Info-H-415 - Advanced Databases

Mobility Data Science

Lesson 3: The Discrete Model

The discrete model

- The **abstract model** is implemented using a **discrete model**
- **Goal:** design & formally define finite representations for types and constructors of the abstract model
- Two discrete data models: Sliced- & Sequenced- based

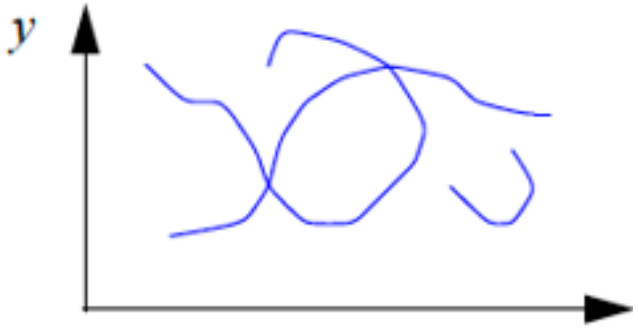
Two discrete data models

Sliced representation
Ralf Hartmut Güting, Michael H. Böhlen, Martin Erwig, Christian S. Jensen, Nikos A. Lorentzos, Markus Schneider, and Michalis Vazirgiannis. 2000. A foundation for representing and querying moving objects. *ACM Transactions on Database Systems*. 25, 1 (March 2000), 1–42. DOI:https://doi.org/10.1145/352958.352963

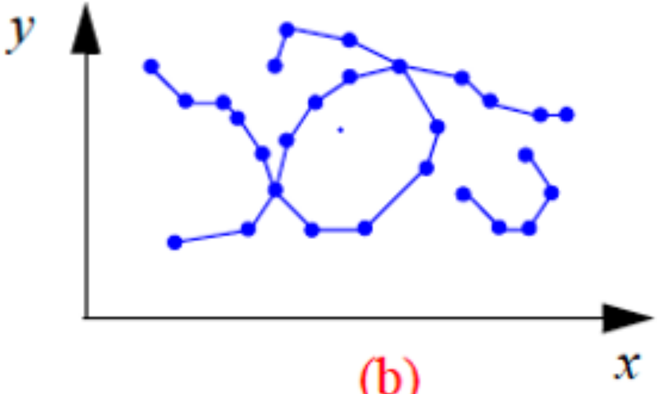
Sequence representation
Esteban Zimányi, Mahmoud Sakr, Arthur Lesuisse, MobilityDB: A Mobility Database based on PostgreSQL and PostGIS. *ACM Transactions on Database* Systems, 45(4): 19:1-19:42 (2020). Preprint

Non-temporal data types in the sliced representation

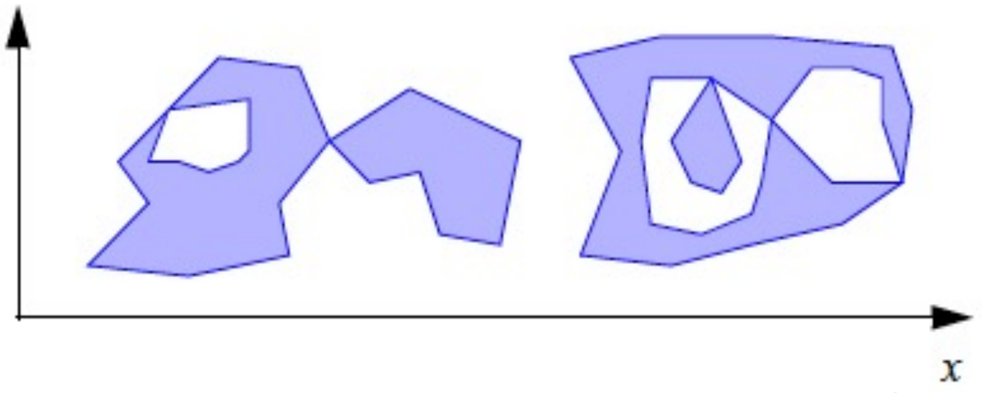
- Base types (*int, real, string, bool*) of the abstract model: types of a programming language
- Spatial abstract types *point* & *points* have direct representations
- Discrete versions of *line* and *region* based on linear approximations
  - E.g., a *curve* is represented by a polyline (b)
- A *region* in the discrete model is a finite set of simple polygons each of which may have polygonal holes



(a)

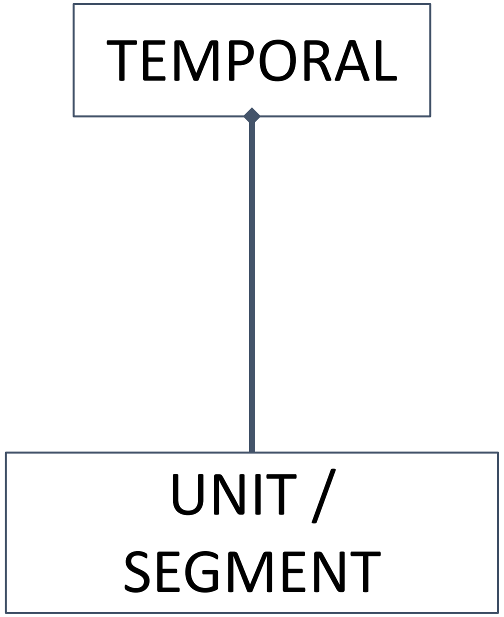


(b)



Temporal types: the sliced representation

- For *temporal* types the **sliced representation** is used
- Decomposes the evolution of values along the time dimension, into fragment intervals called *slices*
- Within each slice, the movement is represented by a simple function
- The UNIT type constructor is a pair (*time interval*, *function*) where *function* describes the evolution of the object during the *time interval*
- No constructor automatically **assembles** the static types, a type constructor called *mapping* is defined



Temporal data types in the sliced representation

- Correspondence between the abstract and discrete temporal types:

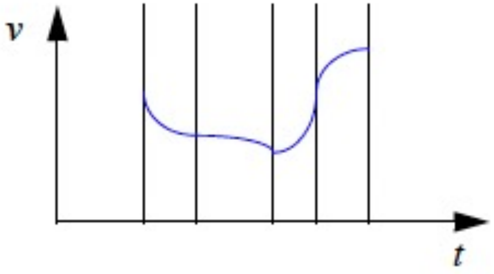
Abstract type	*Discrete type*
temporal(int)	*Mapping(const(int))*
temporal(text)	*Mapping(const(text))*
temporal(bool)	*Mapping(const(bool))*
temporal(float)	*Mapping(ufloat)*
temporal(point)	*Mapping(upoint)*
temporal(points)	*Mapping(upoints)*
temporal(line)	*Mapping(uline)*
temporal(region)	*Mapping(uregion)*

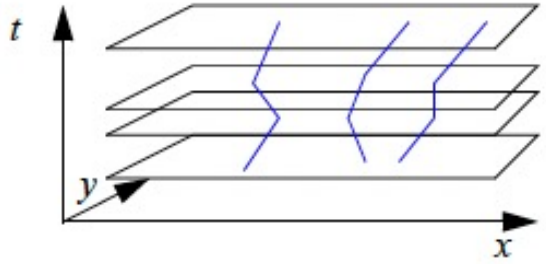
Temporal data types in the sliced representation

- Data types *int, text, bool* only admit **discrete** changes
  - The simple function is the value itself built by constructor *const*
  - Units are *const(int*), *const(text*), *const(bool*)
- The type constructor *const* applies to base types to represent **stepwise constant** developments
- The *const* constructor yields unit types with a constant value during the unit interval

Temporal data types in the sliced representation

- For **continuous** changes, within each slice the development is represented by a simple function
- Sliced representation for a *temporal(real)* (left); sliced representation for a *temporal(point)* (right)





- A *unit* is a p *ing T)*
- Type *ufloat* represents pieces of a ***temporal(float),*** producing values:

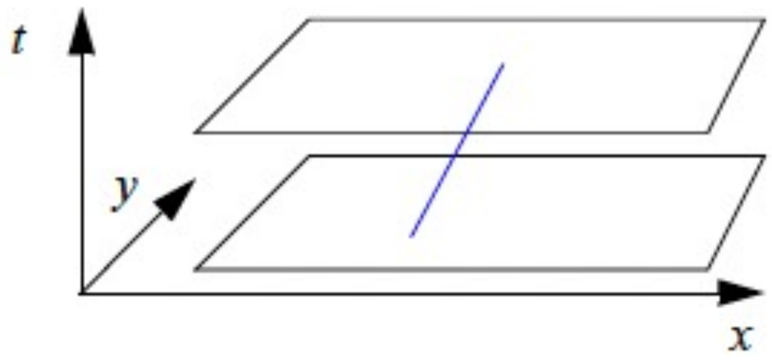
  D_{ufloat} = *Interval(instant)* x {(a, b, c, r) | a,b,c $\in$ *real*} where a, b, c are coefficients of a quadratic function over t (time)

- The constructor *mapping* assembles the units
- A *temporal(float)* as a whole is represented by the type ***mapping(ufloat)***

Temporal units for spatial types

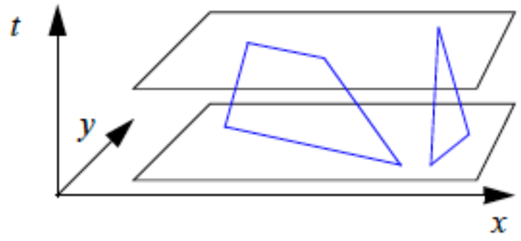
- *tpoint* is a set whose elements describe 2D points moving as linear functions of time
- This defines a unit *upoint*
- The carrier set (the values) of *upoint* is

$$D_{upoint} = Interval(instant) \times tpoint$$

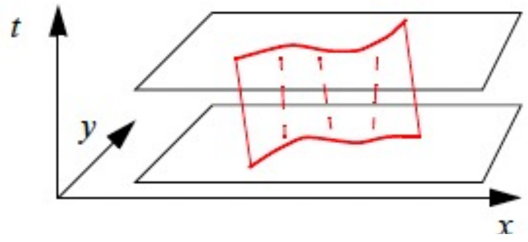


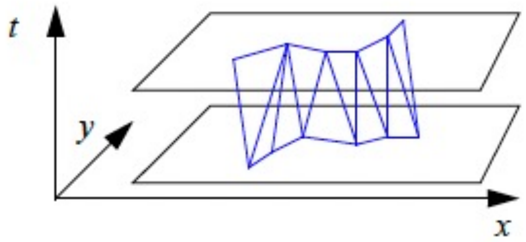
Temporal units for spatial types

- Segments do not turn while moving
- Two moving segments

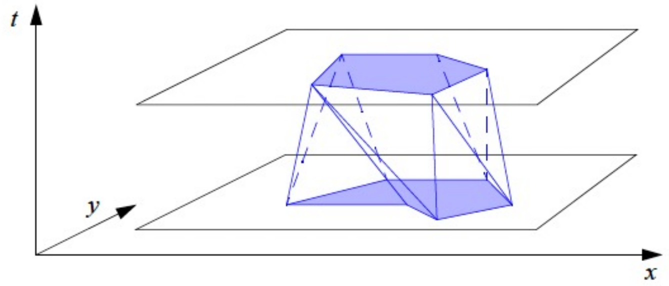


- A moving line is represented as a set of moving segments
- A moving curve approximated as a line unit



