Info-H-415 - Advanced Databases

Mobility (Moving Objects) Databases

Lesson 2: The abstract Model

Alejandro Vaisman
avaisman@itba.edu.ar

Abstract vs. Discrete models

Two options:

- Abstract Model: Definitions in terms of infinite sets allowed. We don't care about finite representation.
- Discrete Model: Only definitions in terms of finite representations are allowed.

Abstract vs. Discrete models

- Two options:
- Abstract Model: Definitions in terms of infinite sets. Don't care about finite representation.
- Discrete Model: Only definitions in terms of finite representations are allowed.

- In both cases:

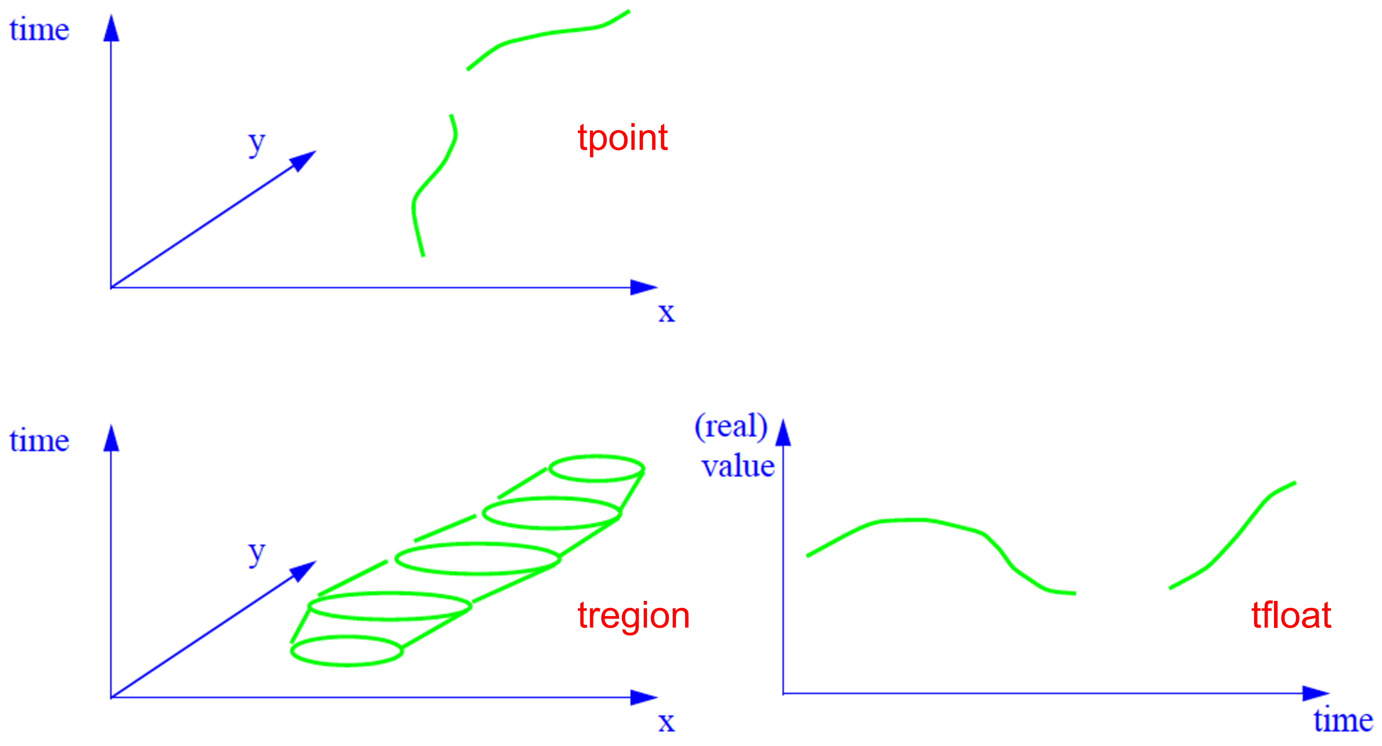
  1. Define a signature (data types + operations)

