[NPoint(2, 0.3)@2001-01-01, NPoint(2, 0.7)@2001-01-03]',
  'NPoint(2, 0.5)');
/* {[NPoint(2,0.3)@2001-01-01, NPoint(2,0.5)@2001-01-02),
   (NPoint(2,0.5)@2001-01-02, NPoint(2,0.7)@2001-01-03]} */
```

- Restrict to (the complement of) a geometry

  `atGeometry(tnpoint,geometry) → tnpoint`

  `minusGeometry(tnpoint,geometry) → tnpoint`

```
SELECT atGeometry(tnpoint '[NPoint(2, 0.3)@2001-01-01, NPoint(2, 0.7)@2001-01-03]',
  'Polygon((40 40,40 50,50 50,50 40,40 40))');
SELECT minusGeometry(tnpoint '[NPoint(2, 0.3)@2001-01-01, NPoint(2, 0.7)@2001-01-03]',
  'Polygon((40 40,40 50,50 50,50 40,40 40))');
/* {(NPoint(2,0.342593)@2001-01-01 05:06:40.364673+01,
   NPoint(2,0.7)@2001-01-03 00:00:00+01]} */
```

12.9 Distance Operations

- Return the smallest distance ever

  `{geo,npoint,tnpoint} |=| {geo,npoint,tnpoint} → float`

  The operator `|=|` can be used for doing nearest neightbor searches using a GiST or an SP-GiST index (see Section 10.2).

```
SELECT tnpoint '[NPoint(2, 0.3)@2001-01-01, NPoint(2, 0.7)@2001-01-02]' |=|
   geometry 'SRID=5676;Linestring(50 50,55 55)';
-- 31.69220882252415
SELECT tnpoint '[NPoint(2, 0.3)@2001-01-01, NPoint(2, 0.7)@2001-01-02]' |=|
  npoint 'NPoint(1, 0.5)';
-- 19.49691305292373
SELECT tnpoint '[NPoint(2, 0.3)@2001-01-01, NPoint(2, 0.7)@2001-01-02]' |=|
  tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.7)@2001-01-02]';
-- 5.231180723735304
```

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

  `nearestApproachInstant({geo,npoint,tnpoint},{geo,npoint,tnpoint}) → tnpoint`

```
SELECT nearestApproachInstant(tnpoint '[NPoint(2, 0.3)@2001-01-01,
  NPoint(2, 0.7)@2001-01-02]', geometry 'Linestring(50 50,55 55)');
-- NPoint(2,0.349928)@2001-01-01 02:59:44.402905+01
SELECT nearestApproachInstant(tnpoint '[NPoint(2, 0.3)@2001-01-01,
  NPoint(2, 0.7)@2001-01-02]', npoint 'NPoint(1, 0.5)');
-- NPoint(2,0.592181)@2001-01-01 17:31:51.080405+01
```

- Return 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)@2001-01-01,
  NPoint(2, 0.7)@2001-01-02]', geometry 'Linestring(50 50,55 55)'));
-- LINESTRING(50.7960725266492 48.8266286733015,50 50)
SELECT ST_AsText(shortestLine(tnpoint '[NPoint(2, 0.3)@2001-01-01,
  NPoint(2, 0.7)@2001-01-02]', npoint 'NPoint(1, 0.5)'));
-- LINESTRING(77.0902838115125 66.6659083092593,90.8134936900394 46.4385792121146)
```

- Return the temporal distance

  `tnpoint <-> tnpoint → tfloat`

```
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-03]' <->
  npoint 'NPoint(1, 0.2)';
-- [2.34988300875063@2001-01-02 00:00:00+01, 2.34988300875063@2001-01-03 00:00:00+01]
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-03]' <->
  geometry 'Point(50 50)';
-- [25.0496666945044@2001-01-01 00:00:00+01, 26.4085688426232@2001-01-03 00:00:00+01]
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-03]' <->
  tnpoint '[NPoint(1, 0.3)@2001-01-02, NPoint(1, 0.5)@2001-01-04]'
-- [2.34988300875063@2001-01-02 00:00:00+01, 2.34988300875063@2001-01-03 00:00:00+01]
```

12.10 Spatial Operations

- Return the time-weighted centroid

  `twCentroid(tnpoint) → geometry(Point)`

```
SELECT ST_AsText(twCentroid(tnpoint '{[NPoint(1, 0.3)@2001-01-01,
  NPoint(1, 0.5)@2001-01-02, NPoint(1, 0.5)@2001-01-03, NPoint(1, 0.7)@2001-01-04)}'));
-- POINT(79.9787466444847 46.2385558051041)
```

12.11 Bounding Box Operations

- Topological operators

  `{tstzspan,stbox,tnpoint} {&&, <@, @>, ~=, -|-} {tstzspan,stbox,tnpoint} → boolean`

```
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02]' &&
  tstzspan '[2001-01-02,2001-01-03]';
-- true
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02]' @>
  stbox(npoint 'NPoint(1, 0.5)');
-- true
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-03]' ~=
  tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.35)@2001-01-02,
  NPoint(1, 0.5)@2001-01-03]';
-- true
```

- Position operators

  `{stbox,tnpoint} {<<, &<, >>, &>} {stbox,tnpoint} → boolean`

  `{stbox,tnpoint} {<<|, &<|, |>>, |&>} {stbox,tnpoint} → boolean`

  `{tstzspan,stbox,tnpoint} {<<#, &<#, #>>, #&>} {tstzspan,stbox,tnpoint} → boolean`

```
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02]' <<
  stbox(npoint 'NPoint(1, 0.2)');
-- true
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-02]' <<|
  stbox(npoint 'NPoint(1, 0.5)');
-- false
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-03, NPoint(1, 0.5)@2001-01-05]' #&>
  tstzspan '[2001-01-01,2001-01-03]';
-- true
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-03, NPoint(1, 0.3)@2001-01-05]' #>>
  tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.3)@2001-01-02]';
-- true
```

12.12 Spatial Relationships

The topological and distance relationships described in Section 8.5 such as `eIntersects`, `aDwithin`, or `tContains` are defined over the geographical space while the temporal network points are defined over the network space. To be able to apply these relationships to the temporal network points, they must be transformed to temporal geometry points. This can be easily performed using the functions `geometry` and `tgeompoint` or using an explicit casting `::` as shown in the examples below.

- Ever and always spatial relationships

```
SELECT eContains(geometry 'SRID=5676;Polygon((0 0,0 50,50 50,50 0,0 0))',
  tgeompoint(tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)'));
-- false
SELECT eDisjoint(geometry(npoint 'NPoint(2, 0.0)'),
  tgeompoint(tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)'));
-- true
SELECT eIntersects(
  tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)'::tgeompoint,
  tnpoint '[NPoint(2, 0.0)@2001-01-01, NPoint(2, 1)@2001-01-03)'::tgeompoint);
-- false
```

- Spatiotemporal relationships

```
SELECT tContains(geometry 'SRID=5676;Polygon((0 0,0 50,50 50,50 0,0 0))',
  tgeompoint(tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)'));
-- [f@2001-01-01 00:00:00+01, f@2001-01-03 00:00:00+01)
SELECT tDisjoint(geometry(npoint 'NPoint(2, 0.0)'),
  tgeompoint(tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)'));
-- [t@2001-01-01 00:00:00+01, t@2001-01-03 00:00:00+01)
SELECT tDwithin(
  tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-03)'::tgeompoint,
  tnpoint '[NPoint(1, 0.5)@2001-01-01, NPoint(1, 0.3)@2001-01-03)'::tgeompoint, 1);
/* {[t@2001-01-01 00:00:00+01, t@2001-01-01 22:35:55.379053+01],
   (f@2001-01-01 22:35:55.379053+01, t@2001-01-02 01:24:04.620946+01,
   t@2001-01-03 00:00:00+01)} */
```

12.13 Comparisons

- Traditional comparisons

`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
```

- Ever and always comparisons

  `{npoint,tnpoint} {?=, %=} {npoint,tnpoint} → boolean`

```
SELECT tnpoint '[Npoint(1, 0.2)@2001-01-01, Npoint(1, 0.4)@2001-01-04)' ?= Npoint(1, 0.3);
-- true
SELECT tnpoint '[Npoint(1, 0.2)@2001-01-01, Npoint(1, 0.2)@2001-01-04)' &= Npoint(1, 0.2);
-- true
```

- Temporal comparisons

  `{npoint,tnpoint} {#=, #<>} {npoint,tnpoint} → tbool`

```
SELECT tnpoint '[NPoint(1, 0.2)@2001-01-01, NPoint(1, 0.4)@2001-01-03)' #=
  npoint 'NPoint(1, 0.3)';
-- {[f@2001-01-01, t@2001-01-02], (f@2001-01-02, f@2001-01-03)}
SELECT tnpoint '[NPoint(1, 0.2)@2001-01-01, NPoint(1, 0.8)@2001-01-03)' #<>
  tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.7)@2001-01-03)';
-- {[t@2001-01-01, f@2001-01-02], (t@2001-01-02, t@2001-01-03)}
```

12.14 Aggregations

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

- Temporal count

  `tCount(tnpoint) → {tintSeq,tintSeqSet}`

```
WITH Temp(temp) AS (
  SELECT tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)' UNION
  SELECT tnpoint '[NPoint(1, 0.2)@2001-01-02, NPoint(1, 0.4)@2001-01-04)' UNION
  SELECT tnpoint '[NPoint(1, 0.3)@2001-01-03, NPoint(1, 0.5)@2001-01-05)' )
SELECT tCount(Temp)
FROM Temp;
-- {[1@2001-01-01, 2@2001-01-02, 1@2001-01-04, 1@2001-01-05)}
```

- Window count

  `wCount(tnpoint) → {tintSeq,tintSeqSet}`

```
WITH Temp(temp) AS (
  SELECT tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)' UNION
  SELECT tnpoint '[NPoint(1, 0.2)@2001-01-02, NPoint(1, 0.4)@2001-01-04)' UNION
  SELECT tnpoint '[NPoint(1, 0.3)@2001-01-03, NPoint(1, 0.5)@2001-01-05)' )
SELECT wCount(Temp, '1 day')
FROM Temp;
/* {[1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 2@2001-01-04, 1@2001-01-05,
   1@2001-01-06)} */
```

- Temporal centroid

  `tCentroid(tnpoint) → tgeompoint`