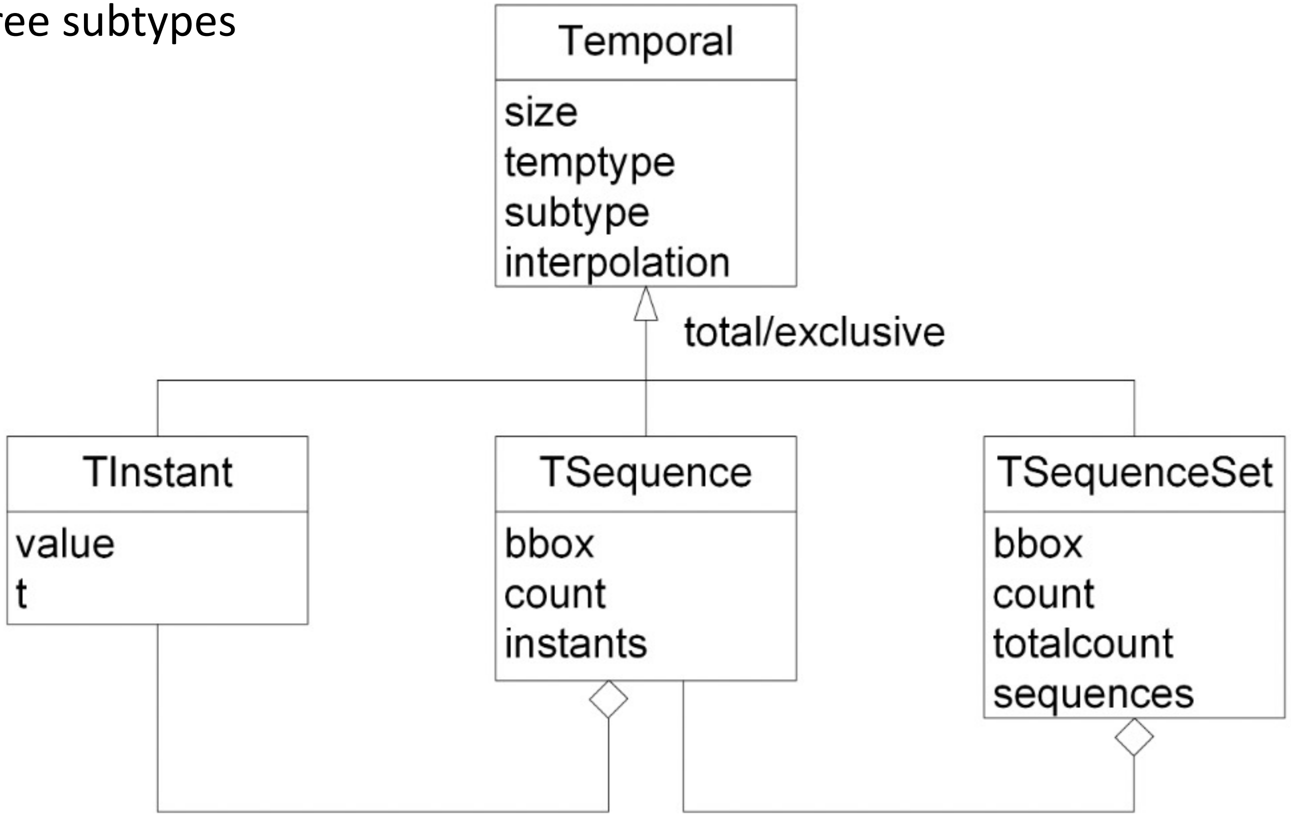


- A *region* unit is a set of moving faces, composed of moving cycles



Discrete model: Sequence representation

One **Temporal** type and three subtypes



https://libmeos.org/documentation/datastructures/

Data types in the sequence representation

TEMPORAL(timeType, baseType): time → value

timeType	baseType
timestamptz	geometry(point)
tstzset	geography(point)
tstzspan	bool
tstzspanset	float
	int
	string
	...

TEMPORAL(timestamptz, geometry(point))

//ex.: database of car accidents

TEMPORAL(tstzset, geometry(point))

//ex.: foursquare check-ins

TEMPORAL(tstzspan, geometry(point))

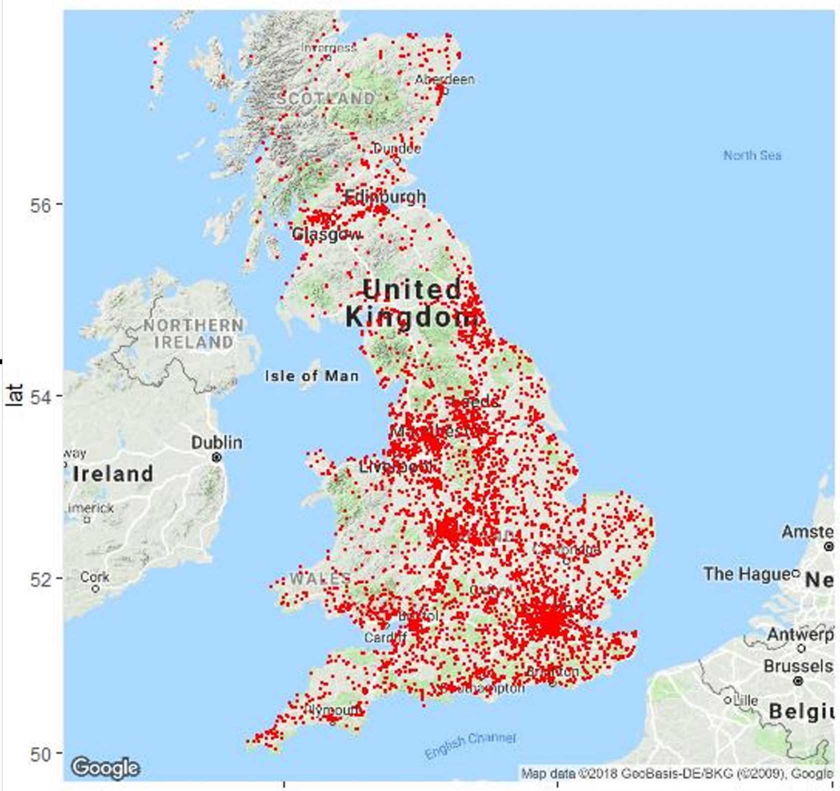
//ex.: car trajectory

...

Mobility Data: Points

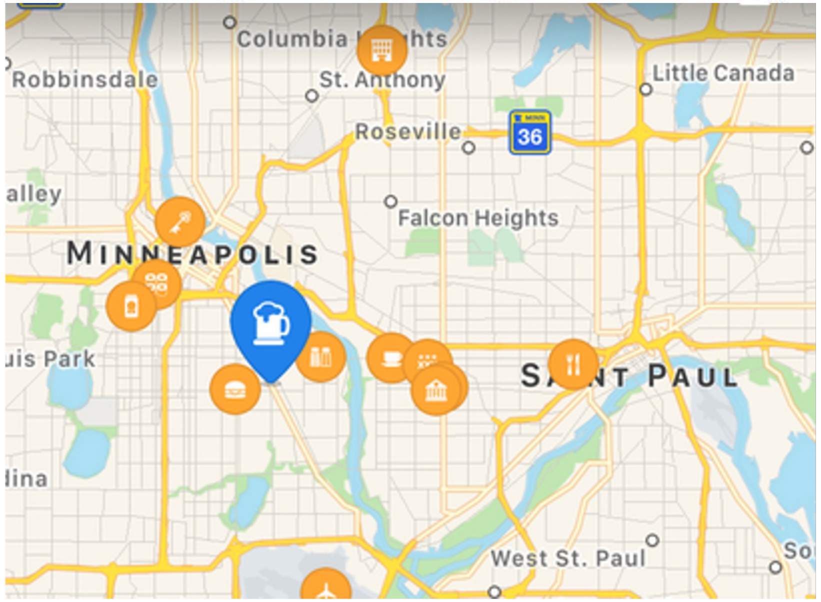
tgeogpoint(inst): UK road accidents 2012-

https://www.kaggle.com/daveianhickey/2000-16-traffic-flow-england-scotland-wales



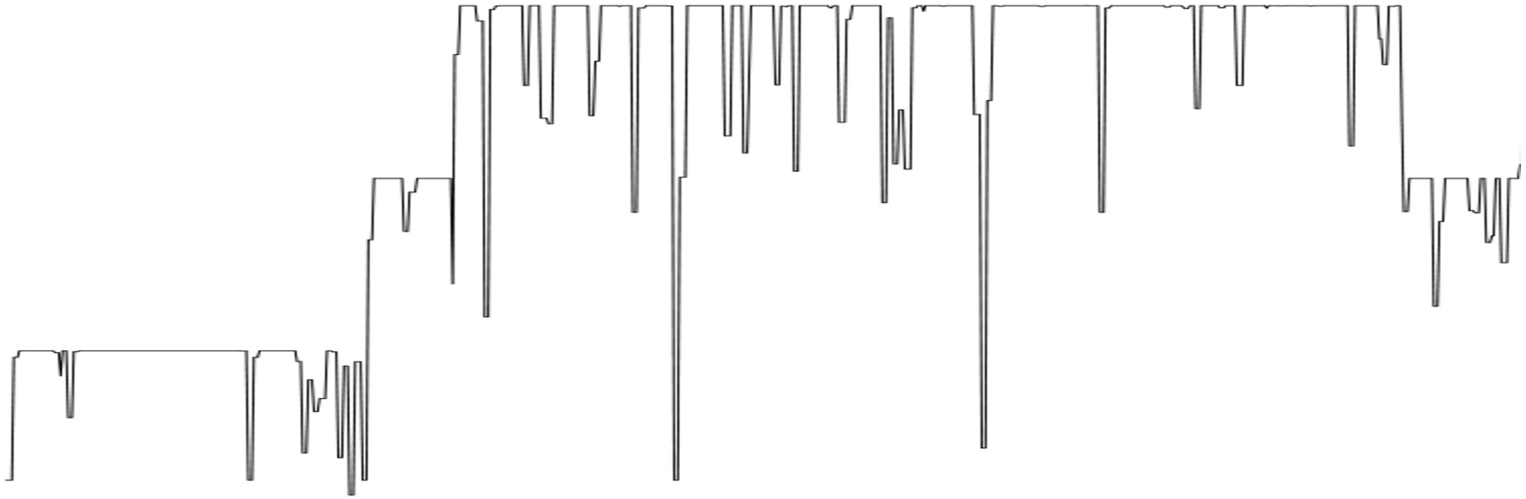
tgeogpoint(instants): foursquare check-ins

https://support.foursquare.com/

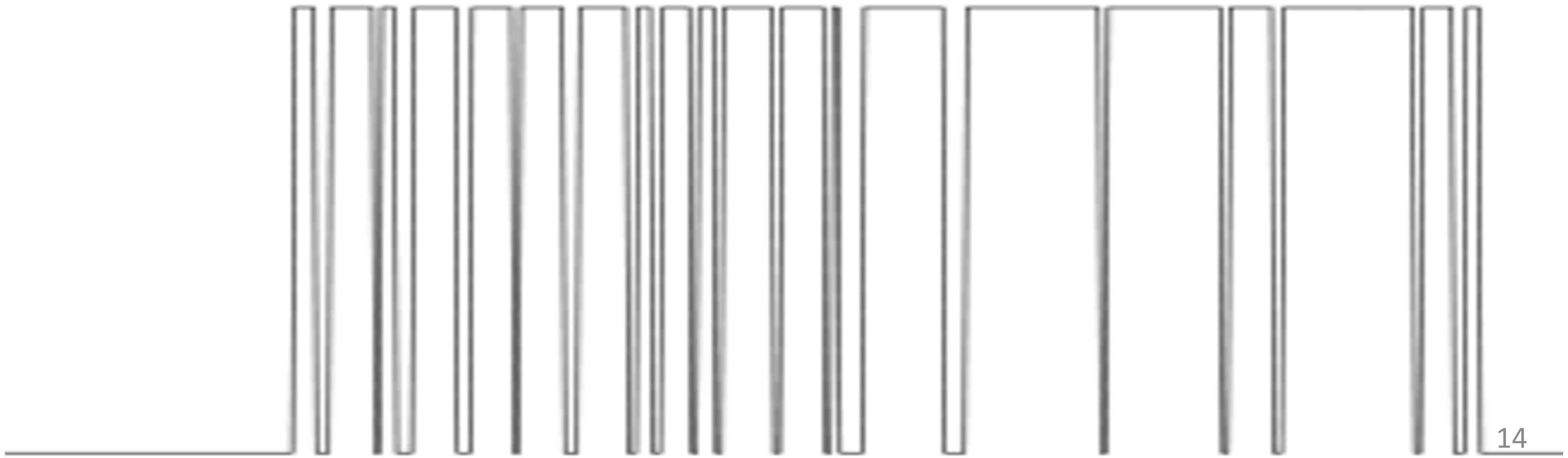


Mobility Data: Temporal Types

tfloat: speed(Trip).