  2. Define a semantics for the above

Abstract vs. Discrete models

Abstract – Continuous - Infinite	Discrete – Concrete – Finite
A region is a closed subset of $\mathbb{R}^2$ with non-empty interior	A region is a set of polygons which may have polygonal holes
A line is a curve in $\mathbb{R}^2$	A line is a polyline (a list of line segments)
A temporal (point) is a function from time into Point values	A temporal(point) is a polyline in a 3D space
A temporal (float) is a function from time into Real values	A temporal(float) is a piecewise quadratic function
A temporal(region) is a function from time into Region values	A temporal(region) is a polyhedron in 3D

Temporal points, lines, regions, reals (float)



Abstract Data Types for Moving Objects

- Basic abstractions in spatial databases: point, line, region.
    - Point: An entity for which only the position in space is relevant.
    - Line: Usually ways for moving, connections in space.
    - Region: An entity for which also the extent is relevant.

- We focus on *moving points* and *moving regions. "Temporal" and "moving" used indistinctly*
- Temporal (moving) points: people, animals, stars, cars, planes, ships, …
- Temporal (moving) regions: forests, forest fires, oil spills, countries, hurricanes, bird flock, fish colony, …
- Support all kinds of queries about such moving objects.

Abstract vs. discrete models

- Abstract models are
  - + simple
  - + generic (admit several discrete implementations)
  - – not (directly) implementable

- Discrete models
  - are more complex
  - represent specific choices not suitable for all applications
  - offer a direct implementation

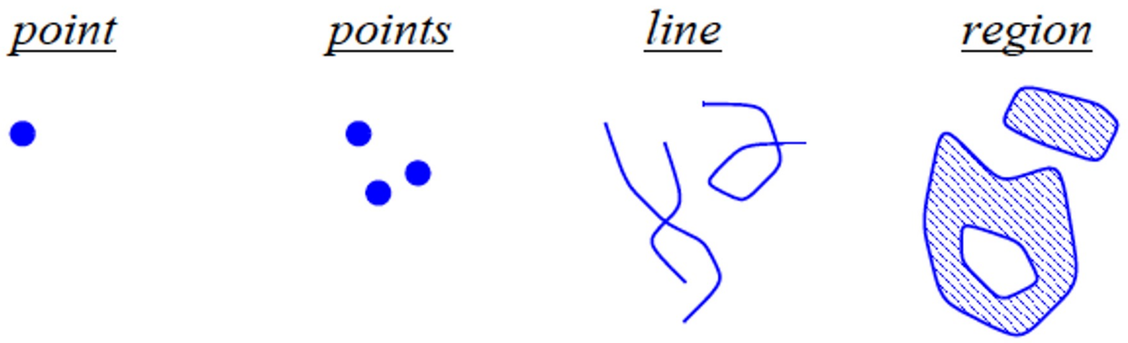
- Conclusion: Proceed in two steps

  1. Design abstract model as a target.

  2. Design and implement a discrete models as an instantiations of the abstract model

Abstract model: Base (non-temporal) data types

- Non-spatial types : as usual, int, float, bool, etc.
- Spatial types



Making base types temporal

- base type + time

tpoint = time x point

tregion = time x region

tfloat = time x float

Continuous changes

tint = time x int

tbool = time x bool

ttext = time x text

Discrete changes

Temporal data types: constructors

Two data types to represent time: **instants and periods**

- Type *instant*: isomorphic to *real*
- Type *periods*: set of disjoint time intervals)

Two type constructors for temporal data types: **temporal and intime**

- Type constructor *temporal* yields for each base and spatial type, a **temporal type**
- Type constructor *intime* yields for each base or spatial type, a type whose values are pairs e.g., *intime(int)* yields a pair *(instant, int)*

Temporal data types: constructors

When α is an ordered domain (*int*, *real*, *bool*, *string*, *instant*)

- *range*(α) Yields a finite set of disjoint, non-adjacent intervals (closed, open, half-open) over α (e.g., periods = range(instant)).