```
WITH Temp(temp) AS (
SELECT tnpoint '[NPoint(1, 0.1)@2001-01-01, NPoint(1, 0.3)@2001-01-03)' UNION
SELECT tnpoint '[NPoint(1, 0.2)@2001-01-01, NPoint(1, 0.4)@2001-01-03)' UNION
SELECT tnpoint '[NPoint(1, 0.3)@2001-01-01, NPoint(1, 0.5)@2001-01-03)' )
SELECT astext(tCentroid(Temp))
FROM Temp;
/* {[POINT(72.451531682218 76.5231414472853)@2001-01-01,
   POINT(55.7001249027598 72.9552602410653)@2001-01-03)} */
```

12.15 Indexing

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.

Chapter 13

Temporal Circular Buffers

A static circular buffer type `cbuffer` represents a 2D point and a radius greater than or equal to 0 around the point. The radius represent the "impact" of the point on its surroundings. This impact could be interpreted as noise, pollution, or similar aspects depending on application requirements.

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

Figure 13.1 illustrates the use of the `tcbuffer` type for modelling the circular buffer around the wind swath of tropical storms in 2024. The data is obtained from the National Oceanic and Atmospherique Administration (NOAA). As illustrated in the figure, the types `tgeompoint` and `tcbuffer` allow *linear* interpolation, while as we have seen in Figure 7.2, the type `tgeometry` only allows *step* interpolation.

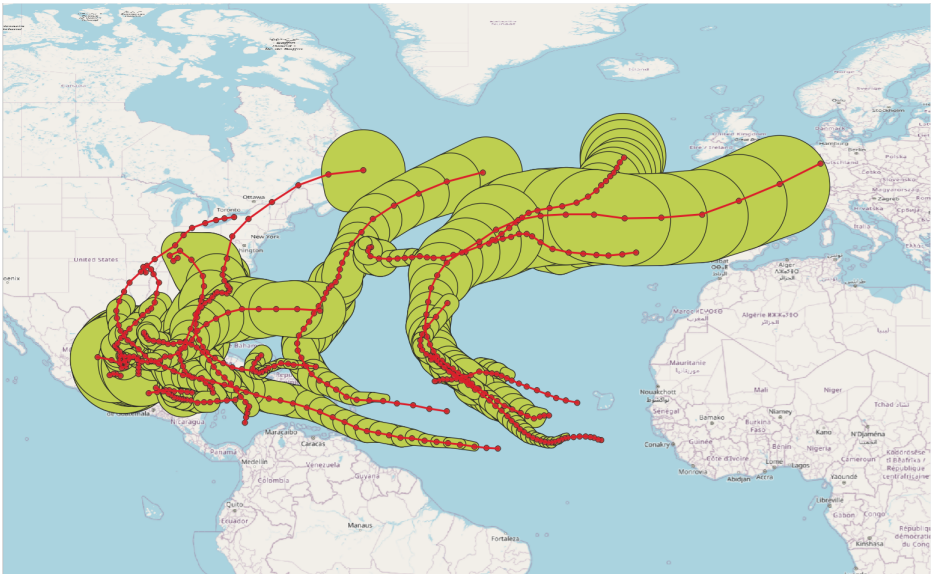


Figure 13.1: Illustration of the use of the `tcbuffer` type for modelling the circular buffer around the wind swath of tropical storms in 2024.

13.1 Static Circular Buffers

A `cbuffer` value is a couple of the form `(point,radius)` where `point` is a geometric point and `radius` is a `float` value greater than or equal to zero representing a radius around the point expressed using the units of the SRID of the point. Examples of input of circular buffer values are as follows:

```
SELECT cbuffer 'Cbuffer(Point(1 1), 0.3)';
SELECT cbuffer 'Cbuffer(Point(1 1), 10.0)';
```

The `cbuffer` type corresponds to the result of the PostGIS function ST_MinimumBoundingRadius.

Values of the circular buffer type must satisfy several constraints so that they are well defined. Examples of incorrect circular buffer type values are as follows.

```
-- incorrect point value
SELECT cbuffer 'Cbuffer(Linestring(1 1,2 2), 1.0)';
-- incorrect 3D point
SELECT cbuffer 'Cbuffer(Point Z(1 1 1), 1.0)';
-- incorrect radius value
SELECT cbuffer 'Cbuffer(Point(1 1), -2.0)';
```

We give next the functions and operators for the circular buffer type.

13.1.1 Constructors

- Constructor for circular buffers

  `cbuffer(geompoint,float)` → `cbuffer`

```
SELECT asText(cbuffer(ST_Point(1,1), 0.3));
-- Cbuffer(POINT(1 1),0.3)
```

13.1.2 Conversions

Values of the `cbuffer` type can be converted to the `geometry` type using an explicit `CAST` or using the `::` notation as shown below. Similarly, a `geometry` can be converted to a `cbuffer` value using the `ST_MinimumBoundingRadius` function in PostGIS.

- Convert between a circular buffer and a geometry

  `geometry::cbuffer`

  `cbuffer::geometry`

```
SELECT ST_AsText(cbuffer(ST_Point(1,1), 1)::geometry);
-- CURVEPOLYGON(CIRCULARSTRING(0 1,2 1,0 1))
SELECT asText(geometry 'SRID=5676;CIRCULARSTRING(0 1,2 1,0 1)'::cbuffer);
-- Cbuffer(POINT(1 1),1)
SELECT asText(geometry 'SRID=5676;LINESTRING(0 0,2 2)'::cbuffer);
-- Cbuffer(POINT(1 1),1.414213562373095)
```

- Convert a circular buffer and, optionally, a timestamp or a period, to a spatiotemporal box

  `stbox(cbuffer)` → `stbox`

  `stbox(cbuffer,{timestamptz,tstzspan})` → `stbox`

```
SELECT stbox(cbuffer 'SRID=5676;Cbuffer(Point(1 1),0.3)');
-- SRID=5676;STBOX X((0.7,0.7),(1.3,1.3))
SELECT stbox(cbuffer 'Cbuffer(Point(1 1),0.3)', timestamptz '2001-01-01');
-- STBOX XT(((0.7,0.7),(1.3,1.3)),[2001-01-01, 2001-01-01])
SELECT stbox(cbuffer 'Cbuffer(Point(1 1),0.3)', tstzspan '[2001-01-01,2001-01-02]');
-- STBOX XT(((0.7,0.7),(1.3,1.3)),[2001-01-01, 2001-01-02])
```

13.1.3 Accessors

- Return the point

  `point(cbuffer) → geompoint`

```
SELECT ST_AsText(point(cbuffer 'Cbuffer(Point(1 1), 0.3)'));
-- Point(1 1)
```

- Return the radius

  `radius(cbuffer) → float`

```
SELECT radius(cbuffer 'Cbuffer(Point(1 1), 0.3)');
-- 0.3
```

13.1.4 Transformations

- Round the point and the radius of the circular buffer to the number of decimal places

  `round(cbuffer,integer=0) → cbuffer`

```
SELECT asText(round(cbuffer(ST_Point(1.123456789,1.123456789), 0.123456789), 6));
-- Cbuffer(POINT(1.123457 1.123457),0.123457)
```

13.1.5 Spatial Operations

- Return or set the spatial reference identifier

  `SRID(cbuffer) → integer`

  `setSRID(cbuffer) → cbuffer`

```
SELECT SRID(cbuffer 'Cbuffer(SRID=5676;Point(1 1), 0.3)');
-- 5676
SELECT asEWKT(setSRID(cbuffer 'Cbuffer(Point(0 0),1)', 4326));
-- SRID=4326;Cbuffer(POINT(0 0),1)
```

- Transform to a spatial reference identifier

  `transform(cbuffer,integer) → cbuffer`

  `transformPipeline(cbuffer,pipeline text,to_srid integer,is_forward bool=true) → cbuffer`

  The `transform` function specifies the transformation with a target SRID. An error is raised when the input circular buffer has an unknown SRID (represented by 0).

  The `transformPipeline` function specifies the transformation with a defined coordinate transformation pipeline represented with the following string format:

  `urn:ogc:def:coordinateOperation:AUTHORITY::CODE`

  The SRID of the input circular buffer is ignored, and the SRID of the output circular buffer will be set to zero unless a value is provided via the optional `to_srid` parameter. As stated by the last parameter, the pipeline is executed by default in a forward direction; by setting the parameter to false, the pipeline is executed in the inverse direction.

```
SELECT asEWKT(transform(cbuffer 'SRID=4326;Cbuffer(Point(4.35 50.85),1)', 3812));
-- SRID=3812;Cbuffer(POINT(648679.0180353033 671067.0556381135),1)
```

```
WITH test(cbuffer, pipeline) AS (
  SELECT cbuffer 'Cbuffer(SRID=4326;Point(4.3525 50.846667),1)',
    text 'urn:ogc:def:coordinateOperation:EPSG::16031' )
SELECT asEWKT(transformPipeline(transformPipeline(cbuffer, pipeline, 4326), pipeline,
  4326, false), 6)
FROM test;
-- SRID=4326;Cbuffer(POINT(4.3525 50.846667),1)
```

13.1.6 Comparisons

The comparison operators (=, <, and so on) are available for circular buffers. They compare first the points and then the radius of the arguments. 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 circular buffers.

- Traditional comparisons

  `cbuffer {=, <>, <, >, <=, >=} cbuffer`

```
SELECT cbuffer 'Cbuffer(Point(3 3), 0.5)' = cbuffer 'Cbuffer(Point(3 3), 0.5)';
-- true
SELECT cbuffer 'Cbuffer(Point(3 3), 0.5)' <> cbuffer 'Cbuffer(Point(3 3), 0.6)';
-- true
SELECT cbuffer 'Cbuffer(Point(3 3), 0.5)' < cbuffer 'Cbuffer(Point(3 3), 0.6)';
-- true
SELECT cbuffer 'Cbuffer(Point(3 3), 0.6)' > cbuffer 'Cbuffer(Point(2 2), 0.6)';
-- true
SELECT cbuffer 'Cbuffer(Point(1 1), 0.5)' <= cbuffer 'Cbuffer(Point(2 2), 0.5)';
-- true
SELECT cbuffer 'Cbuffer(Point(1 1), 0.6)' >= cbuffer 'Cbuffer(Point(1 1), 0.5)';
-- true
```

13.2 Temporal Circular Buffers

The temporal circular buffer type `tcbuffer` allows to represent the movement of objects together with a circular radius around them. As all temporal types, it comes in three subtypes, namely, instant, sequence, and sequence set. Examples of `tcbuffer` values in these subtypes are given next.

```
SELECT tcbuffer 'Cbuffer(Point(1 1), 0.5)@2001-01-01';
SELECT tcbuffer '{Cbuffer(Point(1 1), 0.3)@2001-01-01, Cbuffer(Point(1 1), 0.5)@2001-01-02,
  Cbuffer(Point(1 1), 0.5)@2001-01-03}';
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01, Cbuffer(Point(1 1), 0.4)@2001-01-02,
  Cbuffer(Point(1 1), 0.5)@2001-01-03]';
SELECT tcbuffer '{[Cbuffer(Point(1 1), 1)@2001-01-01, Cbuffer(Point(2 2), 1)@2001-01-02],
  [Cbuffer(Point(2 2), 2)@2001-01-04, Cbuffer(Point(2 2), 3)@2001-01-05]}';
```

The temporal circular buffer type accepts type modifiers (or `typmod` in PostgreSQL terminology) to specify the subtype and/or the spatial reference identifier (SRID). The possible values for the subtype are `Instant`, `Sequence`, and `SequenceSet`. The two arguments are optional and if any of them is not specified for a column, values of any subtype and/or SRID are allowed.