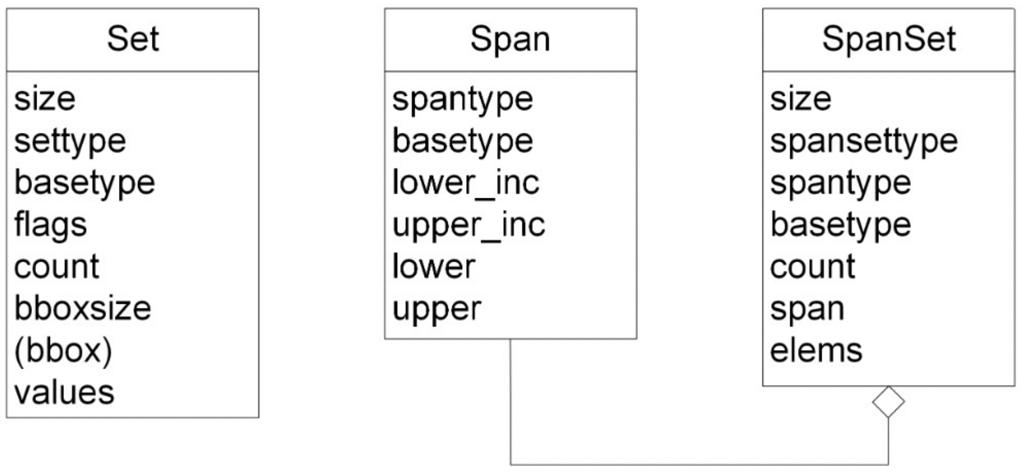
tbool: speed(Trip) > 90



Collection types and constructors in the sequence representation

- **Notation**

- *time:* any time type, that is, timestamptz, *tstzspan*, *tstzset*, or *tstzspanset*
- *number* represents any number type, that is, *integer* or *float*
- *set* represents a set of values
- *span*: represents a range of values of different types, e.g., *intspan, floatspan, tstzspan*
- *spanset*: represent sets of ranges of values, e.g., *intspanset, floatspanset, tstzspanset*
- *spans*: represents any span or spanset type
- *type*[] represents an array of *type*



Time types and constructors

- Based on the timestamptz type (PostgreSQL) and three new types: *tstzset, tstzspan and tstzspanset*.
- A value of *period* type has two bounds, the lower bound and the upper bound, timestamptz values. The constructor is:

tstzspan`('[timestamptz,timestamptz)')`: *tstzspan*
tstzspan`('[timestamptz,timestamptz]')`: *tstzspan*
tstzspan`('(timestamptz,timestamptz)')`: *tstzspan*
tstzspan`('(timestamptz,timestamptz]')`: *tstzspan*

For instance:

`SELECT tstzspan ('[2001-01-01 08:00:00, 2001-01-03 09:30:00)’)`

`SELECT tstzspan '[2001-01-01 08:00:00, 2001-01-03 09:30:00)'` -- alternative option without parenthesis

`SELECT '[2001-01-01 08:00:00, 2001-01-03 09:30:00)'::tstzspan` -- alternative option with casting

Time types and constructors

- The *tstzset* type represents a set of **different** *timestamptz* values (sets of instants)
- The *tstzspanset* type represents a set of **disjoint** *tstzspan* values (sets of intervals)

- Constructors for *tstzspanset* and *tstzset*

tstzset`('{ list of comma-separated timestamptz }'):` *tstzset*
tstzspanset`('{ list of comma-separated tstzspan }'):` *tstzspanset*

(alternative options are also available: without parenthesis/with casting options)

For instance:

```
SELECT tstzset ('{2001-01-01 08:00:00, 2001-01-03 09:30:00, 2001-01-03 012:30:00}')
SELECT tstzspanset('{[2001-01-01 08:00, 2001-01-01 08:10], [2001-01-01 08:20, 2001-01-01
       08:40]}')
```

- Can be accessed through accessor functions. Eg.:

```
duration({tstzspan, tstzspanset}): interval
SELECT duration(tstzspan '[2001-01-01 08:00:00, 2001-01-03 09:30:00)') --2 days 01:30:00
```

Time types and constructors

- **Accessor functions**

- Get the lower bound `lower(tstzspan): timestamptz`
  ```
  SELECT lower(tstzspan '[2011-01-01, 2011-01-05)'); -- "2011-01-01"
  ```
- Get the upper bound `upper(tstzspan): timestamptz`
  ```
  SELECT upper(tstzspan '[2011-01-01, 2011-01-05)' ); -- "2011-01-05"
  ```
- Is the lower bound inclusive? lower_inc(`tstzspan`): boolean
  ```
  SELECT lower_inc(tstzspan '[2011-01-01, 2011-01-05)');  -- true
  ```
- Get the duration. `duration({tstzspan, tstzspanset}): interval`
  ```
  SELECT duration(tstzspan '[2012-01-01 8:00:00, 2012-01-03 10:00:00 )');-- 2 days 02:00:00
  SELECT duration(tstzspanset '{[2012-01-01,2012-01-02),[2012-01-04, 2012-01-05)}’);-- 2days
  ```
- Get the timespan ignoring the potential time gaps. `span({tstzset, tstzspanset}): interval`
  ```
  SELECT span(tstzset '{2012-01-01, 2012-01-03}'); -- [2012-01-01, 2012-01-03]
  SELECT span(tstzspanset '{[2012-01-01,2012-01-02),[2012-01-04, 2012-01-05)}'); -- [2012-
         01-01,2012-01-05)
  ```
- Get the start timestamp `startTimestamp({tstzset, tstzspanset}): timestamptz`
  ```
  SELECT startTimestamp(tstzspanset '{[2012-01-01, 2012-01-03), (2012-01-03, 2012-01-05)}');
  -- "2012-01-01"
  ```

Time types operations

- **Topological operators for time types**

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

  `{tstzset, tstzspan, tstzspanset} && {tstzset, tstzspan, tstzspanset}: Boolean`

  First time value contains the second one?

  `{tstzset, tstzspan, tstzspanset} @> time: Boolean`

  First time value contained by the second one? `time <@ {tstzset, tstzspan, tstzspanset}: Boolean`

  `SELECT timestamptz '[2011-02-01]' <@ tstzspan '[2011-01-01, 2011-05-01)';-- true`

- **Aggregate operators for time types**

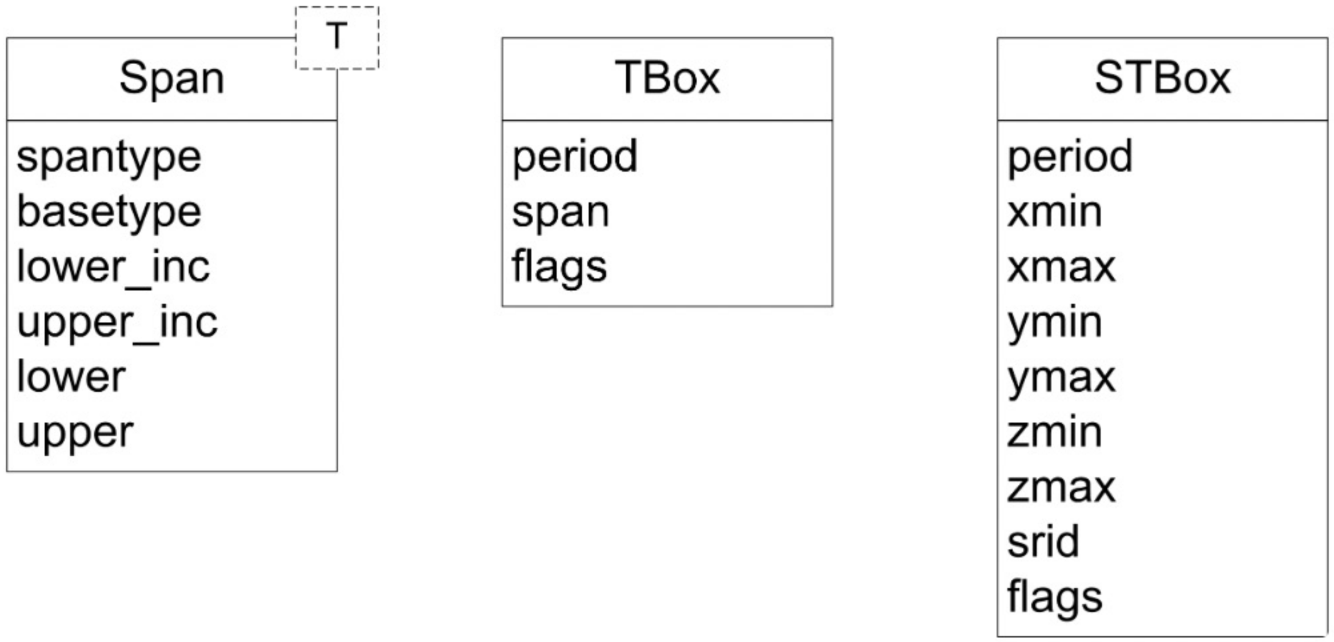
  Function `extent` returns a bounding period that encloses a set of time values

  `extent({tstzset, tstzspan, tstzspanset}): span; extent(range): range`

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

Bounding box types

- Bounding boxes are used for efficient manipulation of temporal types.
- For example, when determining whether a temporal point (e.g., a moving vehicle) overlaps a geometry (e.g., a county), a bounding box test is applied to quickly filter out the temporal points whose bounding box does not overlap the bounding box of the geometry.