When α is a standard or spatial type (*int*, *real*, *bool*, *string*, *point*, *points*, *line*, *region*)

- *temporal*(α) Values are feined by a partial function *f*: $A_{instant} \to A_\alpha$ *where* A_α *is the carrier set of* α.

- *intime*(α) Produces a value that is a pair in $A_{instant} \times A_\alpha$.
  (In MobilityDB implemented by ***instants()****)*

Operations

- Four steps:

  1. Define operations on non-temporal types
  2. Define (unary) operations for temporal types
  3. **"Lift"** operations on non-temporal types to temporal types: extend these operations to the temporal version of non-temporal types (e.g., distance computation)
  4. Include operations from various domains (e.g., topological operations)

Topological relationships and predicates (reminder)

- Topological Relationships
  - Invariant under elastic deformation (without tear, merge)
  - Two countries which touch each other in a planar paper map will continue to do so in spherical globe maps
- Example queries with topological operations
  - What is the topological relationship between two objects A and B ?
  - Find all objects which have a given topological relationship to object A ?
- Expressed by spatial predicates
- A spatial (non-temporal) predicate is a function with signature

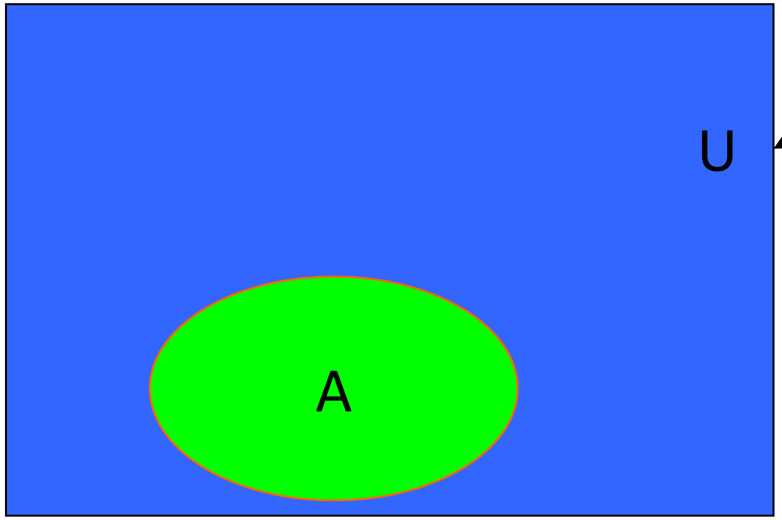
  T1 x ….x Tn. - > bool

  where *Ti* is of base type int, text, point, etc. For example

  E.g.: Grand Place ***within*** Bruxelles

Topological relationships

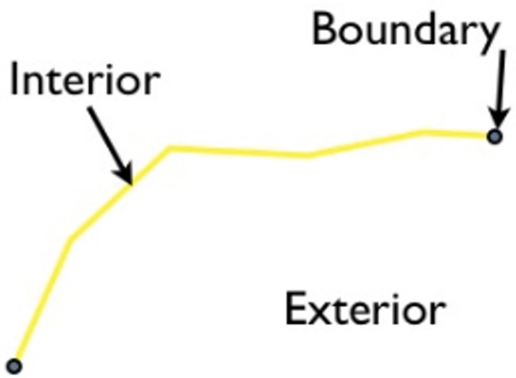
- Interior, boundary, exterior
    - Let A be an object in a "Universe" U



Green is A interior (A^o)

Red is boundary of A (∂A)

Blue – (Green + Red) is A exterior (A^-)