```
SELECT asEWKT(tcbuffer(Sequence,5676) 'SRID=5676;[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-03]');
-- SRID=5676;[Cbuffer(POINT(1 1),0.2)@2001-01-01, Cbuffer(POINT(1 1),0.5)@2001-01-03]
SELECT tcbuffer(Sequence) 'Cbuffer(Point(1 1), 0.2)@2001-01-01';
-- ERROR: Temporal type (Instant) does not match column type (Sequence)
SELECT tcbuffer(5676) 'Cbuffer(Point(1 1), 0.2)@2001-01-01';
-- ERROR:  Temporal circular buffer SRID (0) does not match column SRID (5676)
```

Temporal circular buffer 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 asText(tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(2 2), 0.4)@2001-01-02, Cbuffer(Point(3 3), 0.6)@2001-01-03)');
-- [Cbuffer(Point(1 1),0.2)@2001-01-01, Cbuffer(Point(3 3),0.6)@2001-01-03)
SELECT asText(tcbuffer '{[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(2 2), 0.3)@2001-01-02, Cbuffer(Point(2 2), 0.5)@2001-01-03),
  [Cbuffer(Point(2 2), 0.5)@2001-01-03, Cbuffer(Point(2 2), 0.7)@2001-01-04)}');
/* {[Cbuffer(Point(1 1),0.2)@2001-01-01, Cbuffer(Point(2 2),0.3)@2001-01-02,
   Cbuffer(Point(2 2),0.7)@2001-01-04)} */
```

13.3 Validity of Temporal Circular Buffers

Temporal circular buffer values must satisfy the constraints specified in Section 4.3 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 tcbuffer 'NULL@2001-01-01 08:05:00';
SELECT tcbuffer 'Point(0 0)@NULL';
-- Base type is not a circular buffer
SELECT tcbuffer 'Point(0 0)@2001-01-01 08:05:00';
```

We give next the functions and operators for temporal circular buffer. Most functions and operators for temporal types described in the previous chapters can be applied for temporal circular buffer types. Therefore, in the signatures of the functions, the notation `base` represents a `cbuffer` and the notations `ttype`, `tpoint`, and `tgeompoint` also represent a `tcbuffer`. Furthermore, the functions that have an argument of type `geometry` accept in addition an argument of type `cbuffer`. To avoid redundancy, we only present next some examples of these functions and operators for temporal circular buffers.

13.4 Input and Output

- Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation

  `asText({tcbuffer,tcbuffer[],cbuffer[]}) → {text,text[]}`

  `asEWKT({tcbuffer,tcbuffer[],cbuffer[]}) → {text,text[]}`

```
SELECT asText(tcbuffer 'SRID=4326;[Cbuffer(Point(0 0),1)@2001-01-01,
  Cbuffer(Point(1 1),2)@2001-01-02)');
-- [Cbuffer(Point(0 0),1)@2001-01-01, Cbuffer(Point(1 1),2)@2001-01-02)
SELECT asText(ARRAY[cbuffer 'Cbuffer(Point(0 0),1)', 'Cbuffer(Point(1 1),2)']);
-- {"Cbuffer(POINT(0 0),1)","Cbuffer(POINT(1 1),2)"}
SELECT asEWKT(tcbuffer 'SRID=4326;[Cbuffer(Point(0 0),1)@2001-01-01,
  Cbuffer(Point(1 1),2)@2001-01-02)');
-- SRID=4326;[Cbuffer(Point(0 0),1)@2001-01-01, Cbuffer(Point(1 1),2)@2001-01-02)
SELECT asEWKT(ARRAY[cbuffer 'Cbuffer(SRID=5676;Point(0 0),1)',
  'Cbuffer(SRID=5676;Point(1 1),2)']);
-- {"Cbuffer(SRID=5676;POINT(0 0),1)","Cbuffer(SRID=5676;POINT(1 1),2))"}
```

- Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB) representation, or the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `asBinary(tcbuffer,endian text='') → bytea`

  `asEWKB(tcbuffer,endian text='') → bytea`

  `asHexEWKB(tcbuffer,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 the encoding of the machine is used.

```
SELECT asBinary(tcbuffer 'Cbuffer(Point(1 2),1)@2001-01-01');
-- \x013b0001000000000000f03f0000000000000040000000000000f03f009c57d3c11c0000
SELECT asEWKB(tcbuffer 'SRID=7844;Cbuffer(Point(1 2),1)@2001-01-01');
-- \x013b0041a41e0000000000000000f03f0000000000000040000000000000f03f009c57d3c11c0000
SELECT asHexEWKB(tcbuffer 'SRID=3812;Cbuffer(Point(1 2),1)@2001-01-01');
-- 013B0041E40E0000000000000000F03F0000000000000040000000000000F03F009C57D3C11C0000
```

- Input from the Well-Known Text (WKT) representation or from the Extended Well-Known Text (EWKT) representation

  `tcbufferFromText(text) → tcbuffer`

  `tcbufferFromEWKT(text) → tcbuffer`

```
SELECT asEWKT(tcbufferFromText(text '[Cbuffer(Point(1 2),1)@2001-01-01,
  Cbuffer(Point(3 4),2)@2001-01-02]'));
-- [Cbuffer(POINT(1 2),1)@2001-01-01, Cbuffer(POINT(3 4),2)@2001-01-02]
SELECT asEWKT(tcbufferFromEWKT(text 'SRID=3812;[Cbuffer(Point(1 2),1)@2001-01-01,
  Cbuffer(Point(3 4),2)@2001-01-02]'));
-- SRID=3812;[Cbuffer(Point(1 2),1)@2001-01-01, Cbuffer(Point(3 4),2)@2001-01-02]
```

- Input from the Well-Known Binary (WKB) representation, from the Extended Well-Known Binary (EWKB) representation, or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

  `tcbufferFromBinary(bytea) → tcbuffer`

  `tcbufferFromEWKB(bytea) → tcbuffer`

  `tcbufferFromHexEWKB(text) → tcbuffer`

```
SELECT asEWKT(tcbufferFromBinary(
  '\x013b0001000000000000f03f0000000000000040000000000000f03f009c57d3c11c0000'));
-- Cbuffer(POINT(1 2),1)@2001-01-01
SELECT asEWKT(tcbufferFromEWKB(
  '\x013b0041a41e0000000000000000f03f0000000000000040000000000000f03f009c57d3c11c0000'));
-- SRID=7844;Cbuffer(Point(1 1),2)@2001-01-01
SELECT asEWKT(tcbufferFromHexEWKB(
  '013B0041E40E0000000000000000F03F0000000000000040000000000000F03F009C57D3C11C0000'));
-- SRID=3812;Cbuffer(POINT(1 2),1)@2001-01-01
```

13.5 Constructors

- Constructor for temporal circular buffers having a constant value

  `tcbuffer(cbuffer,timestamptz) → tcbufferInst`

  `tcbuffer(cbuffer,tstzset) → tcbufferDiscSeq`

  `tcbuffer(cbuffer,tstzspan,interp='linear') → tcbufferContSeq`

  `tcbuffer(cbuffer,tstzspanset,interp='linear') → tcbufferSeqSet`

```
SELECT asText(tcbuffer('Cbuffer(Point(1 1), 0.5)', timestamptz '2001-01-01'));
-- Cbuffer(Point(1 1),0.5)@2001-01-01
SELECT asText(tcbuffer('Cbuffer(Point(1 1), 0.3)',
  tstzset '{2001-01-01, 2001-01-03, 2001-01-05}'));
/* {Cbuffer(Point(1 1),0.3)@2001-01-01, Cbuffer(Point(1 1),0.3)@2001-01-03,
   Cbuffer(Point(1 1),0.3)@2001-01-05} */
SELECT asText(tcbuffer('Cbuffer(Point(1 1), 0.5)',
  tstzspan '[2001-01-01, 2001-01-02]'));
-- [Cbuffer(Point(1 1),0.5)@2001-01-01, Cbuffer(Point(1 1),0.5)@2001-01-02]
SELECT asText(tcbuffer('Cbuffer(Point(1 1), 0.2)',
  tstzspanset '{[2001-01-01, 2001-01-03]}', 'step'));
-- Interp=Step;{[Cbuffer(Point(1 1),0.2)@2001-01-01, Cbuffer(Point(1 1),0.2)@2001-01-03]}
```

- Constructor for temporal circular buffers of sequence subtype

  `tcbufferSeq(tcbufferInst[],interp='linear',leftInc bool=true, rightInc bool=true) →tcbufferSeq`

```
SELECT asText(tcbufferSeq(ARRAY[tcbuffer 'Cbuffer(Point(1 1), 0.3)@2001-01-01',
  'Cbuffer(Point(2 2), 0.5)@2001-01-02', 'Cbuffer(Point(1 1), 0.5)@2001-01-03']));
/* [Cbuffer(Point(1 1),0.3)@2001-01-01, Cbuffer(Point(2 2),0.5)@2001-01-02,
   Cbuffer(Point(1 1),0.5)@2001-01-03] */
SELECT asText(tcbufferSeq(ARRAY[tcbuffer 'Cbuffer(Point(1 1), 0.3)@2001-01-01',
  'Cbuffer(Point(2 2), 0.5)@2001-01-02', 'Cbuffer(Point(1 1), 0.5)@2001-01-03'],
  'discrete'));
/* {Cbuffer(POINT(1 1),0.3)@2001-01-01, Cbuffer(POINT(2 2),0.5)@2001-01-02,
   Cbuffer(POINT(1 1),0.5)@2001-01-03} */
SELECT asText(tcbufferSeq(ARRAY[tcbuffer 'Cbuffer(Point(1 1), 0.2)@2001-01-01',
  'Cbuffer(Point(1 1), 0.4)@2001-01-02', 'Cbuffer(Point(1 1), 0.5)@2001-01-03'], 'step'));
/* Interp=Step;[Cbuffer(Point(1 1),0.2)@2001-01-01, Cbuffer(Point(1 1),0.4)@2001-01-02,
   Cbuffer(Point(1 1),0.5)@2001-01-03] */
```

- Constructor for temporal circular buffers of sequence set subtype

  `tcbufferSeqSet(tcbuffer[]) → tcbufferSeqSet`

  `tcbufferSeqSetGaps(tcbufferInst[],maxt=NULL,maxdist=NULL,interp='linear') → tcbufferSeqSet`

```
SELECT asText(tcbufferSeqSet(ARRAY[tcbuffer
  '[Cbuffer(Point(1 1),0.2)@2001-01-01, Cbuffer(Point(2 2),0.4)@2001-01-02]',
  '[Cbuffer(Point(2 2),0.6)@2001-01-03, Cbuffer(Point(2 2),0.8)@2001-01-04]']));
/* {[Cbuffer(Point(1 1),0.2)@2001-01-01, Cbuffer(Point(2 2),0.4)@2001-01-02],
   [Cbuffer(Point(2 2),0.6)@2001-01-03, Cbuffer(Point(2 2),0.6)@2001-01-04]} */
SELECT asText(tcbufferSeqSetGaps(ARRAY[tcbuffer 'Cbuffer(Point(1 1),0.1)@2001-01-01',
  'Cbuffer(Point(1 1),0.3)@2001-01-03', 'Cbuffer(Point(1 1),0.5)@2001-01-05'], '1 day'));
/* {[Cbuffer(Point(1 1),0.1)@2001-01-01], [Cbuffer(Point(1 1),0.3)@2001-01-03],
   [Cbuffer(Point(1 1),0.5)@2001-01-05]} */
```

- Construct a temporal circular buffer from a temporal geometry point and a temporal float

  `tcbuffer(tgeompoint, tfloat) → tcbuffer`

  The time frame of the result is the intersection of the time frames of the temporal point and the temporal float. If they do not intersect, a null value is returned