https://libmeos.org/dc

Bounding box types: tbox and stbox

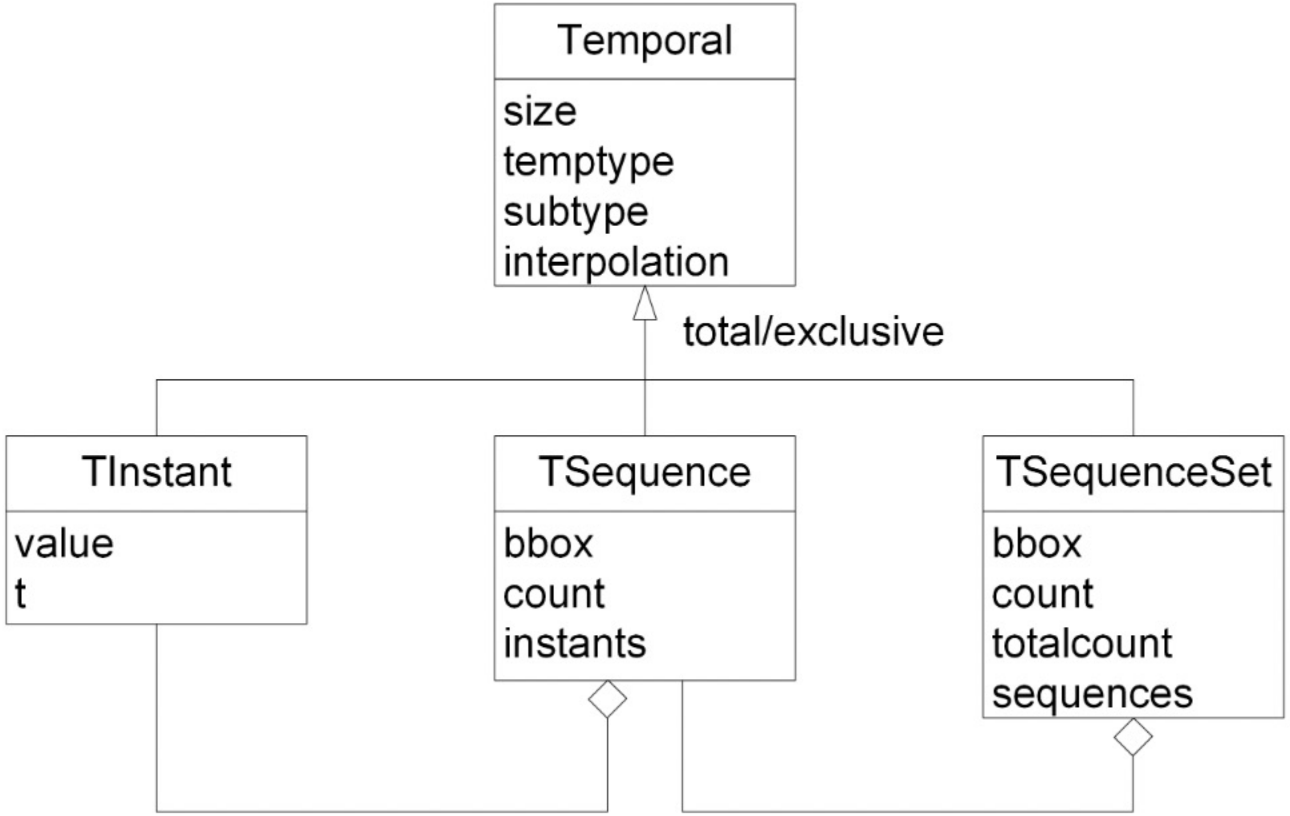
- *Tbox*: composed of a numeric and/or time dimensions. For each dimension, a span is given, e.g., a *floatspan* for the value dimension and a *span* for the time dimension

```
--  tbox for value and time dimensions
SELECT tbox 'TBOX XT([1.0,2.0],[2000-01-01,2000-01-02])';
SELECT tbox(floatspan '[1.0, 2.0]', tstzspan '[2001-01-01,2001-01-02]'); --TBOXFLOAT XT(
      [1, 2],[2000-01-01 00:00:00-03, 2000-01-02 00:00:00-03])
--  tbox for value dimension (min and max values for X)
SELECT tbox 'TBOX X((1.0,2.0))';
--  tbox for time dimension
SELECT tbox 'TBOX T((2000-01-01,2000-01-02))';
```

- *stbox*: composed of a spatial (2, 3D) and/or time dimensions
  For the temporal dimension a *span* is given
  For the spatial dimension minimum and maximum coordinate values are given
  SRID of the coordinates needed,   0 (Cartesian) and 4326 (geodetic)

```
-- 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])';
-- SRID is given
SELECT stbox 'SRID=5676;STBOX XT(((1.0,2.0),(1.0,2.0)),[2001-01-04,2001-01-04])';
```

Discrete model: Sequence representation reminder



Temporal types in the sequence representation reminder

TEMPORAL(timeType, baseType): time → value

Timestamptz
tstzset
tstzspan
tstzspanset

geometry(point)
geography(point)
bool
float
int
string
...

TEMPORAL(timestamptz, geometry(point))
//ex.: database of car accidents
TEMPORAL(tstzset, geometry(point))
//ex.: foursquare check-ins
TEMPORAL(tstzspan, geometry(point))
//ex.: car trajectory
...

Temporal data types in the sequence representation

- Six built-in temporal types: *tbool, tint, tfloat, ttext, tgeompoint, tgeogpoint,* based on the base types *bool, int, float, text, geometry, geography* (restricted to 2D or 3D points with Z dimension)

- Data are discrete => interpolation function needed (UNITS in the sliced representation)

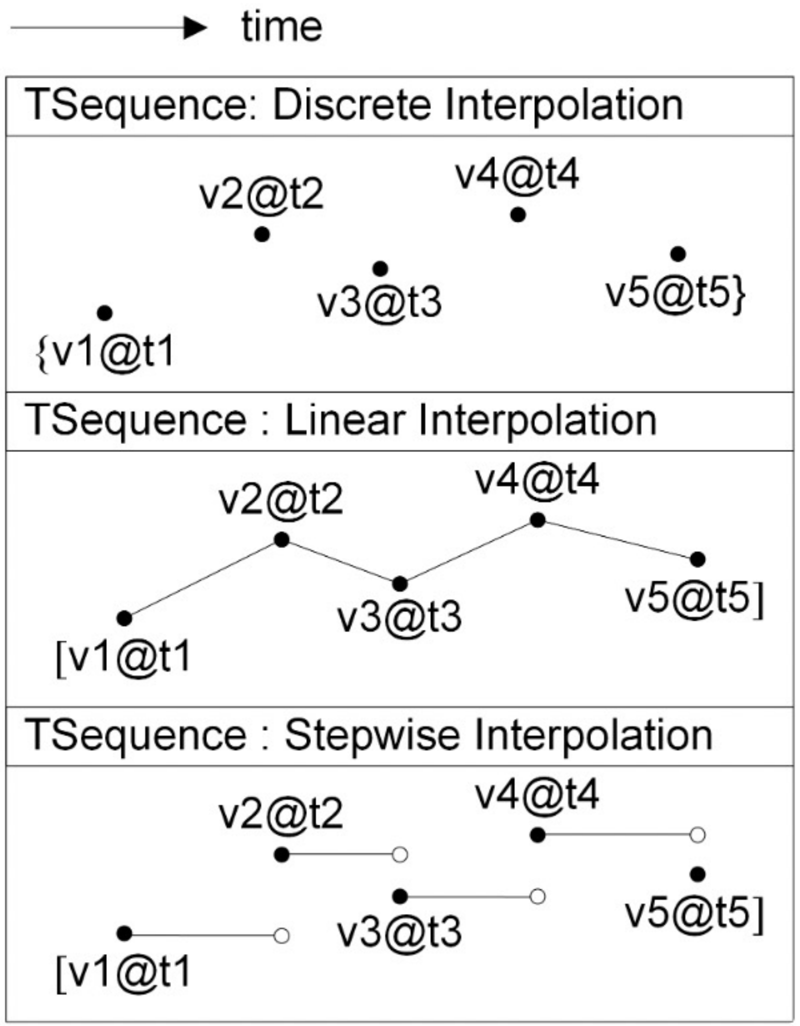
- The interpolation of a temporal value states how the value evolves between successive instants.

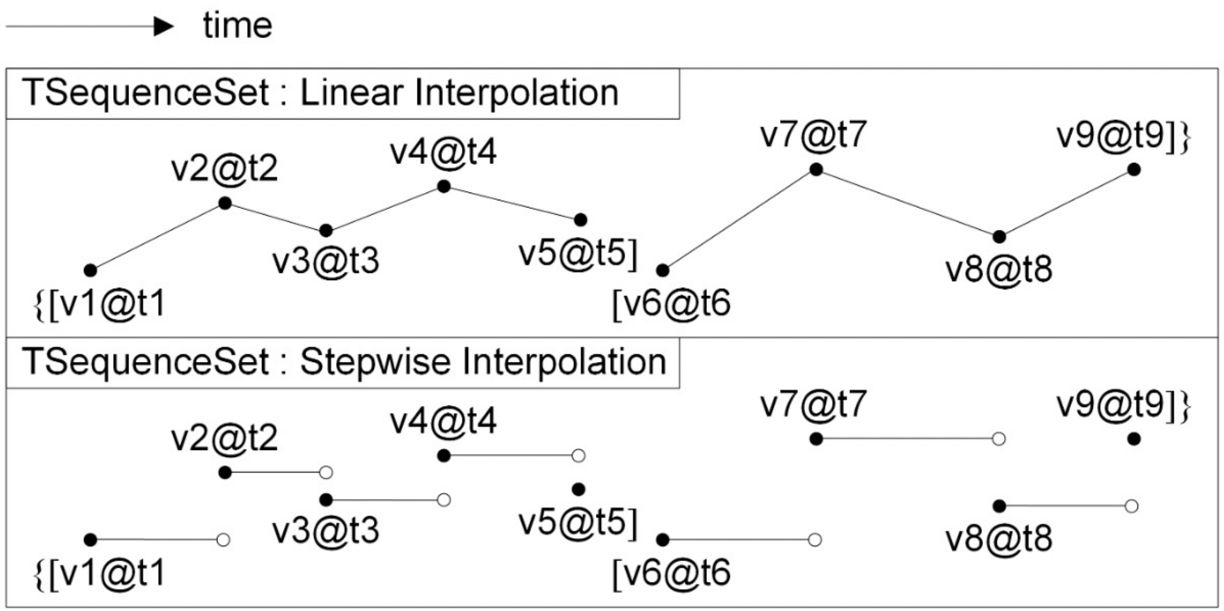
  - **Discrete**: No value can be inferred.

  - **Stepwise**: the value remains constant between two successive instants. (e.g., # of employees in a department represented by a temporal integer, e.g., constant between two instants)

  - **Linear**: the value evolves linearly between two successive instants (e.g., temperature of a room may be represented with a temporal float)
- Types based on **discrete** base types (*tbool, tint, ttext*) evolve necessarily in a stepwise manner;
- Types based on **continuous** base types (*tfloat, tgeompoint, tgeogpoint*) may evolve stepwisely or linearly.