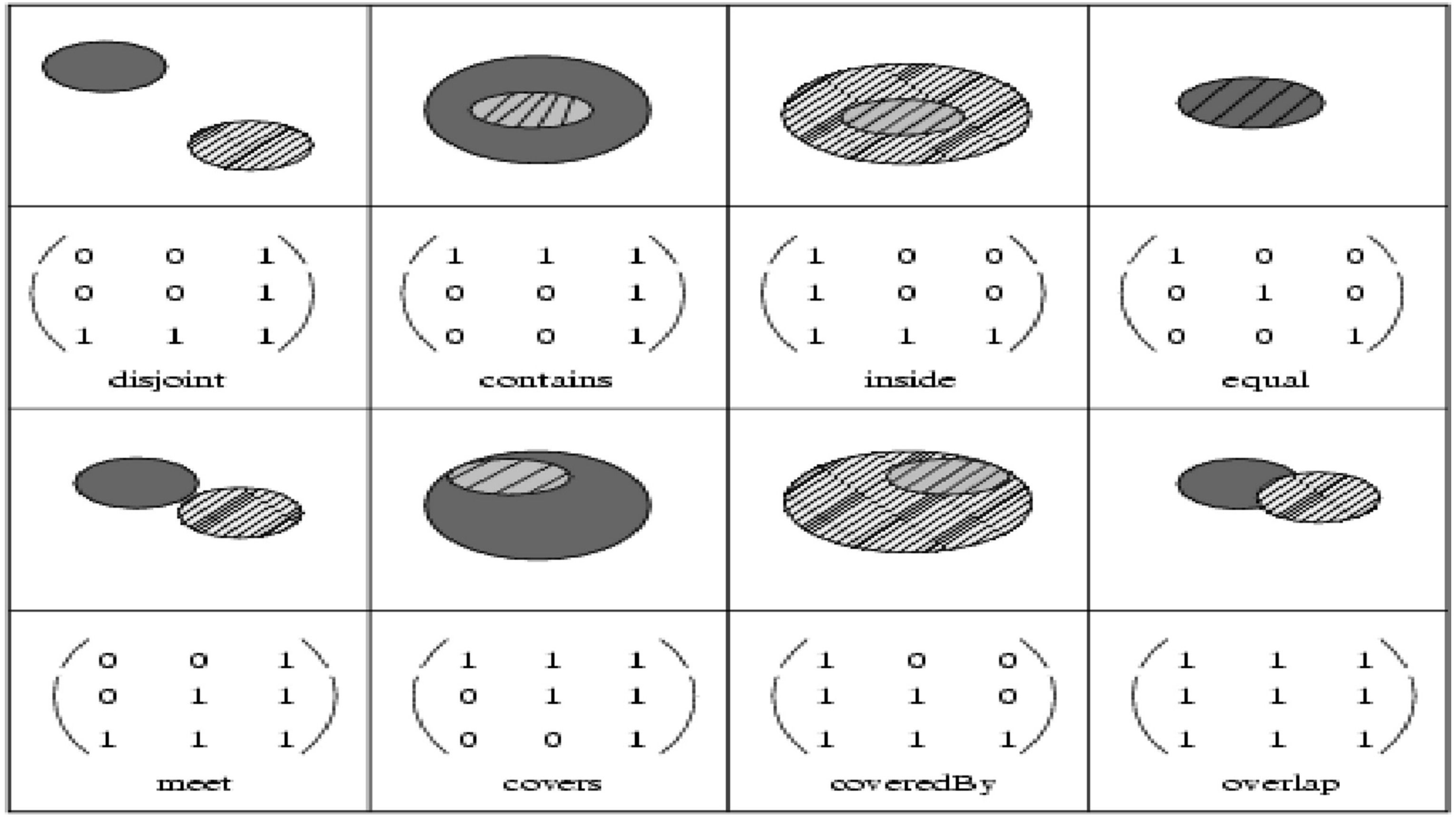
The 9-intersection model (Egenhofer, 1993)

- Topological relationships between A and B can be specified using the 9-intersection model
- Nine-intersection matrix
  - Intersections between interior, boundary, exterior of A, B
  - Can be arranged as a 3 x 3 Boolean matrix
  - Every different set of 9-intersections describes a different topological relation. Relations with the same specification are topologically equivalent
- For this matrix 2^9 = 512 different configurations are possible, **but only a subset makes sense**
- For **two simple regions**, **eight meaningful configurations** have been identified which leads to eight predicates: *equal, disjoint, coveredBy, covers, overlap, meet, inside, and contains*

$$\begin{bmatrix} \partial A \cap \partial B \neq \varnothing & \partial A \cap B^{\circ} \neq \varnothing & \partial A \cap B^{-} \neq \varnothing \\ A^{\circ} \cap \partial B \neq \varnothing & A^{\circ} \cap B^{\circ} \neq \varnothing & A^{\circ} \cap B^{-} \neq \varnothing \\ A^{-} \cap \partial B \neq \varnothing & A^{-} \cap B^{\circ} \neq \varnothing & A^{-} \cap B^{-} \neq \varnothing \end{bmatrix}$$