```
SELECT asText(tcbuffer(tgeompoint '[POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02)',
  tfloat '[1@2001-01-01, 2@2001-01-02)'));
-- [Cbuffer(Point(1 1),1)@2001-01-01, Cbuffer(Point(1 1),2)@2001-01-02)
SELECT asText(tcbuffer(tgeompoint '[POINT(1 1)@2001-01-01, POINT(2 2)@2001-01-02)',
  tfloat '[2@2001-01-03, 3@2001-01-04)'));
-- NULL
```

13.6 Conversions

A temporal circular buffer 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 `cbuffer` value.

- Convert a temporal circular buffer to a temporal geometry point

  `tcbuffer::tgeompoint`

```
SELECT asText((tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.3)@2001-01-02)')::tgeompoint);
-- [POINT(1 1)@2001-01-01, POINT(1 1)@2001-01-02)
```

- Convert a temporal circular buffer to a temporal float

  `tcbuffer::tfloat`

```
SELECT (tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.3)@2001-01-02)')::tfloat;
-- [0.2@2001-01-01, 0.3@2001-01-02)
```

13.7 Accessors

- Return the values

  `getValues(tcbuffer) → cbufferset`

```
SELECT asText(getValues(tcbuffer '{[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02)}'));
-- {"Cbuffer(Point(1 1),0.3)","Cbuffer(Point(1 1),0.5)"}
SELECT asEWKT(getValues(tcbuffer 'SRID=5676;{[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.3)@2001-01-02)}'));
-- SRID=5676;{"Cbuffer(Point(1 1),0.3)"}
```

- Return the points or the radii

  `points(tcbuffer) → geomset`

  `radius(tcbuffer) → floatset`

```
SELECT asEWKT(points(tcbuffer 'SRID=5676;{Cbuffer(Point(3 3), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02}'));
-- SRID=5676;{Point(1 1), Point(3 3)}
SELECT radius(tcbuffer 'SRID=5676;{Cbuffer(Point(3 3), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02}');
-- {0.3, 0.5}
```

- Return the value at a timestamp

  `valueAtTimestamp(tcbuffer,timestamptz) → cbuffer`

```
SELECT asText(valueAtTimestamp(tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(3 3), 0.5)@2001-01-03)', '2001-01-02'));
-- Cbuffer(Point(2 2),0.4)
```

13.8 Transformations

- Transform a temporal circular buffer to another subtype

  `tcbufferInst(tcbuffer) → tcbufferInst`

  `tcbufferSeq(tcbuffer) → tcbufferSeq`

  `tcbufferSeqSet(tcbuffer) → tcbufferSeqSet`

```
SELECT asText(tcbufferSeq(tcbuffer 'Cbuffer(Point(1 1), 0.5)@2001-01-01', 'discrete'));
-- {Cbuffer(Point(1 1),0.5)@2001-01-01}
SELECT asText(tcbufferSeq(tcbuffer 'Cbuffer(Point(1 1), 0.5)@2001-01-01'));
-- [Cbuffer(Point(1 1),0.5)@2001-01-01]
SELECT asText(tcbufferSeqSet(tcbuffer 'Cbuffer(Point(1 1), 0.5)@2001-01-01'));
-- {[Cbuffer(Point(1 1),0.5)@2001-01-01]}
```

- Transform a temporal circular buffer to another interpolation

  `setInterp(tcbuffer, interp) → tcbuffer`

```
SELECT asText(setInterp(tcbuffer 'Cbuffer(Point(1 1),0.2)@2001-01-01','linear'));
-- [Cbuffer(Point(1 1),0.2)@2001-01-01]
SELECT asText(setInterp(tcbuffer '{[Cbuffer(Point(1 1),0.1)@2001-01-01],
  [Cbuffer(Point(1 1),0.2)@2001-01-02]}', 'discrete'));
-- {Cbuffer(Point(1 1),0.1)@2001-01-01, Cbuffer(Point(1 1),0.2)@2001-01-02}
```

- Round the points and the radii of the temporal circular buffer to the number of decimal places

  `round(tcbuffer,integer) → tcbuffer`

```
SELECT asText(round(tcbuffer '{[Cbuffer(Point(1 1.123456789),0.123456789)@2001-01-01,
  Cbuffer(Point(1 1),0.5)@2001-01-02)}', 3));
-- {[Cbuffer(Point(1 1.123),0.123)@2001-01-01, Cbuffer(Point(1 1),0.5)@2001-01-02)}
```

13.9 Spatial Operations

- Return or set the spatial reference identifier

  `SRID(tcbuffer) → integer`

  `setSRID(tcbuffer) → tcbuffer`

```
SELECT SRID(tcbuffer 'SRID=5676;[Cbuffer(Point(0 0),1)@2001-01-01,
  Cbuffer(Point(1 1),2)@2001-01-02)');
-- 5676
SELECT asEWKT(setSRID(tcbuffer '[Cbuffer(Point(0 0),1)@2001-01-01,
  Cbuffer(Point(1 1),2)@2001-01-02)', 5676));
-- SRID=5676;[Cbuffer(POINT(0 0),1)@2001-01-01, Cbuffer(POINT(1 1),2)@2001-01-02)
```

- Transform to a spatial reference identifier

  `transform(tcbuffer,integer) → tcbuffer`

  `transformPipeline(tcbuffer,pipeline text,to_srid integer,is_forward bool=true) → tcbuffer`

  The `transform` function specifies the transformation with a target SRID. An error is raised when the input temporal circular buffer has an unknown SRID (represented by 0).

  The `transformPipeline` function specifies the transformation with a defined coordinate transformation pipeline represented with the following string format: `urn:ogc:def:coordinateOperation:AUTHORITY::CODE`. The SRID of the input temporal circular buffer is ignored, and the SRID of the output temporal circular buffer will be set to zero unless a value is provided via the optional `to_srid` parameter. As stated by the last parameter, the pipeline is executed by default in a forward direction; by setting the parameter to false, the pipeline is executed in the inverse direction.

```
SELECT asEWKT(transform(tcbuffer 'SRID=4326;Cbuffer(Point(4.35 50.85),1)@2001-01-01',
  3812));
-- SRID=3812;Cbuffer(POINT(648679.0180353033 671067.0556381135),1)@2001-01-01
```

```
WITH test(tcbuffer, pipeline) AS (
  SELECT tcbuffer 'SRID=4326;{Cbuffer(Point(4.3525 50.846667),1)@2001-01-01,
    Cbuffer(Point(-0.1275 51.507222),2)@2001-01-02}',
    text 'urn:ogc:def:coordinateOperation:EPSG::16031' )
SELECT asEWKT(transformPipeline(transformPipeline(tcbuffer, pipeline, 4326),  pipeline,
  4326, false), 6)
FROM test;
/* SRID=4326;{Cbuffer(POINT(4.3525 50.846667),1)@2001-01-01,
   Cbuffer(POINT(-0.1275 51.507222),2)@2001-01-02} */
```

- Return the area traversed by the temporal circular buffer

  `traversedArea(tcbuffer) → geometry`

```
SELECT ST_AsText(traversedArea(tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02]'));
-- CURVEPOLYGON(CIRCULARSTRING(0.7 1,1.3 1,0.7 1))
```

13.10 Distance Operations

- Return the smallest distance ever

  `{geo,cbuffer,tcbuffer} |=| {geo,cbuffer,tcbuffer} → float`

  The operator `|=|` that can be used for doing nearest neightbor searches using a GiST or an SP-GiST index (see Section 10.2).

```
SELECT tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-02]' |=| geometry 'Linestring(50 50,55 55)';
-- 67.58225099390856
SELECT tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-02]' |=| cbuffer 'Cbuffer(Point(1 1), 0.5)';
-- 0.6142135623730951
```

- TODO Return the instant of the first temporal circular buffer at which the two arguments are at the nearest distance

  `nearestApproachInstant({geo,cbuffer,tcbuffer},{geo,cbuffer,tcbuffer}) → tcbuffer`

```
SELECT nearestApproachInstant(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-02]', geometry 'Linestring(50 50,55 55)');
-- Cbuffer(Point(2 2),0.349928)@2001-01-01 02:59:44.402905+01
SELECT nearestApproachInstant(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-02]', cbuffer 'Cbuffer(Point(1 1), 0.5)');
-- Cbuffer(Point(2 2),0.592181)@2001-01-01 17:31:51.080405+01
```

- TODO Return the line connecting the nearest approach point between the two arguments

  `shortestLine({geo,cbuffer,tcbuffer},{geo,cbuffer,tcbuffer}) → geometry`

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

```
SELECT ST_AsText(shortestLine(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-02]', geometry 'Linestring(50 50,55 55)'));
-- LINESTRING(50 50,2.212132034355964 2.212132034355964)
SELECT ST_AsText(shortestLine(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-02]', cbuffer 'Cbuffer(Point(1 1), 0.5)'));
-- LINESTRING(1.353553390593274 1.353553390593274,1.787867965644036 1.787867965644036)
```

- TODO Return the temporal distance

  `{geo,cbuffer,tcbuffer} <-> {geo,cbuffer,tcbuffer} → tfloat`

```
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-03]' <-> cbuffer 'Cbuffer(Point(1 1), 0.2)';
-- [2.34988300875063@2001-01-02 00:00:00+01, 2.34988300875063@2001-01-03 00:00:00+01]
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-03]' <-> geometry 'Point(50 50)';
-- [25.049666694504@2001-01-01 00:00:00+01, 26.4085688426232@2001-01-03 00:00:00+01]
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-03]' <->
  tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-02, Cbuffer(Point(1 1), 0.5)@2001-01-04]'
-- [2.34988300875063@2001-01-02 00:00:00+01, 2.34988300875063@2001-01-03 00:00:00+01]
```

13.11 Restrictions

- TODO Restrict to (the complement of) a set of values

  `atValues(tcbuffer,values) → tcbuffer`

  `minusValues(tcbuffer,values) → tcbuffer`

```
SELECT asText(atValues(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-03]', cbuffer 'Cbuffer(Point(2 2), 0.5)'));
-- {[Cbuffer(Point(2 2),0.5)@2001-01-02]}
SELECT asText(minusValues(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-03]', cbuffer 'Cbuffer(Point(2 2), 0.5)'));
/* {[Cbuffer(Point(2 2),0.3)@2001-01-01, Cbuffer(Point(2 2),0.5)@2001-01-02),
   (Cbuffer(Point(2 2),0.5)@2001-01-02, Cbuffer(Point(2 2),0.7)@2001-01-03]} */
```

- TODO Restrict to (the complement of) a geometry

  `atGeometry(tcbuffer,geometry) → tcbuffer`

  `minusGeometry(tcbuffer,geometry) → tcbuffer`

```
SELECT asText(atGeometry(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-03]', 'Polygon((0 0,0 2,2 2,2 0,0 0))'));
-- {[Cbuffer(POINT(2 2),0.3)@2001-01-01, Cbuffer(POINT(2 2),0.7)@2001-01-03]}
SELECT asText(minusGeometry(tcbuffer '[Cbuffer(Point(2 2), 0.3)@2001-01-01,
  Cbuffer(Point(2 2), 0.7)@2001-01-03]', 'Polygon((0 0,0 2,2 2,2 0,0 0))'));
/* {(Cbuffer(Point(2 2),0.342593)@2001-01-01 05:06:40.364673+01,
   Cbuffer(Point(2 2),0.7)@2001-01-03 00:00:00+01]} */
```

13.12 Comparisons

- Traditional comparisons

  `tcbuffer {=, <>, <, >, <=, >=} tcbuffer → boolean`

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

- Ever and always comparisons

  `{tcbuffer,cbuffer} {?=, %=} {tcbuffer,cbuffer} → boolean`