Temporal data types: interpolation





Note: only linear or stepwise interpolation for sequence set values

Temporal data types

- Temporal values come in three subtypes: **instant, sequence, and sequence set**
- A temporal value of ***instant*** subtype represents the value at a time instant

```
SELECT tfloat '17@2018-01-01 08:00:00'; --17@2018-01-01 08:00:00-03
```

- A temporal value of ***sequence*** subtype represents the evolution of the value during a sequence of time instants; values between these instants are interpolated using a discrete, stepwise, or a linear function.

```
SELECT tfloat '{17@2018-01-01 08:00:00, 17.5@2018-01-01 08:05:00, 18@2018-01-01 08:10:00}'
-- Set of instant values (curly brackets, only Discrete interpolation)
SELECT tfloat '(10@2018-01-01 08:00:00, 20@2018-01-01 08:05:00, 15@2018-01-01 08:10:00]'
-- Linear interpolation – default (do not write Interp = linear, would get an error message)
-- Also, round or squared parenthesis, i.e., a range of values, no discrete interpolation
allowed
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]' -- Stepwise interpolation
```

- A sequence value has a lower and an upper bound, inclusive ('[' and ']') or exclusive ('(' and ')')
- **Must be inclusive** when interpolation is discrete or when the sequence has a single instant (an instantaneous sequence)

`SELECT tfloat 'Interp=Step;(10@2018-01-01 08:00:00]'` – this will return an error message

Temporal data types

- Example of temporal types
- Temporal integer

```
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)')
(40, 'Marketing', tint '{[10@2001-01-01, 12@2001-04-01, 22@2001-08-01)}');
```

- Temporal geometry

```
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)}'),
(30, 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}');
```

```
SELECT  tempSubType(trip),  interp(trip), *
FROM trips;
```

	tempsubtype text	interp text	carid integer	tripid integer	trip tgeompoint
1	SequenceSet	Linear	10	1	{[0101000000000000000000000000000000000000@20
2	SequenceSet	Linear	20	1	{[0101000000000000000000000000000000000000@20
3	Sequence	Discrete	30	1	{0101000000000000000000000000000000000000@20

Temporal type constructors

- Each temporal type has a constructor function with a type name and a suffix for the subtype
    - Suffixes *'_inst', '_seq', '_seqset'* correspond to subtypes *instant, sequence, and sequence set*
    - Eg.: *tint_seq, tgeompoint_seqset*

- *Constructors:*
    - Constructor for temporal types of *instant* subtype
    - Constructor for temporal types of *sequence subtype* with *discrete interpolation*
    - Constructor for temporal types of *sequence* subtype with *step* and *linear* interpolation
    - Constructors for temporal types of *sequence set* subtype

Temporal data types: *instant* subtype constructors

- Constructor for temporal types of *instant* subtype

```
<ttype>(base, timestamptz) ->  ttype_inst
<ttype>(base, tstzset) ->  ttype_inst
```

```
SELECT tbool_inst('true@2001-01-01');

SELECT tint_inst(1, '2001-01-01');

SELECT tfloat_inst(1.5, '2001-01-01');

SELECT tgeogpoint_inst('SRID=7844;Point(1 1)@2001-01-01’);

SELECT tgeompoint_inst('Point(0 0)', timestamptz '[2001-01-01]');
```

Temporal data types: *sequence* subtype

- Constructor for temporal types of *sequence* subtype

```
Step interpolation
SELECT tbool_seq(ARRAY[tbool 'true@2001-01-01 08:00:00','false@2001-01-01 08:05:00']);
SELECT tint_seq(ARRAY[tint '1@2001-01-01 08:00:00', '2@2001-01-01 08:05:00']);
SELECT ttext_seq(ARRAY[ttext 'AAA@2001-01-01 08:00:00', 'BBB@2001-01-01 08:05:00']);
SELECT tfloat_seq(ARRAY[tfloat 'Interp=Step;('1.0@2001-01-01 08:00:00', '2.0@2001-01-01
       08:05:00’], 'step');

Linear interpolation
SELECT tfloat_seq(ARRAY[tfloat '1.0@2001-01-01 08:00:00', '2.0@2001-01-01 08:05:00']);
SELECT tgeompoint_seq(ARRAY[tgeompoint 'Point(0 0)@2001-01-01 08:00:00’, 'Point(0 1)@2001-01-
       01 08:05:00', 'Point(1 1)@2001-01-01 08:10:00']);
SELECT tgeogpoint_seq(ARRAY[tgeogpoint 'Point(1 1)@2001-01-01 08:00:00','Point(2 2)@2001-01-
       01 08:05:00']);
Try:

SELECT tempsubtype(tbool_seq(ARRAY [tbool 'true@2001-01-01 08:00:00','false@2001-01-01
       08:05:00'])), interp(tbool_seq(ARRAY [tbool 'true@2001-01-01
       08:00:00','false@2001-01-01 08:05:00’])); (you should obtain step interpolation)

Do the same with the other types
```

Temporal data types: *sequence set* subtype constructors

- Constructor for temporal types of *sequence set* subtype

```
SELECT tbool_seqset(ARRAY[tbool '[false@2001-01-01 08:00:00, false@2001-01-01
       08:05:00)','[true@2001-01-01 08:05:00]','(false@2001-01-01 08:05:00,
       false@2001-01-01 08:10:00)']); - step interpolation
SELECT tint_seqset(ARRAY[tint '[1@2001-01-01 08:00:00, 2@2001-01-01 08:05:00,
       2@2001-01-01 08:10:00, 2@2001-01-01 08:15:00)']); - step interpolation
SELECT tfloat_seqset(ARRAY[tfloat '[1.0@2001-01-01 08:00:00, 2.0@2001-01-01 08:05:00,
       2.0@2001-01-01 08:10:00]', '[2.0@2001-01-01 08:15:00, 3.0@2001-01-01
       08:20:00)']); - linear interpolation (default)
SELECT tfloat_seqset(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 ttext_seqset(ARRAY[ttext '[AAA@2001-01-01 08:00:00, AAA@2001-01-01 08:05:00)',
       '[BBB@2001-01-01 08:10:00, BBB@2001-01-01 08:15:00)']);
SELECT tgeogpoint_seqset(ARRAY[tgeogpoint
       'Interp=Step;[Point(0 0)@2001-01-01 08:00:00, Point(0 0)@2001-01-01   08:05:00)',
       'Interp=Step;[Point(1 1)@2001-01-01 08:10:00, Point(1 1)@2001-01-01 08:15:00)']);
```

Temporal data types: accessor functions

- Accessor functions (some of them previously used)

- Get the temporal type

```
tempSubtype(ttype): {'Instant','Sequence','SequenceSet'}
SELECT tempSubtype(tint '[1@2001-01-01, 2@2001-01-02, 3@2001-01-03]'); -- Sequence
```

- Get the interpolation

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

- Get the value

```
getValue(ttype_inst): base
SELECT getValue(tint '1@2001-01-01'); -- 1
```

- Get the values

```
getValues(ttype): (base[], floatspan[],geo)
SELECT getValues(tint '[1@2001-01-01, 2@2001-01-03, 8@2001-01-05]');--{[1,3),[8,9)}
SELECT getValues{tint '{1@2001-01-01, 7@2001-01-03, 4@2001-01-06}'};-- {[1,2), [4,5),[7,8)}
SELECT getValues(tfloat '(1@2001-01-01, 1@2001-01-02,  4@2001-01-04)'); --{[1, 4)}
```

Temporal data types: accessor functions

- Accessor functions

- Get the timestamp `getTimestamp(ttype_inst): timestamptz`

```
SELECT getTimestamp(tint '1@2001-01-01');  -- 2001-01-01 00:00:00-03
```

- Get the time `getTime(ttype): periodset`

```
SELECT getTime(tint '[1@2001-01-01,2@2001-01-11,1@2001-01-15)');--{[2001-01-01, 2001-01-15)}
SELECT getTime(tint '{[1@2001-01-01,2@2001-01-11],[1@2001-01-15]}');--{[2001-01-01 00:00:00 -03, 2001-01-11 00:00:00-03], [2001-01-15 00:00:00-03, 2001-01-15 00:00:00-03]}
```

- Get the different timestamps `timestamps(ttype): timestamptz[]`

```
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"}
```

- Get the sequences `sequences({ttype_seq,ttype_seqset}): ttype_seq[]`

```
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)"}
```

- Get the instants

```
SELECT instants(tfloat '{[1@2000-01-01, 2@2000-01-02), (2@2000-01-02, 3@2000-01-03)}');
-- {"1@2000-01-01 00:00:00-03","2@2000-01-02 00:00:00-03","2@2000-01-02 00:00:00-03","3@2000-
   01-03 00:00:00-03"}
```

Temporal data types: restriction functions

- Restriction functions

- Restrict to a set of values `atValues(ttype,values) -> ttype`

```
SELECT atValues(tint '{1@2001-01-01, 2@2001-01-03, 6@2001-01-05}',intset '{1, 6}');        --
-- {1@2001-01-01 00:00:00-03, 6@2001-01-05 00:00:00-03}
```

- Restrict to the minimum value `atMin(torder) ->torder`

```
SELECT atMin(tint '{1@2001-01-01, 2@2001-01-03,1@2001-01-05}');
--{1@2001-01-01, 1@2001-01-05}
```

- Restrict to a geometry `atGeometry(tgeompoint,geometry)->  tgeompoint`

```
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]"}
```

- Restrict to a timestamp `atTime(ttype,time) -> ttype`

```
SELECT atTime(tfloat '[1@2001-01-01, 5@2001-01-05)', timestamptz '2001-01-02');
-- 2@2001-01-02
SELECT atTime(tfloat '[1@2001-01-01, 5@2001-01-05)', tstzspan '[2001-01-02,2001-01-04]’);
-- [2@2001-01-02 00:00:00-03, 4@2001-01-04 00:00:00-03]
```

Temporal data types: comparison operators

- **Traditional comparison operators** (=, <, …)
- Left and right operands must be of the same base type.
- Operators compare (in this order) the **bounding periods**, then the **bounding boxes**, 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.

```
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
```