Topological configurations of the 9-intersection matrix

The eight meaningful configurations for two simple regions



Operations on non–temporal types

- π: variables ranging over types whose values are single elements
- σ: variables ranging over types whose values are subsets of the space (e.g., ranges, points)

- For example, the *within* operation

  within: π x σ → *bool*

- This is an abbreviation for:

  ***within*:** *int* x *range(int)* → *bool*
  bool x *range(bool)* → *bool*
  text x *range(string)* → *bool*
  float x *range(float)* → *bool*
  instant x periods → *bool*
  point x points → *bool*
  point x line → *bool*
  point x region → *bool*

Operations on non–temporal types

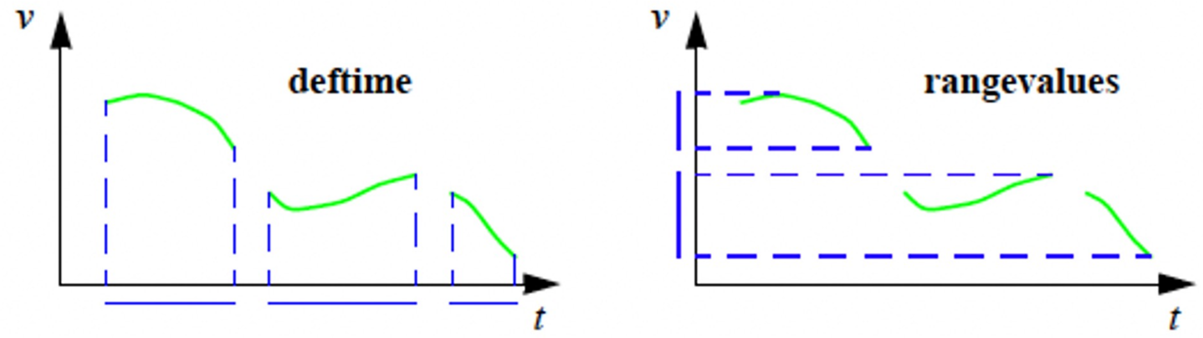
Class	Operations
Predicates	**isempty** =, /=, **intersects**, **within** <, <=, >=, >, **before** **touches**, **attached**, **overlaps**, **on_border**, **in_interior**
Set Operations	**intersection**, **union**, **minus** **crossings**, **touch_points**, **common_border**
Aggregation	**min**, **max**, **avg**, **center**, **single**
Numeric	**no_components**, **size**, **perimeter**, **duration**, **length**, **area**
Distance and Direction	**distance**, **direction**
Base Type Specific	**and**, **or**, **not**

Unary operations on temporal types

Projection to domain or range

deftime **(getTime):** returns the set of time intervals when a temporal function is defined (projection into domain)
rangevalues: performs the projection into the range for the one-dimensional types