```
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.4)@2001-01-03)' ?= cbuffer 'Cbuffer(Point(1 1), 0.3)';
-- true
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.4)@2001-01-03)' ?=
  tcbuffer '[Cbuffer(Point(1 1), 0.4)@2001-01-01, Cbuffer(Point(1 1), 0.2)@2001-01-03)';
-- true
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
```

```
  Cbuffer(Point(1 1), 0.2)@2001-01-04)' %= cbuffer 'Cbuffer(Point(1 1), 0.2)';
-- true
```

- Temporal comparisons

`tcbuffer {#=, #<>} tcbuffer → tbool`

```
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.4)@2001-01-03)' #= cbuffer 'Cbuffer(Point(1 1), 0.3)';
-- {[f@2001-01-01, t@2001-01-02], (f@2001-01-02, f@2001-01-03)}
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.8)@2001-01-03)' #<>
  tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01, Cbuffer(Point(1 1), 0.7)@2001-01-03)';
-- {[t@2001-01-01, f@2001-01-02], (t@2001-01-02, t@2001-01-03)}
```

13.13 Bounding Box Operations

- Topological operators

`{tstzspan,stbox,tcbuffer} {&&, <@, @>, ~=, -|-} {tstzspan,stbox,tcbuffer} → boolean`

```
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02]' && cbuffer 'Cbuffer(Point(1 1), 0.5)';
-- true
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02]' @> stbox(cbuffer 'Cbuffer(Point(1 1), 0.5)');
-- true
SELECT cbuffer 'Cbuffer(Point(1 1), 0.5)'::geometry <@
  tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01, Cbuffer(Point(1 1), 0.5)@2001-01-02]';
-- true
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-03]' ~= tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.35)@2001-01-02, Cbuffer(Point(1 1), 0.5)@2001-01-03]';
-- true
```

- Position operators

`{stbox,tcbuffer} {<<, &<, >>, &>} {stbox,tcbuffer} → boolean`

`{stbox,tcbuffer} {<<|, &<|, |>>, |&>} {stbox,tcbuffer} → boolean`

`{tstzspan,stbox,tcbuffer} {<<#, &<#, #>>, #&>} {tstzspan,stbox,tcbuffer} → boolean`

```
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02]' << cbuffer 'Cbuffer(Point(1 1), 0.2)'
-- false
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02]' <<| stbox(cbuffer 'Cbuffer(Point(1 1), 0.5)')
-- false
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02]' &> cbuffer 'Cbuffer(Point(1 1), 0.3)'::geometry
-- true
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-02]' >>#
  tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-03, Cbuffer(Point(1 1), 0.5)@2001-01-05]'
-- true
```

13.14 Spatial Relationships

- TODO Ever and always relationships

  `eContains(geometry,tcbuffer) → boolean`

  `aContains(geometry,tcbuffer) → boolean`

  `eDisjoint({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

  `aDisjoint({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

  `eDwithin({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer},float) → boolean`

  `aDwithin({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer},float) → boolean`

  `eIntersects({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

  `aIntersects({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

  `eTouches({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

  `aTouches({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

```
SELECT eContains(geometry 'Polygon((0 0,0 50,50 50,50 0,0 0))',
  tcbuffer '[Cbuffer(Point(1 1), 0.1)@2001-01-01, Cbuffer(Point(1 1), 0.3)@2001-01-03)');
-- false
SELECT eDisjoint(cbuffer 'Cbuffer(Point(2 2), 0.0)',
  tcbuffer '[Cbuffer(Point(1 1), 0.1)@2001-01-01, Cbuffer(Point(1 1), 0.3)@2001-01-03)');
-- true
SELECT eIntersects(tcbuffer '[Cbuffer(Point(1 1), 0.1)@2001-01-01,
  Cbuffer(Point(1 1), 0.3)@2001-01-03)',
  tcbuffer '[Cbuffer(Point(2 2), 0.0)@2001-01-01, Cbuffer(Point(2 2), 1)@2001-01-03)');
-- false
```

- TODO Spatiotemporal relationships

  `tContains(geometry,tcbuffer) → boolean`

  `tDisjoint({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

  `tDwithin({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer},float) → boolean`

  `tIntersects({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

  `tTouches({geometry,cbuffer,tcbuffer},{geometry,cbuffer,tcbuffer}) → boolean`

```
SELECT tDisjoint(geometry 'Polygon((0 0,0 50,50 50,50 0,0 0))',
  tcbuffer '[Cbuffer(Point(1 1), 0.1)@2001-01-01, Cbuffer(Point(1 1), 0.3)@2001-01-03)');
-- {[t@2001-01-01 00:00:00+01, t@2001-01-03 00:00:00+01)}
SELECT tDwithin(tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-03)', tcbuffer '[Cbuffer(Point(1 1), 0.5)@2001-01-01,
  Cbuffer(Point(1 1), 0.3)@2001-01-03)', 1);
/* {[t@2001-01-01 00:00:00+01, t@2001-01-01 22:35:55.379053+01],
   (f@2001-01-01 22:35:55.379053+01, t@2001-01-02 01:24:04.620946+01,
   t@2001-01-03 00:00:00+01)} */
```

13.15 Aggregations

The three aggregate functions for temporal circular buffers are illustrated next.

- Temporal count

  `tCount(tcbuffer) → {tintSeq,tintSeqSet}`

```
WITH Temp(temp) AS (
  SELECT tcbuffer '[Cbuffer(Point(1 1), 0.1)@2001-01-01,
    Cbuffer(Point(1 1), 0.3)@2001-01-03)' UNION
  SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-02,
    Cbuffer(Point(1 1), 0.4)@2001-01-04)' UNION
  SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-03,
    Cbuffer(Point(1 1), 0.5)@2001-01-05)' )
SELECT tCount(Temp)
FROM Temp;
-- {[1@2001-01-01, 2@2001-01-02, 1@2001-01-04, 1@2001-01-05)}
```

- Window count

  `wCount(tcbuffer) → {tintSeq,tintSeqSet}`

```
WITH Temp(temp) AS (
  SELECT tcbuffer '[Cbuffer(Point(1 1), 0.1)@2001-01-01,
    Cbuffer(Point(1 1), 0.3)@2001-01-03)' UNION
  SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-02,
    Cbuffer(Point(1 1), 0.4)@2001-01-04)' UNION
  SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-03,
    Cbuffer(Point(1 1), 0.5)@2001-01-05)' )
SELECT wCount(Temp, '1 day')
FROM Temp;
/* {[1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 2@2001-01-04, 1@2001-01-05,
   1@2001-01-06)} */
```

- TODO Temporal centroid

  `tCentroid(tcbuffer) → tgeompoint`