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

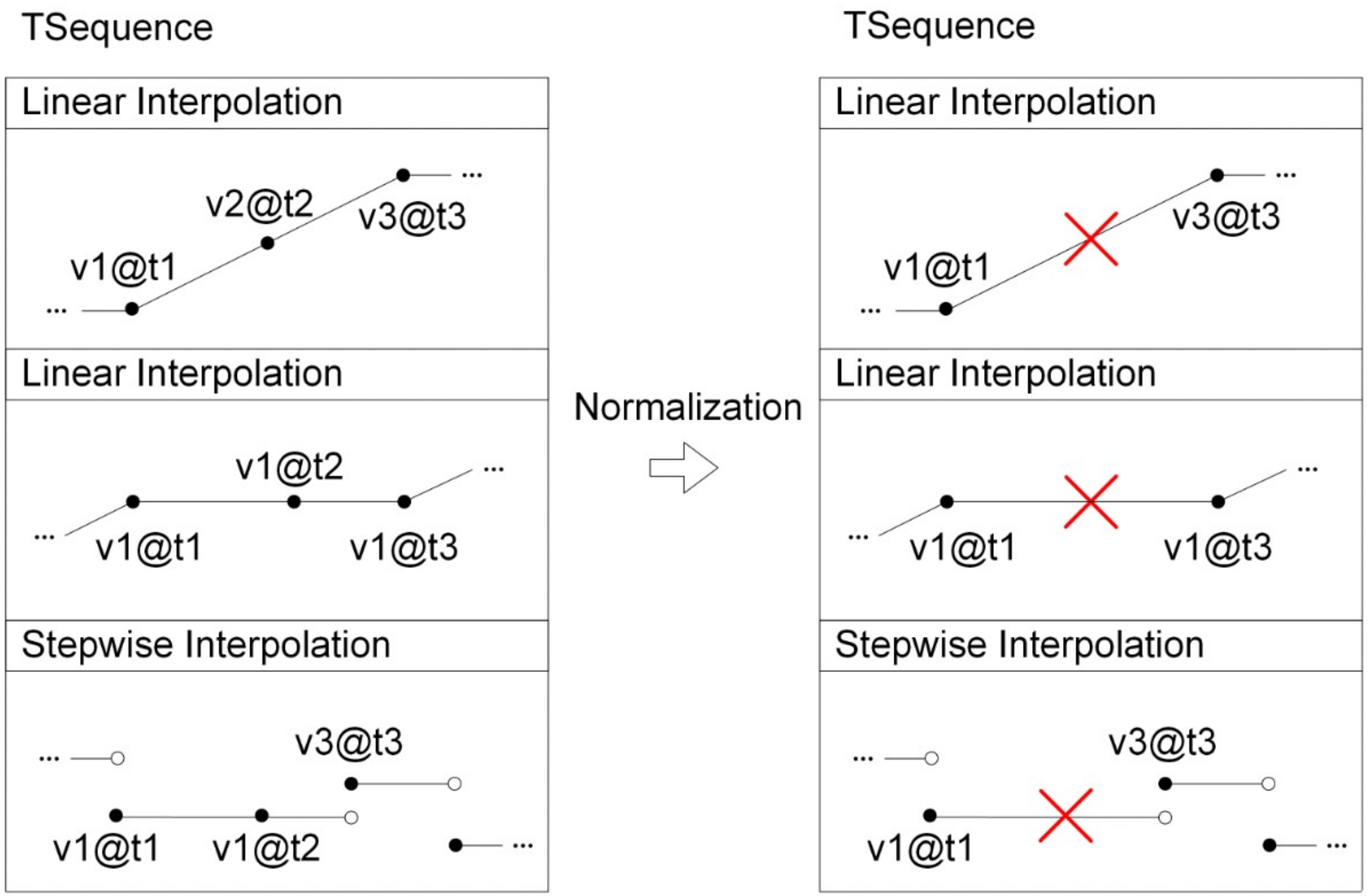
```
ttype < ttype: boolean
SELECT tint '[1@2001-01-03, 2@2001-01-06]' < tint '[1@2001-01-03, 2@2001-01-05]'--false
SELECT tint '[1@2001-01-03, 2@2001-01-05, 3@2001-01-08]' < tint '[1@2001-01-03, 3@2001-01-
09]'--true
SELECT tint '[1@2001-01-03, 4@2001-01-05]' < tint '[1@2001-01-03, 3@2001-01-05]’--false
```

Temporal data types - Normalization

- *Sequence* or *sequence set* values that are continuous **(that is, when the interpolation is linear or stepwise),** are normalized.
- **Consecutive instant values are merged when possible.**
- Given three consecutive instant values, the middle value can be deleted if the linear functions defining the evolution of values are the same (see next slides)
- The normalization process is performed by the constructors of the *TSequence* and *TSequenceSet* data types.
- This occurs both at the time mobility data is input and when computing the result of any operation.
- Normalization thus performs lossless compression that can achieve up to 400% compression rate when real-world mobility data is input.

Normalization: *sequence* types

- Sequence values that are continuous (i.e., interpolation is linear or stepwise), are normalized
- Consecutive instant values are merged whenever possible
- **Given three consecutive instant values, the middle value can be deleted**

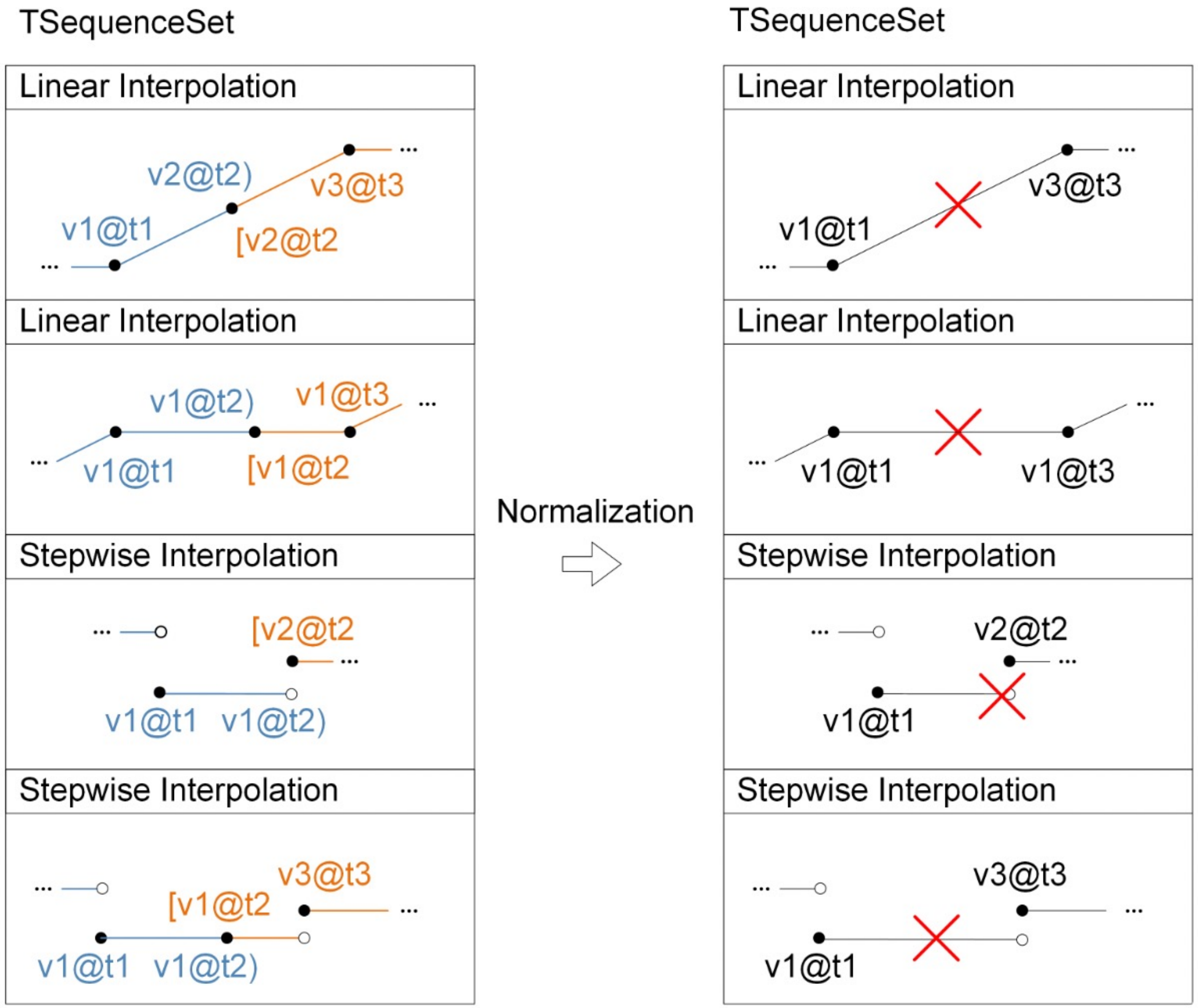


Normalization - Examples

```
SELECT tint '[1@2001-01-01, 2@2001-01-03, 2@2001-01-04, 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, 3@2001-01-05]'

--> [1@2001-01-01, 3@2001-01-05]

SELECT tfloat '[1@2001-01-01, 2@2001-01-02, 3@2001-01-03, 4@2001-01-04, 5@2001-01-05]'

--> [1@2001-01-01 00:00:00-03, 5@2001-01-05 00:00:00-03]

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 08:00:00-03, POINT(1 1)@2001-01-01 08:10:00-03)
```

Normalization : *sequence set* types

- In the case of *sequence set* values, two composing sequences are merged whenever possible.
- This happens when the instant to be removed connects two consecutive sequences
- In the example, the first sequence is right exclusive and the second one is left inclusive.



Normalization - Examples

```
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]}
```

Deleted during normalization

```
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 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)}
```

Temporal data types - Lifting

- Lifting: extends a static (non-temporal) operator/function with temporal capabilities
- Examples:
  - Arithmetic operators (+, -, *, /): require two numbers (integer or float), return a number
    - Lifting: one or both arguments are temporal numbers and the is also a temporal number
  - Distance between two points (st_distance in PostGIS): requires either two geometries or two geographies, returns a float
    - Lifting: one or both arguments are temporal points and the result is a temporal float

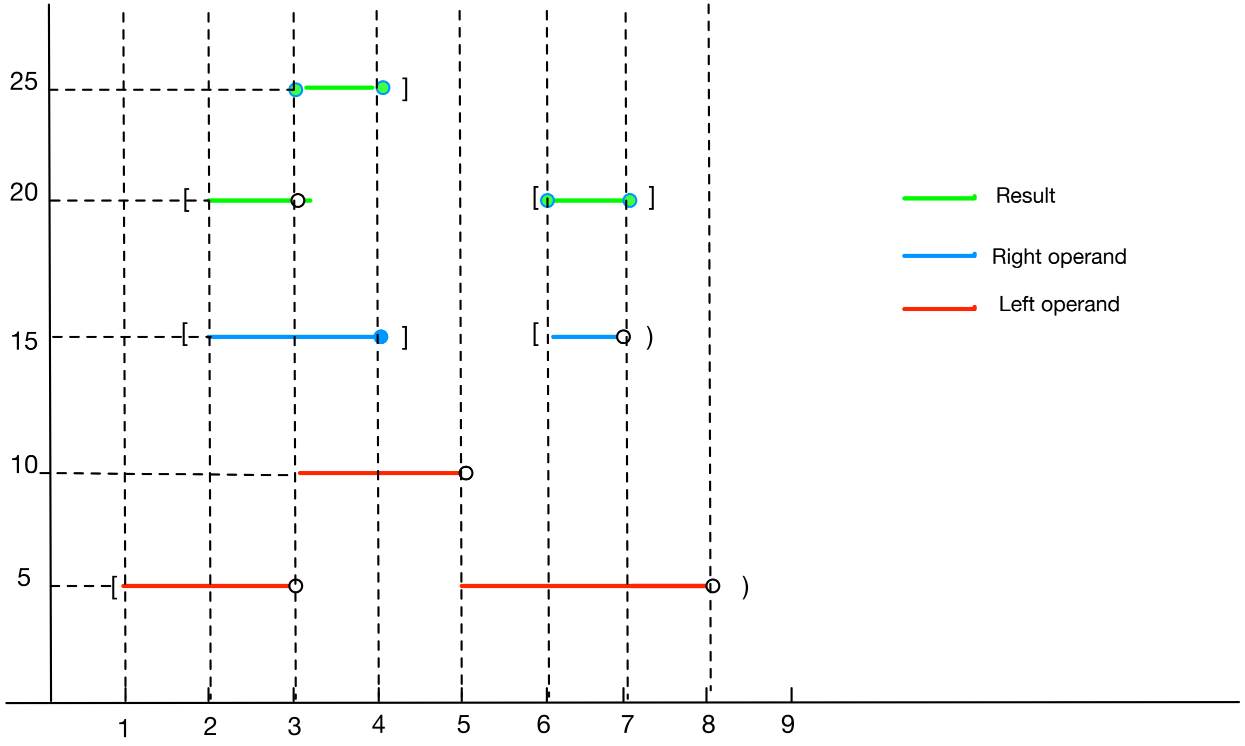
$$
\begin{array}{llllll}
\multicolumn{6}{l}{\underline{\texttt{LIFTED}(op)}} \\
 & \mathrm{TEMPORAL}(\alpha) & \times & \mathrm{TEMPORAL}(\beta) & \rightarrow & \mathrm{TEMPORAL}(\gamma) \\
 & \alpha & \times & \mathrm{TEMPORAL}(\beta) & \rightarrow & \mathrm{TEMPORAL}(\gamma) \\
 & \mathrm{TEMPORAL}(\alpha) & \times & \beta & \rightarrow & \mathrm{TEMPORAL}(\gamma)
\end{array}
$$

- To implement lifting in the sequence model, we first need to introduce the notions of **synchronization** and **turning point** (normalization is also used)

Synchronization

```
SELECT tint '{[5@2007-05-01, 10@2007-05-03, 5@2007-05-05, 5@2007-05-08)}' +
tint '{[15@2007-05-02,15@2007-05-04], [15@2007-05-06,15@2007-05-07)}'
```

→{[20@2007-05-02 , 25@2007-05-03 , 25@2007-05-04], [20@2007-05-06, 20@2007-05-07)}



Note that the result of the operation is only defined over the time intervals **where all the arguments** are defined.

Lifting algorithm – turning points

- Temporal multiplication between two operands is more involved .
- The result of the product of two tfloat values (linear interpolation), is quadratic
- The result must **also** be approximated by a linear function
- **The approximation keeps the local maxima and minima of the quadratic result (the turning points)**
- **The first step synchronizes the two operands**

Lifting algorithm – turning points

- Consider the query:

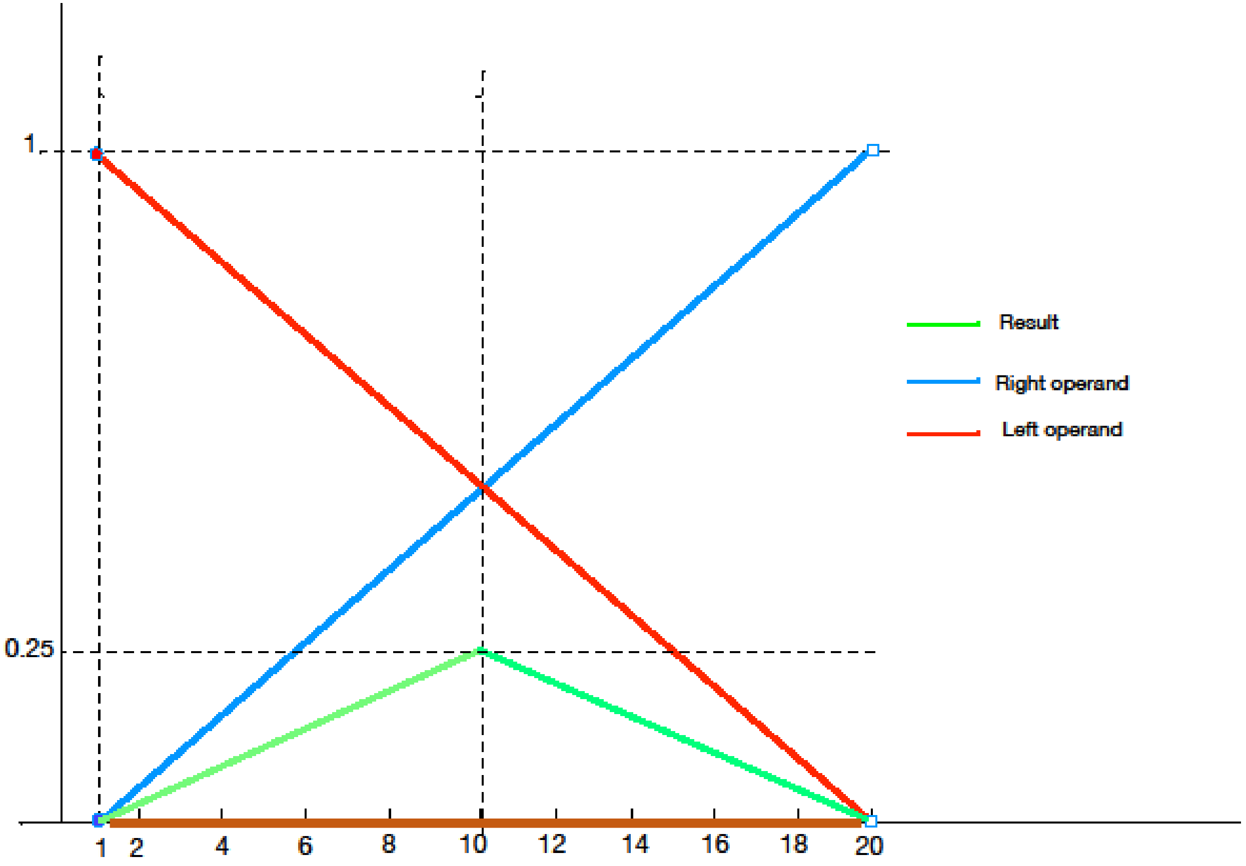
```
WITH tt as (SELECT tfloat '{[1@2007-05-01, 0@2007-05-20)}' * tfloat '{[0@2007-05-01,1@2007-05-20)}' as tinteg)
```

- The quadratic function has a maximum at day 10. Thus, we need a new interval at this point. The green line is the approximation of the product. Otherwise, we would obtain the brown line.

The values returned at days 10 and 15 are computed as:

```
SELECT atTime(tinteg,timestamptz '2007-05-10 12:00:00') from tt
-- 0.25@2007-05-10 12:00:00-03

SELECT atTime(tinteg,timestamptz '2007-05-15 00:00:00') from tt
-- 0.1315@2007-05-15 00:00:00-03
```

(Compare against the actual product)



Lifting algorithm – turning points

- We see that between 6 and 7 we have a quadratic function.
- However, no turning point appears, since intervals are small
- Below, in **bold** the synchronization instants

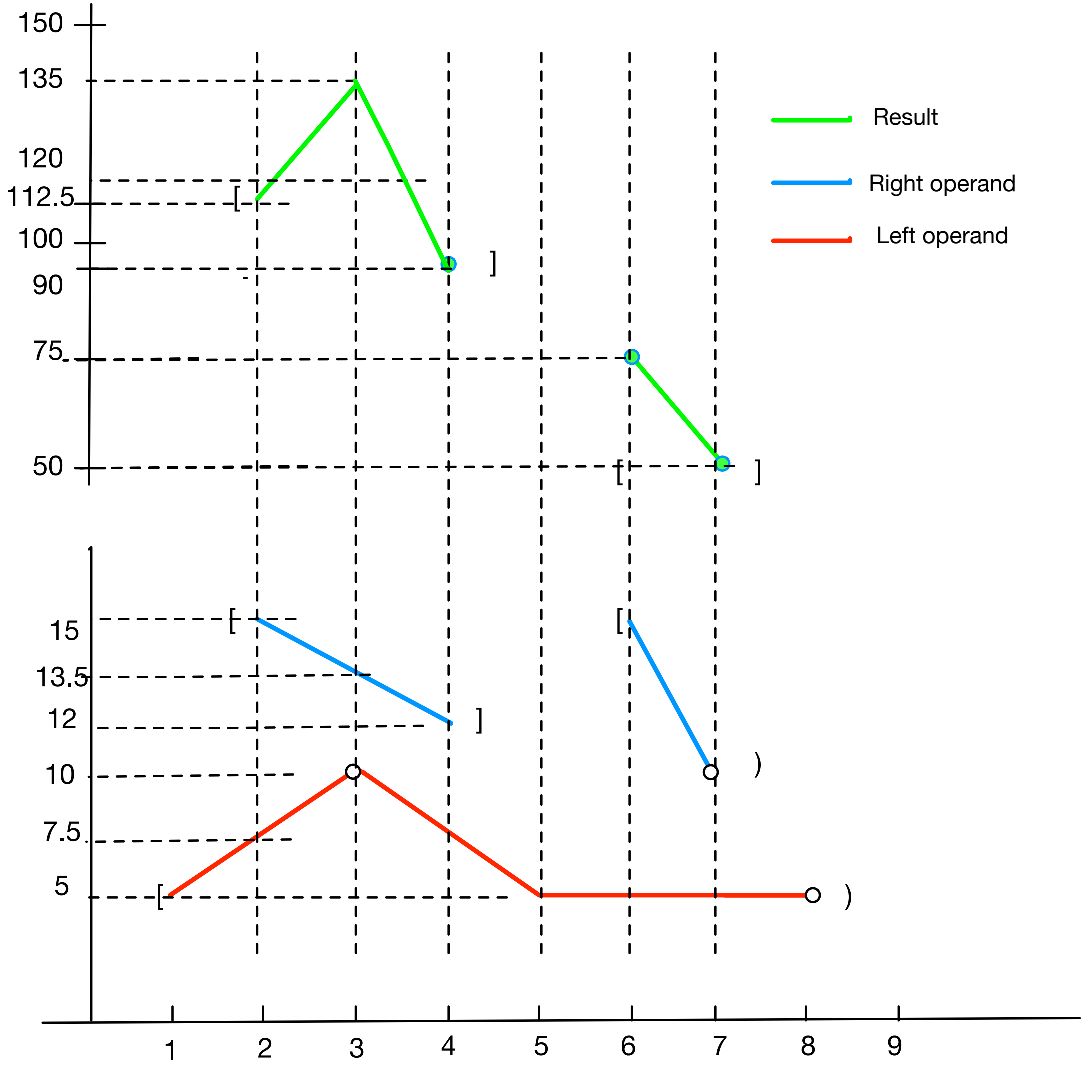
```
WITH tt as (SELECT tfloat '{[5@2007-05-01,
10@2007-05-03, 5@2007-05-05, 5@2007-05-08)}' *
tfloat '{[15@2007-05-02,12@2007-05-04],
[15@2007-05-06,10@2007-05-07)}' as tinteg)
```

--> {[112.5@2007-05-**02** 00:00:00-03, 135@2007-05-**03** 00:00:00-03, 90@2007-05-0**4** 00:00:00-03], [75@2007-05-0**6** 00:00:00-03, 50@2007-05-0**7** 00:00:00-03)}

Note the returned value for the product at midday on day 2

```
SELECT atTime(tinteg,timestamptz '2007-05-02
12:00:00-3') from tt
```

→123.75@2007-05-02 12:00:00-03
→This is a linear approximation, not the real value of the product.