```
WITH Temp(temp) AS (
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.1)@2001-01-01,
  Cbuffer(Point(1 1), 0.3)@2001-01-03)' UNION
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.2)@2001-01-01,
  Cbuffer(Point(1 1), 0.4)@2001-01-03)' UNION
SELECT tcbuffer '[Cbuffer(Point(1 1), 0.3)@2001-01-01,
  Cbuffer(Point(1 1), 0.5)@2001-01-03)' )
SELECT astext(tCentroid(Temp))
FROM Temp;
/* {[POINT(72.451531682218 76.5231414472853)@2001-01-01,
   POINT(55.7001249027598 72.9552602410653)@2001-01-03)} */
```

13.16 Indexing

GiST and SP-GiST indexes can be created for table columns of temporal circular buffers. 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 circular buffers, which is an `stbox`, as all spatiotemporal types.

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

- `<<`, `&<`, `&>`, `>>`, `<<|`, `&<|`, `|&>`, `|>>`, which only consider the spatial dimension in temporal circular buffers,
- `<<#`, `&<#`, `#&>`, `#>>`, which only consider the time dimension in temporal circular buffers,

- `&&`, `@>`, `<@`, `~=`, `-|-`, 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 MobilityDB Types

MobilityDB defines four *template types* that act as type constructors over *base types*. These template types are `set`, `span`, `spanset`, and `temporal`. Table A.1 shows the types defined over these template types. Furthermore, MobilityDB defines two bounding box types, namely, `tbox` and `stbox`.

Base Types	Template Types			
	set	**span**	**spanset**	**temporal**
bool				tbool
text	textset			ttext
integer	intset	intspan	intspanset	tint
bigint	bigintset	bigintspan	bigintspanset	
float	floatset	floatspan	floatspanset	tfloat
date	dateset	datespan	datespanset	
timestamptz	tstzset	tstzspan	tstzspanset	
geometry	geomset			tgeompoint, tgeometry
geography	geogset			tgeogpoint, tgeography
pose	poseset			tpose
npoint	npointset			tnpoint
cbuffer	cbufferset			tcbuffer

Table A.1: MobilityDB current instantiations of template types

A.2 Set and Span Types

A.2.1 Input and Output

- asText: Return the Well-Known Text (WKT) representation
- asBinary: Return the Well-Known Binary (WKB) or the Hexadecimal Well-Known Binary (HexWKB) representation
- settypeFromBinary, spantypeFromBinary, spansettypeFromBinary: Input from the Well-Known Binary (WKB) representation
- settypeFromHexWKB, spantypeFromHexWKB, spansettypeFromHexWKB: Input from the Hexadecimal Well-Known Binary (HexWKB) representation

A.2.2 Constructors

- set: Constructor for set values
- span: Constructor for span values
- spanset: Constructor for span set values

A.2.3 Conversions

- ::, set(base), span(base), spanset(base): Convert a base value to a set, span, or span set value
- ::, spanset(set): Convert a set value to a span set value
- ::, spanset(span): Convert a span value to a span set value
- ::, range(span), span(range): Convert a span value to and from a PostgreSQL range value
- ::, multirange(spanset), spanset(multirange): Convert a span set value to and from a PostgreSQL multirange value

A.2.4 Accessors

- memSize: Return the memory size in bytes
- lower, upper: Return the lower or upper bound
- lowerInc, upperInc: Is the lower or upper bound inclusive?
- width: Return the width of the span as a float
- duration: Return the duration
- span: Return the bounding span of the set or span set ignoring the potential time gaps
- numValues, getValues: Return the (number of) values
- startValue, endValue, valueN: Return the start, end, or n-th value
- numSpans, spans: Return (the number of) composing spans
- startSpan, endSpan, spanN: Return the start, end, or n-th span
- numDates, dates: Return the (number of) different dates
- startDate, endDate, dateN: Return the start, end, or n-th date
- numTimestamps, timestamps: Return the (number of) different timestamps
- startTimestamp, endTimestamp, timestampN: Return the start, end, or n-th timestamp

A.2.5 Transformations

- expand: Expand or shrink the bounds by a value or an interval
- shift, scale, shiftScale: Shift and/or scale by a value or an interval
- floor, ceil: Round down or up to the lowest integer
- round: Round to a number of decimal places
- degrees, radians: Transform to degrees or radians
- lower, upper, initcap: Transform to lowercase, uppercase, or initcap
- ||: Text concatenation
- tprecision: Set the temporal precision to the interval with respect to the origin

A.2.6 Spatial Reference System

- SRID, setSRID: Return or set the spatial reference identifier
- transform, transformPipeline: Transform to a spatial reference identifier

A.2.7 Set Operations

- +, -, *: Union, difference, and intersection of sets or spans

A.2.8 Bounding Box Operations

A.2.8.1 Topological Operations

- &&: Do the sets or spans overlap (have values in common)?
- @>: Does the first set or span contain the second one?
- <@: Is the first set or span contained by the second one?
- -|-: Is the first set or span adjacent to the second one?

A.2.8.2 Position Operations

- <<, <<#: Is the first set or span scritly to the left of the second one?
- >>, #>>: Is the first set or span strictly to the right of the second one?
- &<, &<#: Is the first set or span not to the right of the second one?
- &>, #&>: Is the first set or span not to the left of the second one?

A.2.8.3 Splitting Operations

- splitNSpans: Return an array of N spans obtained by merging the elements of a set or the spans of a spanset
- splitEachNSpans: Return an array of spans obtained by merging N consecutive elements of a set or N consecutive spans of a spanset

A.2.9 Distance Operations

- <->: Return the smallest distance ever

A.2.10 Comparisons

- =, <>, <, >, <=, >=, set_cmp, span_cmp, and spanset_cmp: Traditional comparisons

A.2.11 Aggregations

- tCount: Temporal count
- extent: Bounding span
- setUnion, spanUnion: Aggregate union

A.3 Box Types

A.3.1 Input and Output

- asText: Return the Well-Known Text (WKT) representation
- asBinary, asHexWKB: Return the Well-Known Binary (WKB) or the Hexadecimal Well-Known Binary (HexWKB) representation as text
- boxFromBinary, boxFromHexWKB: Input from the Well-Known Binary (WKB) or from the Hexadecimal Well-Known Binary (HexWKB) representation

A.3.2 Constructors

- tbox: Constructor for `tbox`
- stboxX, stboxZ, stboxT, stboxXT, stboxZT, geodstboxZ, geodstboxT, geodstboxZT: Constructor for `stbox`

A.3.3 Conversions

- tbox::type: Convert a `tbox` to another type
- type::tbox: Convert another type to a `tbox`
- stbox::type: Convert an `stbox` to another type
- type::stbox: Convert another type to an `stbox`

A.3.4 Accessors

- hasX, hasZ, hasT: Has X/Z/T dimension?
- isGeodetic: Is geodetic?
- xMin, yMin, zMin, tMin: Return the minimum X/Y/Z/T value
- xMax, yMax, zMax, tMax: Return the maximum X/Y/Z/T value
- xMinInc, tMinInc: Is the minimum X/T value inclusive?
- xMaxInc, tMaxInc: Is the maximum X/T value inclusive?
- area, volume, perimeter: Area, volume, perimeter

A.3.5 Transformations

- shiftValue, scaleValue, shiftScaleValue: Shift and/or scale the span of a bounding box by a value
- shiftTime, scaleTime, shiftScaleTime: Shift and/or scale the period of a bounding box by an interval
- getSpace: Return the spatial dimension of a bounding box, removing the temporal dimension if any
- expandValue, expandSpace, expandTime: Expand the numeric, spatial, or temporal dimension of a bounding box by a value or an interval
- round: Round the values or the coordinates of a bounding box to a number of decimal places

A.3.6 Spatial Reference System

- SRID, setSRID: Return or set the spatial reference identifier
- transform, transformPipeline: Transform to a spatial reference identifier

A.3.7 Splitting Operations

- quadSplit: Split a bounding box in quadrants or octants

A.3.8 Set Operations

- +, *: Union and intersection of bounding boxes

A.3.9 Bounding Box Operations

A.3.9.1 Topological Operations

- &&: 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.3.9.2 Position Operations

- <<, <<|, <</, <<#: Are the X/Y/Z/T values of the first bounding box strictly less than those of the second one?
- >>,|>>, />>, #>>: Are the X/Y/Z/T values of the first bounding box strictly greater than those of the second one?
- &<, &<|, &</, &<#: Are the X/Y/Z/T values of the first bounding box not greater than those of the second one?
- &>, |&>, /&>, #&>: Are the X/Y/Z/T values of the first bounding box not less than those of the second one?

A.3.10 Comparisons

- =, <>, <, >, <=, >=tbox_cmp,: Traditional comparisons

A.3.11 Aggregations

- extent: Bounding box extent

A.4 Temporal Types

A.4.1 Input and Output

- asText: Return the Well-Known Text (WKT) representation
- asMFJSON: Return the Moving Features JSON (MF-JSON) representation
- asBinary, asHexWKB: Return the Well-Known Binary (WKB) or the Hexadecimal Well-Known Binary (HexWKB) representation
- ttypeFromMFJSON: Input from a Moving Features JSON (MF-JSON) representation
- ttypeFromBinary, ttypeFromHexWKB: Input from the Well-Known Binary (WKB) or from the Hexadecimal Well-Known Binary (HexWKB) representation

A.4.2 Constructors

- ttype: Constructor for temporal types from a base value and a time value
- ttypeSeq: Constructor for temporal types of sequence subtype
- ttypeSeqSet, ttypeSeqSetGaps: Constructor for temporal types of sequence set subtype

A.4.3 Conversions

- ttype::tstzspan: Convert a temporal value to a timestamptz span

A.4.4 Accessors

- memSize: Return the memory size in bytes
- tempSubtype: Return the temporal subtype
- interp: Return the interpolation
- getValue, getTimestamp: Return the value or the timestamp of an instant
- getValues, getTime: Return the values or the time on which the temporal value is defined
- startValue, endValue, valueN: Return the start, end, or n-th value
- valueAtTimestamp: Return the value at a timestamp
- duration: Return the duration
- lowerInc, upperInc: Is the start/end instant inclusive?
- numInstants, instants: Return the (number of) different instants
- startInstant, endInstant, instantN: Return the start, end, or n-th instant
- numTimestamps, timestamps: Return the (number of different) timestamps
- startTimestamp, endTimestamp, timestampN: Return the start, end, or n-th timestamp
- numSequences, sequences: Return the (number of) sequences
- startSequence, endSequence, sequenceN: Return the start, end, or n-th sequence
- segments: Return the segments

A.4.5 Transformations

- ttypeInst, ttypeSeq, ttypeSeqSet: Transform a temporal value to another subtype
- setInterp: Transform a temporal value to another interpolation
- shiftTime, scaleTime, shiftScaleTime: Shift and/or scale the time span of a temporal value by one or two interval
- unnest: Transform a nonlinear temporal value into a set of rows, each one containing a base value and a period set during which the temporal value has the base value

A.4.6 Modifications

- insert: Insert a temporal value into another one
- update: Update a temporal value with another one
- deleteTime: Delete the instants of a temporal value that intersect a time value
- beforeTimestamp, afterTimestamp: Keep the instants of a temporal value that are before/after or equal a timestamp
- appendInstant: Append a temporal instant to a temporal value
- appendSequence: Append a temporal sequence to a temporal value
- merge: Merge temporal values

A.4.7 Restrictions

- atValues, minusValues: Restrict to (the complement of) a set of values
- atTime, minusTime: Restrict to (the complement of) a time value

A.4.8 Comparisons

- =, <>, <, >, <=, >=, ttype_cmp: Traditional comparisons
- ?=, %=, ?<>, %<>, ?<, %<, ?>, %>, ?<=, %<=, ?>=, %>=: Ever and always comparisons
- #=, #<>, #<, #>, #<=, #>=: Temporal comparisons

A.4.9 Miscellaneous

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

A.5 Temporal Alphanumeric Types

A.5.1 Conversions

- tnumber::{span,tbox}, valueSpan(tnumber), timeSpan(tnumber), tbox(tnumber): Convert a temporal number to a span or to a temporal box
- tbool::tint, tint(tbool): Convert between a temporal boolean and a temporal integer
- tint::tfloat, tfloat::tint, tfloat(tint), tint(tfloat): Convert between a temporal integer and a temporal float

A.5.2 Accessors

- valueSet: Return the set of values of a temporal number
- minValue, maxValue, avgValue: Return the minimum, maximum or average value
- minInstant, maxInstant: Return the instant with the minimum or maximum value

A.5.3 Transformations

- shiftValue, scaleValue, shiftScaleValue: Shift and/or scale the value span of a temporal number by one or two values
- stops: Extract from a temporal float with linear interpolation the subsequences where the values stay within an span of a given width for at least a given duration

A.5.4 Restrictions

- atMin, atMax, minusMin, minusMax: Restrict to (the complement of) the minimum or maximum value
- atTbox, minusTbox: Restrict to (the complement of) a `tbox`

A.5.5 Boolean Operations

- &, |: Temporal and, temporal or
- ~: Temporal not
- whenTrue: Return the time when a temporal Boolean takes the value true

A.5.6 Mathematical Operations

- +, -, *, /: Temporal addition, subtraction, multiplication, and division
- abs: Return the absolute value of a temporal number
- floor, ceil: Round down or up to the nearest integer
- round: Round to a number of decimal places
- degrees, radians: Convert to degrees or radians
- deltaValue: Return the value difference between consecutive instants of a temporal number
- trend: Return the trend of a temporal float with linear interpolation, which states whether its value is increasing, constant, or decreasing, represented, respectively, by 1, 0, and -1.
- derivative: Return the derivative over time of a temporal float in units per second
- integral: Return the area under the curve
- twAvg: Return the time-weighted average
- ln, log10: Return the natural and base 10 logarithm of a temporal float
- exp: Return the exponential (e raised to the given power) of a temporal float

A.5.7 Text Operations

- ||: Text concatenation
- lower, upper, initcap: Transform in lowercase, uppercase, or initcap

A.6 Temporal Geometry Types

A.6.1 Input and Output

- asText, asEWKT: Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation
- asMFJSON: Return the Moving Features JSON (MF-JSON) representation
- asBinary, asEWKB, asHexEWKB: Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB), or the Hexadecimal Extended Well-Known Binary (EWKB) representation
- tspatialFromText, tspatialFromEWKT: Input from the Well-Known Text (WKT) or from the Extended Well-Known Text (EWKT) representation
- tspatialFromMFJSON: Input from the Moving Features JSON (MF-JSON) representation
- tspatialFromBinary, tspatialFromEWKB, tspatialFromHexEWKB: Input from the Well-Known Binary (WKB), from the Extended Well-Known Binary (EWKB), or from the Hexadecimal Extended Well-Known Binary (EWKB) representation

A.6.2 Conversions

- ::, stbox(tspatial): Convert a spatiotemporal value to a spatiotemporal box
- tgeometry::tgeography, tgeography::tgeometry, tgeompoint::tgeogpoint, tgeogpoint::tgeompoint: Convert between a temporal geometry and a temporal geography
- tgeompoint::geometry, tgeogpoint::geography, geometry::tgeompoint, geography::tgeogpoint: Convert between a temporal point and a PostGIS trajectory

A.6.3 Spatial Reference System

- SRID, setSRID: Return or set the spatial reference identifier
- transform, transformPipeline: Transform to a spatial reference identifier

A.6.4 Accessors

- trajectory, traversedArea: Return the trajectory or the traversed area
- twCentroid: Return the time-weighted centroid
- getX, getY, getZ: Return the X/Y/Z/T coordinate values as a temporal float
- isSimple: Return true if the temporal point does not spatially self-intersect
- length: Return the length traversed by the temporal point
- cumulativeLength: Return the cumulative length traversed by the temporal point
- speed: Return the speed of the temporal point in units per second
- direction: Return the direction
- azimuth: Return the temporal azimuth
- angularDifference: Return the temporal angular difference
- bearing: Return the temporal bearing

A.6.5 Transformations

- round: Round the coordinate values to a number of decimal places
- makeSimple: Return an array of fragments of the temporal point which are simple
- geoMeasure: Construct a geometry/geography with M measure from a temporal point and a temporal float
- affine: Return the 3D affine transform of a temporal point to do things like translate, rotate, scale in one step
- rotate: Return a temporal point rotated counter-clockwise about the origin point
- scale: Return a temporal point scaled by given factors
- asMVTGeom: Transform a temporal geometric point into the coordinate space of a Mapbox Vector Tile
- stops: Extract from a temporal point with linear interpolation the subsequences where the point stays within an area with a specified maximum size for at least the given duration

A.6.6 Restrictions

- atGeometry, minusGeometry: Restrict to (the complement of) a geometry, a span in Z and/or a period
- atStbox, minuStbox: Restrict to (the complement of) an `stbox`

A.6.7 Distance Operations

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

A.6.8 Spatial Relationships

A.6.8.1 Ever or Always Spatial Relationships

- eContains, aContains: Ever or always contains
- eDisjoint, aDisjoint: Is ever or always disjoint
- eDwithin, aDwithin: Is ever or always at distance within
- eIntersects, aIntersects: Ever or always intersects
- eTouches, aTouches: Ever or always touches

A.6.8.2 Spatiotemporal Relationships

- tContains: Temporal contains
- tDisjoint: Temporal disjoint
- tDwithin: Temporal distance within
- tIntersects: Temporal intersects
- tTouches: Temporal touches

A.7 Operations for Temporal Types: Analytics Operations

A.7.1 Simplification

- minDistSimplify, minTimeDeltaSimplify: Return a temporal float or a temporal point simplified ensuring that consecutive values are at least a certain distance or time interval apart
- maxDistSimplify, douglasPeuckerSimplify: Return a temporal float or a temporal point simplified using the Douglas-Peucker algorithm