Lifted functions and operations

- Get the length traversed by the temporal point `length(tpoint): float`

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

- Get the **cumulative** length traversed by the temporal point `cumulativeLength(tpoint): tfloat_seq`

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

- Get the smallest distance ever `{geo,tpoint} |=| {geo,tpoint}: 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
```

- Get the temporal distance `{point,tpoint} <-> {point,tpoint}: 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)
```

Ever spatial relationships

- Ever within a distance

```
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);-- true
```

- Ever contains

```
econtains({geo,tgeompoint},{geo,tgeompoint}): Boolean
SELECT econtains(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)'); -- true
```

- Ever disjoint

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

- Ever intersects

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

Lifted predicates

- Temporal contains

```
tcontains(geometry, tgeompoint): tbool

SELECT tcontains(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)');
-- {[f@2001-01-01, f@2001-01-02], (t@2001-01-02, f@2001-01-03, f@2001-01-04)}
```

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

```
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 00:00:00-03, f@2001-01-03 00:00:00-03], (t@2001-01-03 00:00:00-03,
t@2001-01-05 00:00:00-03)}

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]}
```

Lifted predicates

- Temporal distance within - The function only allows 3D for two temporal points

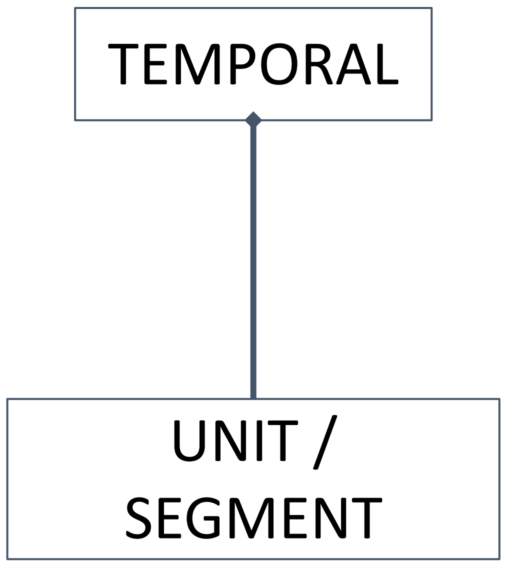
```
tdwithin({geompoint,tgeompoint},{geompoint,tgeompoint},float)-> tbool

SELECT tdwithin(tgeompoint '[Point(1 0)@2000-01-01, Point(1 4)@2000-01-05]',
tgeompoint 'Interp=Step;[Point(1 2)@2000-01-01, Point(1 3)@2000-01-05]', 1);

-- {[f@2000-01-01, t@2000-01-02, t@2000-01-04], (f@2000-01-04, t@2000-01-05]}
```

- Temporal intersects

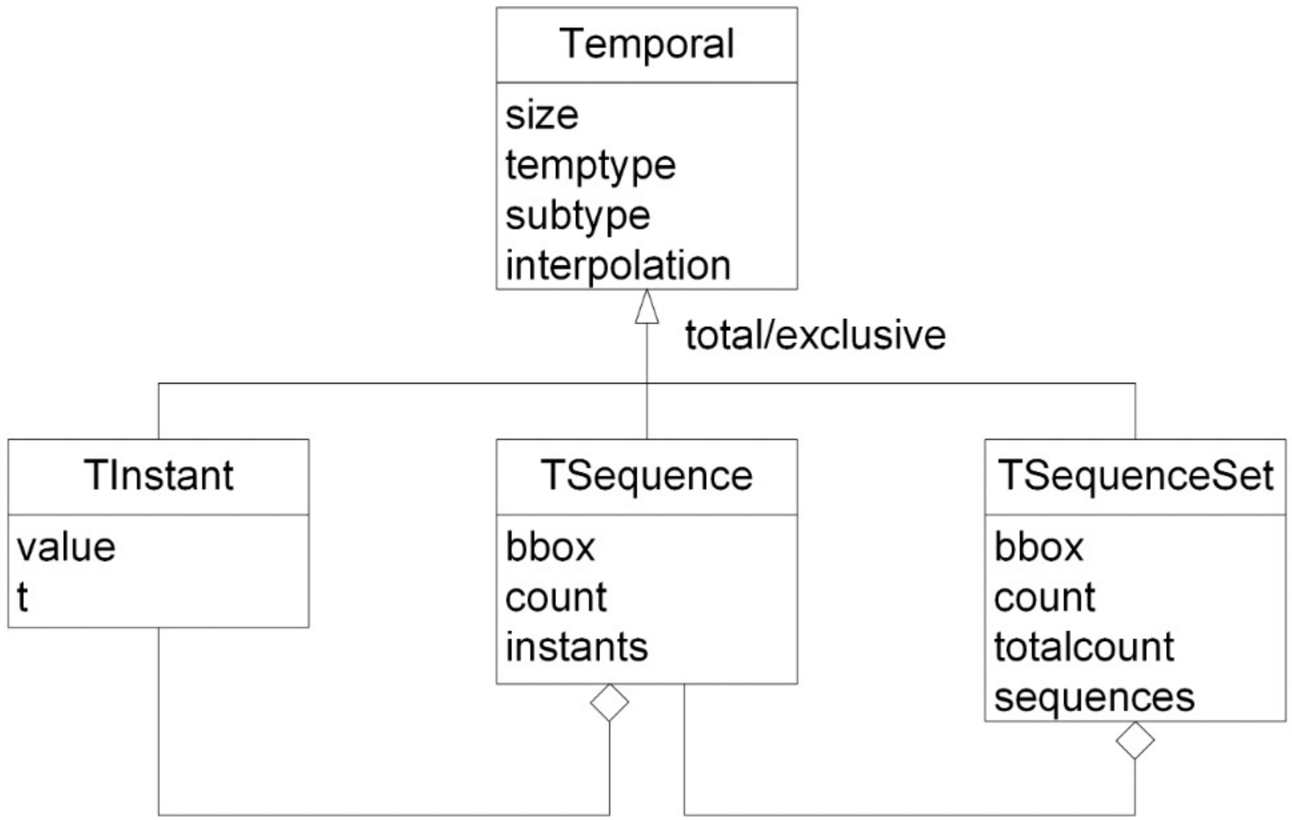
```
tintersects({geo,tpoint},{geo,tpoint})-> tbool

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]} */
```

Discussion: sliced vs sequence representations

Comparing the two models: conceptual level



Sliced
representation

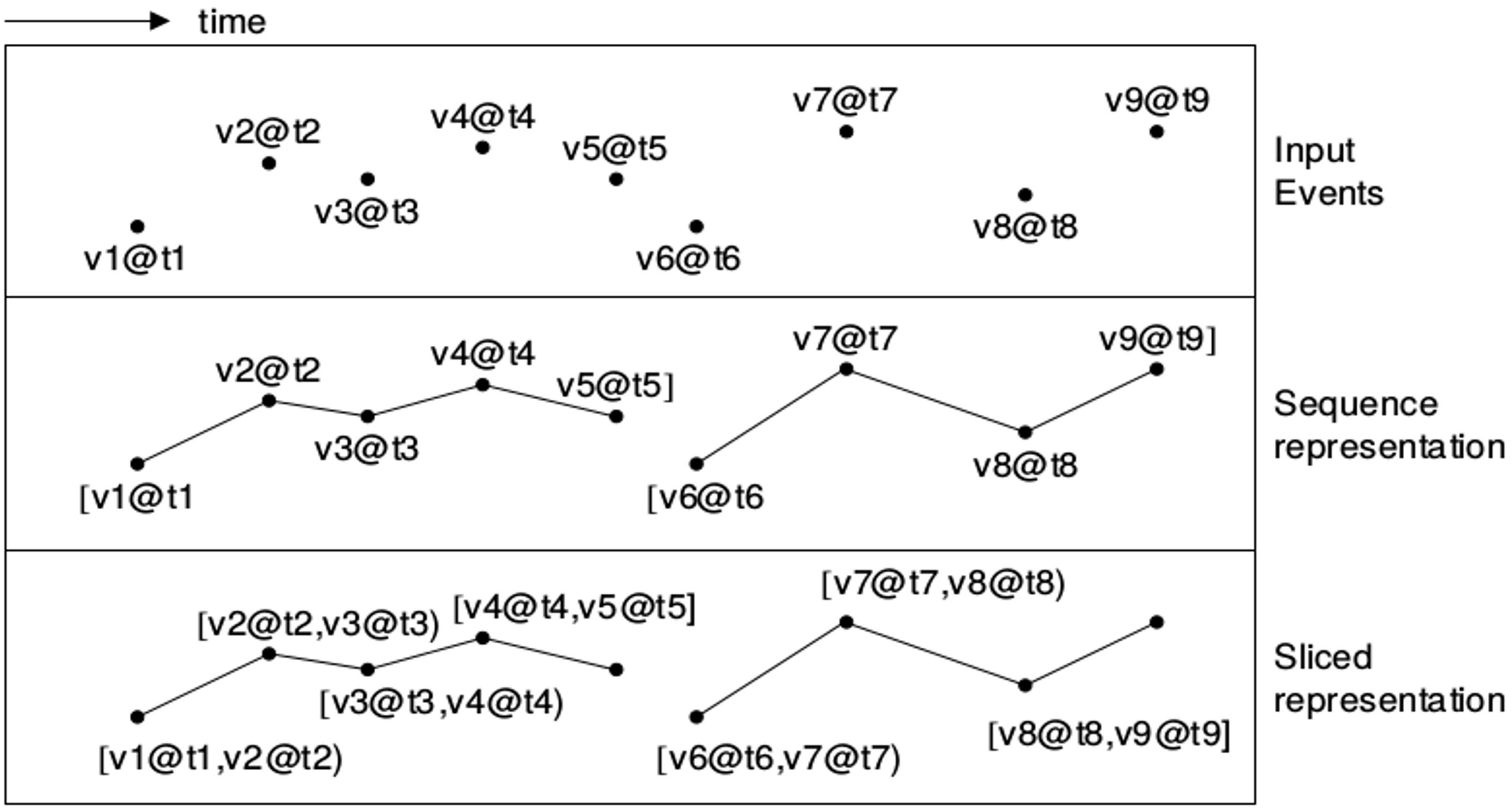


representation

Comparing the two representations

- **Sliced representation:** encodes the evolution of values of a temporal type in time periods, called units.
    - The UNIT type constructor is a pair (*time interval*, *function*)
    - *function* describes the evolution of the object during *time interval*
- The temporal functions of the units are independent = > we can represent different changes in the object value at the boundary between two units
- We can introduce temporal gaps between units
- The sliced representation can uniformly represent all these scenarios using the two type constructors: UNIT and MAPPING

- **The sequence representation:** achieves the same expressiveness by defining more type constructors to address the different evolution scenarios: INSTANT, SEQUENCE, and SEQUENCE SET.



	Sliced representation	Sequence representation
tInstant	[v1@t1, v1@t1]	v1@t1
tSequence	[v1@t1, v2@t2), [v2@t2, v3@t3), ...	[v1@t1, v2@t2, …]
tSequence set	[[v1@t2, v2@t2), ... [v4@t4, v5@t5], [v6@t6),	{[v1@t1, …, v5@t5],…,[v6@t6,..,v9@t9]}

Comparing the two representations

- Top figure: input events, coming from sensors. There is a gap at t5
- The sliced representation creates a list of units, each representing a segment between two consecutive events. It thus duplicates the intermediate events, e.g., (v2, t2) is stored once as the right bound of the first unit and once more as the left bound of the second unit
- The sequence representation creates two sequences out of these input events: during [t1, t5] and [t6, t9]
- Sequence representation: a storage reduction of 1/2 the storage required for the sliced representation (does not duplicate the intermediate events stores a single timestamp instead of a time interval per segment)
- Algorithms can be optimized, because the types encode more information (the temporal continuity information is encoded in the representation)