A.7.2 Reduction

- tsample: Sample a temporal value with respect to an interval
- tprecision: Reduce the temporal precision of a temporal value with respect to an interval computing the time-weighted average/centroid in each time bin

A.7.3 Similarity

- hausdorffDistance, frechetDistance, dynTimeWarpDistance: Return the discrete Hausdorff distance, the discrete Fréchet distance, or the Dynamic Time Warp distance between two temporal values
- frechetDistancePath, dynTimeWarpPath: Return the correspondence pairs between two temporal values with respect to the discrete Fréchet distance or the Dynamic Time Warp distance

A.7.4 Splitting Operations

- splitNSpans: Return an array of N time spans obtained by merging the instants or segments of a temporal value
- splitEachNSpans: Return an array of time spans obtained by merging N consecutive instants or segments of a temporal value
- splitNTboxes: Return an array of N temporal boxes obtained by merging the instants or segments of a temporal number
- splitEachNTboxes: Return an array of temporal boxes obtained by merging N consecutive instants or segments of a temporal number
- splitNStboxes: Return either an array of N spatial boxes obtained by merging the segments of a (multi)line or an array of N spatiotemporal boxes obtained by merging the instants or segments of a temporal point
- splitEachNStboxes: Return either an array of spatial boxes obtained by merging N consecutive segments of a (multi)line or an array of spatiotemporal boxes obtained by merging N consecutive instants or segments of a temporal point

A.7.5 Multidimensional Tiling

A.7.5.1 Bin Operations

- bins: Return a set of bins that cover a value or time span with bins of the same width or duration
- getBin: Return the bin that contains a number or a timestamp

A.7.5.2 Tile Operations

- valueTiles, timeTiles, valueTimeTiles: Return the set of tiles that cover a temporal box with tiles of the same size and/or duration
- spaceTiles, timeTiles, spaceTimeTiles: Return the set of tiles that cover a spatiotemporal box with tiles of the same size and/or duration
- getValueTile, getTboxTimeTile, getValueTimeTile: Return the temporal tile that covers a value and/or a timestamp
- getSpaceTile, getStboxTimeTile, getSpaceTimeTile: Return the spatiotemporal tile that covers a point and/or a timestamp

A.7.5.3 Bounding Box Operations

- valueBins, timeBins: Return an array of value or times spans obtained from the instants or segments of a temporal number with respect to a value or time grid
- valueBoxes, timeBoxes, valueTimeBoxes: Return an array of temporal boxes obtained from the instants or segments of a temporal number with respect to a value and/or time grid
- spaceBoxes, timeBoxes, spaceTimeBoxes: Return an array of spatiotemporal boxes obtained from the instants or segments of a temporal point with respect to a space and/or time grid

A.7.5.4 Splitting Operations

- valueSplit, timeSplit, valueTimeSplit: Split a temporal number with respect to value and/or time bins
- spaceSplit, spaceTimeSplit: Split a temporal point with respect to a spatial or spatiotemporal grid

A.7.6 Aggregations

- tCount: Temporal count
- extent: Bounding box extent
- tAnd, tOr: Temporal and, temporal or
- tMin, tMax, tSum, tAvg: Temporal minimum, maximum, sum, and average
- wCount: Window count
- wMin, wMax, wSum, wAvg: Window minimum, maximum, sum, and average
- tCentroid: Temporal centroid

A.8 Temporal Poses

A.8.1 Static Poses

A.8.1.1 Input and Output

- asText, asEWKT: Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation
- asBinary, asEWKB, asHexEWKB: Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB), or the Hexadecimal Extended Well-Known Binary (HexEWKB) representation
- poseFromText, poseFromEWKT: Input from the Well-Known Text (WKT) or from the Extended Well-Known Text (EWKT) representation
- poseFromBinary, poseFromEWKB, poseFromHexEWKB: Input from the Well-Known Binary (WKB), from the Extended Well-Known Binary (EWKB), or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

A.8.1.2 Constructors

- pose: Constructors for poses

A.8.1.3 Conversions

- pose::stbox, stbox(pose): Convert a pose and, optionally, a timestamp or a period, into a spatiotemporal box
- pose::geometry: Convert a pose into a geometry point

A.8.1.4 Accessors

- point: Return the point
- rotation: Return the rotation
- orientation: Return the orientation

A.8.1.5 Transformations

- round: Round the point and the orientation of the pose to the number of decimal places

A.8.1.6 Spatial Reference System

- SRID, setSRID: Return or set the spatial reference identifier
- transform, transformPipeline: Transform to a spatial reference identifier
- same: Spatial similarity

A.8.1.7 Comparisons

- =, <>, <, >, <=, >=: Traditional comparisons

A.8.2 Temporal Poses

A.8.2.1 Input and Output

- asText, asEWKT: Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation
- asMFJSON: Return the Moving Features JSON (MF-JSON) representation
- asBinary, asEWKB, asHexEWKB: Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB), or the Hexadecimal Extended Well-Known Binary (HexEWKB) representation
- tposeFromText, tposeFromEWKT: Input from the Well-Known Text (WKT) or from the Extended Well-Known Text (EWKT) representation
- tposeFromMFJSON: Input from the Moving Features JSON (MF-JSON) representation
- tposeFromBinary, tposeFromEWKB , tposeFromHexEWKB: Input from the Well-Known Binary (WKB), from the Extended Well-Known Binary (EWKB), or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

A.8.2.2 Constructors

- tpose: Constructor for temporal poses from a base value and a time value
- tposeSeq: Constructors for temporal poses of sequence subtype
- tposeSeqSet, tposeSeqSetGaps: Constructor for temporal poses of sequence set subtype
- tpose: Construct a 2D temporal pose from a temporal geometry point and a temporal float

A.8.2.3 Conversions

- tpose::tgeompoint: Convert a temporal pose into a temporal geometry point

A.8.2.4 Accessors

- getValues: Return the values
- points: Return the points
- rotation: Return the rotation
- valueAtTimestamp: Return the value at a timestamp

A.8.2.5 Transformations

- tposeInst, tposeSeq , tposeSeqSet: Transform to another subtype
- setInterp: Transform to another interpolation
- round: Round the points and the orientations of the temporal pose to the number of decimal places

A.8.2.6 Modifications

- appendInstant, appendSequence: Append a temporal instant or a temporal sequence
- merge: Merge the temporal poses

A.8.2.7 Restrictions

- atValues, minusValues: Restrict to (the complement of) a set of values
- atGeometry, minusGeometry: Restrict to (the complement of) a geometry

A.8.2.8 Spatial Reference System

- SRID, setSRID: Return or set the spatial reference identifier
- transform, transformPipeline: Transform to a spatial reference identifier

A.8.2.9 Bounding Box Operations

- &&, <@, @>, ~=, -|-: Topological operators
- <<, &<, >>, &>, <<|, &<|, |>>, |&>, <<#, &<#, #>>, |&>: Position operators

A.8.2.10 Distance Operations

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

A.8.2.11 Comparisons

- =, <>, <, >, <=, >=: Traditional comparisons
- ?=, &=: Ever and always comparisons
- #=, #<>: Temporal comparisons

A.8.2.12 Aggregations

- tCount: Temporal count
- wCount: Window count

A.9 Temporal Network Points

A.9.1 Static Network Types

A.9.1.1 Constructors

- npoint, nsegment: Constructors for network points and network segments

A.9.1.2 Conversions

- npoint::geometry, nsegment::geometry, geometry::npoint, geometry::nsegment: Convert between a network point or segment and a geometry

A.9.1.3 Accessors

- route: Return the route identifier
- getPosition: Return the position fraction
- startPosition, endPosition: Return the start or end position

A.9.1.4 Transformations

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

A.9.1.5 Spatial Operations

- SRID: Return the spatial reference identifier
- equals: Spatial similarity

A.9.1.6 Comparisons

- =, <>, <, >, <=, >=: Traditional comparisons

A.9.2 Temporal Network Points

A.9.2.1 Constructors

- tnpoint: Constructor for temporal network points from a base value and a time value
- tnpointSeq: Constructors for temporal network points of sequence subtype
- tnpointSeqSet, tnpointSeqSetGaps: Constructor for temporal network points of sequence set subtype

A.9.2.2 Conversions

- tnpoint::tgeompoint, tgeompoint::tnpoint: Convert between a temporal network point and a temporal geometry point

A.9.2.3 Accessors

- getValues: Return the values
- routes: Return the road identifiers
- valueAtTimestamp: Return the value at a timestamp
- length: Return the length traversed by the temporal network point
- cumulativeLength: Return the cumulative length traversed by the temporal network point
- speed: Return the speed of the temporal network point in units per second

A.9.2.4 Transformations

- tnpointInst, tnpointSeq, tnpointSeqSet: Transform a temporal network point to another subtype
- setInterp: Transform a temporal network point to another interpolation
- round: Round the fraction of the temporal network point to the number of decimal places

A.9.2.5 Restrictions

- atValues, minusValues: Restrict to (the complement of) a set of values
- atGeometry, minusGeometry: Restrict to (the complement of) a geometry

A.9.2.6 Bounding Box Operations

- &&, <@, @>, ~=, -|-: Topological operators
- <<, &<, >>, &>, <<|, &<|, |>>, |&>, <<#, &<#, #>>, |&>: Position operators

A.9.2.7 Distance Operations

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

A.9.2.8 Spatial Operations

- twCentroid: Return the time-weighted centroid

A.9.2.9 Comparisons

- =, <>, <, >, <=, >=: Traditional comparisons
- ?=, &=: Ever and always comparisons
- #=, #<>: Temporal comparisons

A.9.2.10 Aggregations

- tCount: Temporal count
- wCount: Window count
- tCentroid: Temporal centroid

A.10 Temporal Circular Buffers

A.10.1 Static Circular Buffers

A.10.1.1 Constructors

- cbuffer: Constructor for circular buffers

A.10.1.2 Conversions

- cbuffer::geometry, geometry::cbuffer: Convert between a circular buffer and a geometry point
- stbox: Convert a circular buffer and, optionally, a timestamp or a period, into a spatiotemporal box

A.10.1.3 Accessors

- point: Return the point
- radius: Return the radius

A.10.1.4 Transformations

- round: Round the point and the radius of the circular buffer to the number of decimal places

A.10.1.5 Spatial Operations

- SRID, setSRID: Return or set the spatial reference identifier
- transform, transformPipeline: Transform to a spatial reference identifier

A.10.1.6 Comparisons

- =, <>, <, >, <=, >=: Traditional comparisons

A.10.2 Temporal Circular Buffers

A.10.2.1 Input and Output

- asText, asEWKT: Return the Well-Known Text (WKT) or the Extended Well-Known Text (EWKT) representation
- asBinary, asEWKB: Return the Well-Known Binary (WKB), the Extended Well-Known Binary (EWKB), or the Hexadecimal Extended Well-Known Binary (HexEWKB) representation as text
- tcbufferFromText, tcbufferFromEWKT: Input from the Well-Known Text (WKT) or from the Extended Well-Known Text (EWKT) representation
- tcbufferFromBinary, tcbufferFromEWKB, tcbufferFromHexEWKB: Input from the Well-Known Binary (WKB), from the Extended Well-Known Binary (EWKB), or from the Hexadecimal Extended Well-Known Binary (HexEWKB) representation

A.10.2.2 Constructors

- tcbuffer: Constructor for temporal circular buffers having a constant value
- tcbufferSeq: Constructors for temporal circular buffers of sequence subtype
- tcbufferSeqSet, tcbufferSeqSetGaps: Constructor for temporal circular buffers of sequence set subtype
- tcbuffer: Construct a temporal circular buffer from a temporal geometry point and a temporal float

A.10.2.3 Conversions

- tcbuffer::tgeompoint: Convert a temporal circular buffer to a temporal geometry point
- tcbuffer::tfloat: Convert a temporal circular buffer to a temporal float

A.10.2.4 Accessors

- getValues: Return the values
- points, radius: Return the points or the radii
- valueAtTimestamp: Return the value at a timestamp

A.10.2.5 Transformations

- tcbufferInst, tcbufferSeq, tcbufferSeqSet: Transform a temporal circular buffer to another subtype
- setInterp: Transform a temporal circular buffer to another interpolation
- round: Round the points and the radii of the temporal circular buffer to the number of decimal places

A.10.2.6 Restrictions

- atValues, minusValue: Restrict to (the complement of) a set of values
- atGeometry, minusGeometry: Restrict to (the complement of) a geometry

A.10.2.7 Distance Operations

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

A.10.2.8 Spatial Operations

- SRID, setSRID: Return or set the spatial reference identifier
- transform, transformPipeline: Transform to a spatial reference identifier
- traversedArea: Return the area traversed by the temporal circular buffer

A.10.2.9 Bounding Box Operations

- &&, <@, @>, ~=, -|-: Topological operators
- <<, &<, >>, &>, <<|, &<|, |>>, |&>, <<#, &<#, #>>, |&>: Position operators

A.10.2.10 Spatial Relationships

- eContains, eDisjoint, eIntersects, eTouches, eDwithin: Possible spatial relationships
- tContains, tDisjoint, tIntersects, tTouches, tDwithin: Spatiotemporal relationships

A.10.2.11 Comparisons

- =, <>, <, >, <=, >=: Traditional comparisons
- ?=, &=: Ever and always comparisons
- #=, #<>: Temporal comparisons

A.10.2.12 Aggregations

- tCount: Temporal count
- wCount: Window count
- tCentroid: Temporal centroid

A.11 Temporal JSONB

A.11.1 Input and Output

- asText: Return the Well-Known Text (WKT) representation
- asBinary, asWKB: Return the Well-Known Binary (WKB) or the Hexadecimal Well-Known Binary (HexEWKB) representation as text
- asMFJSON: Return the Moving Features JSON (MF-JSON) representation as text
- tjsonbFromText, tjsonbFromWKT: Input from the Well-Known Text (WKT) representation
- tjsonbFromBinary, tjsonbFromHexWKB: Input from the Well-Known Binary (WKB) or from the Hexadecimal Well-Known Binary (HexEWKB) representation
- tjsonbFromMFJSON: Input from the Moving Features JSON (MF-JSON) representation

A.11.2 Constructors

- tjsonb: Constructor for temporal JSONB having a constant value
- tjsonbSeq: Constructors for temporal JSONB of sequence subtype
- tjsonbSeqSet, tjsonbSeqSetGaps: Constructor for temporal JSONB of sequence set subtype

A.11.3 Conversions

- tjsonb::tstzspan: Convert a temporal JSONB to a temporal geometry point
- tjsonb::ttext: Convert a temporal JSONB to a temporal text

A.11.4 Accessors

- getValues: Return the values
- valueAtTimestamp: Return the value at a timestamp

A.11.5 Transformations

- tjsonbInst, tjsonbSeq, tjsonbSeqSet: Transform a temporal JSONB to another subtype
- setInterp: Transform a temporal JSONB to another interpolation

A.11.6 Temporal JSON operations

- ->, ->>: Extract a field from a temporal JSON value specified by a key
- #>, #>>: Extract a field from a temporal JSON value specified by a path
- tjson_array_element, tjsonb_array_element, tjsonb_array_element_text: Extract an element from temporal JSON array
- ||: Temporal JSONB concatenation
- -, #-: Temporal JSONB deletion
- tjsonb_set, tjsonb_set_lax: Temporal JSONB set

- ?, ?|, ?&: Temporal JSONB exists
- @>, <@: Temporal JSONB contains/contained
- tbool, tint, tfloat, ttext: Extract a temporal alphanumeric value from a temporal JSONB value given by a key
- tjson_strip_nulls, tjsonb_strip_nulls: Return a temporal JSON value without nulls

A.11.7 Restrictions

- atValues, minusValue: Restrict to (the complement of) a set of values

A.11.8 Bounding Box Operations

- &&, <@, @>, ~=, -|-: Topological operators
- <<#, &<#, #>>, |&>: Position operators

A.11.9 Comparisons

- =, <>, <, >, <=, >=: Traditional comparisons
- ?=, &=: Ever and always comparisons
- #=, #<>: Temporal comparisons

A.11.10 Aggregations

- tCount: Temporal count
- wCount: Window count

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, 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_int4range`: Generate a random integer range
- `random_float`: Generate a random float
- `random_float_array`: Generate a random array of floats
- `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 Span, Time, and Box Types

- `random_intspan`: Generate a random integer span
- `random_floatspan`: Generate a random float span
- `random_tstzspan`: Generate a random `tstzspan`
- `random_tstzspan_array`: Generate a random array of `tstzspan` values
- `random_tstzset`: Generate a random `tstzset`
- `random_tstzspanset`: Generate a random `tstzspanset`
- `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_discseq`: Generate a random temporal Boolean of sequence subtype and discrete interpolation
- `random_tint_discseq`: Generate a random temporal integer of sequence subtype and discrete interpolation
- `random_tfloat_discseq`: Generate a random temporal float of sequence subtype and discrete interpolation
- `random_ttext_discseq`: Generate a random temporal text of sequence subtype and discrete interpolation
- `random_tgeompoint_discseq`: Generate a random temporal geometric 2D point of sequence subtype and discrete interpolation
- `random_tgeompoint3D_discseq`: Generate a random temporal geometric 3D point of sequence subtype and discrete interpolation
- `random_tgeogpoint_discseq`: Generate a random temporal geographic 2D point of sequence subtype and discrete interpolation
- `random_tgeogpoint3D_discseq`: Generate a random temporal geographic 3D point of sequence subtype and discrete interpolation
- `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 integer DEFAULT 100)
RETURNS text AS $$
DECLARE
  perc integer;
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_floatspan` containing random `floatspan` 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_floatspan AS
/* Add perc NULL values */
SELECT k, NULL AS f
FROM generate_series(1, perc) AS k UNION
SELECT k, random_floatspan(0, 100, 10)
FROM generate_series(perc+1, size) AS k;
```

The creation of a table `tbl_tstzset` containing random `tstzset` 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_tstzset AS
/* Add perc NULL values */
SELECT k, NULL AS ts
FROM generate_series(1, perc) AS k UNION
SELECT k, random_tstzset('2001-01-01', '2001-12-31', 10, 5, 10)
FROM generate_series(perc+1, size) AS k;
```

The creation of a table `tbl_tstzspan` containing random `tstzspan` 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_tstzspan AS
/* Add perc NULL values */
SELECT k, NULL AS p
FROM generate_series(1, perc) AS k UNION
SELECT k, random_tstzspan('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_discseq` containing random `tint` values of sequence subtype and discrete interpolation 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_discseq AS
/* Add perc NULL values */
SELECT k, NULL AS ti
FROM generate_series(1, perc) AS k UNION
SELECT k, random_tint_discseq(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_discseq t1
SET ti = (SELECT ti FROM tbl_tint_discseq 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_discseq t1
SET ti = (SELECT ti + random_int(1, 2) FROM tbl_tint_discseq 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_discseq t1
SET ti = (SELECT shift(ti, endTimestamp(ti)-startTimestamp(ti))
  FROM tbl_tint_discseq 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_discseq t1
SET ti = (SELECT shift(ti, date_trunc('minute',(endTimestamp(ti)-startTimestamp(ti))/2))
  FROM tbl_tint_discseq 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_discseq` containing random `tgeompoint` 2D values of sequence subtype and discrete interpolation 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_discseq AS
SELECT k, random_tgeompoint_discseq(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_discseq t1
SET ti = (SELECT ti FROM tbl_tgeompoint_discseq 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_discseq t1
SET ti = (SELECT round(ti,6) FROM tbl_tgeompoint_discseq 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_discseq t1
SET ti = (SELECT shift(ti, endTimestamp(ti)-startTimestamp(ti))
  FROM tbl_tgeompoint_discseq 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_discseq t1
SET ti = (SELECT shift(ti, date_trunc('minute',(endTimestamp(ti)-startTimestamp(ti))/2))
  FROM tbl_tgeompoint_discseq 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_discseq`: Generate a random temporal network point of sequence subtype and discrete interpolation
- `random_tnpoint_seq`: Generate a random temporal network point of sequence subtype and linear or step interpolation
- `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.