For a *temporal(real):*

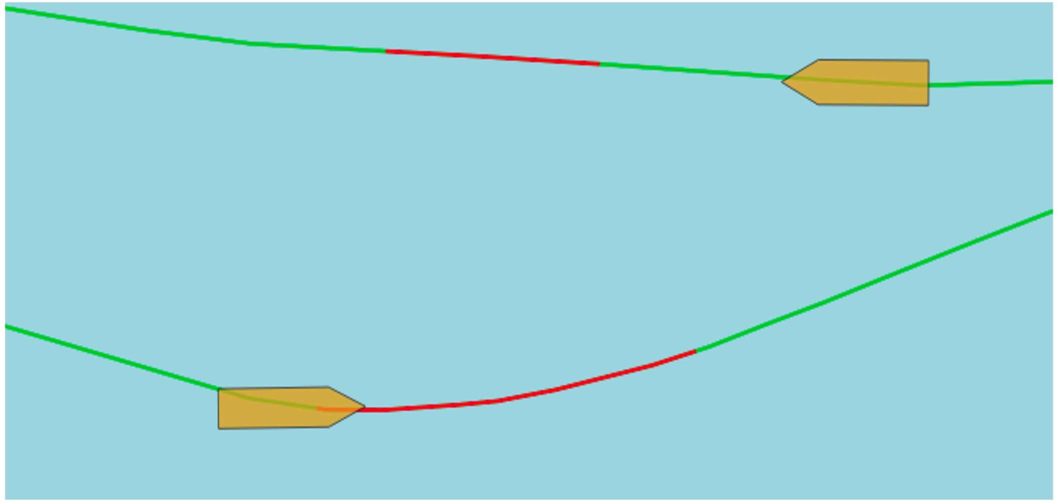


locations: projection of a *temporal(point)* into the plane as points
trajectory (trajectory): projection of a *temporal(point) into* the plane as lines
inst and val: access the components of **intime** types

Unary operations on temporal types

Projection to domain or range

locations: projection of a *temporal(point)* into the plane as points
trajectory (trajectory): projection of a *temporal(point) into* the plane as lines
Traversed: projection of a temporal(line/region) into the plane as regions
inst and val: allow accessing the components of **intime** types



Trajectory

Traversed

Unary operations on temporal types

Projection to domain or range

Operation	Signature	
Deftime (getTime)	*temporal(∝)*	→ *periods*
rangevalues	*temporal(∝)*	→ *range(∝)*
locations	*temporal(point)* *temporal(points)*	→ *points* → *points*
trajectory	*Temporal(point)* *temporal(points)*	→ *line* → *line*
traversed	*temporal(line)* *temporal(region)*	→ *region* → *region*
Inst (getTimestamp)	*intime(∝)*	→ *instant*
Val (getValue)	*intime(∝)*	→ *∝*

Unary operations on temporal types

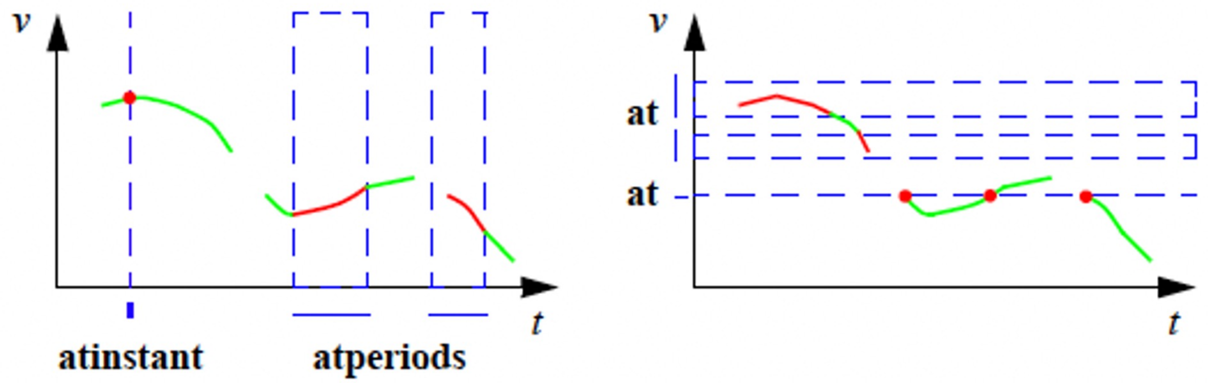
Interaction with points and point sets in domain or range

atinstant and **atperiods (atTime):** restrict the function to a given instant of time or set of time intervals, respectively

at (or atValues): restricts to a point or point set in the range of the function.

- If a *temporal(point)* is restricted by a region, the result is a *temporal(point)*.
- If a *temporal(region)* is restricted by a point value, the result is a *temporal(point)*

For a *temporal(real):*



Unary operations on temporal types

Interaction with points and point sets in domain or range

atmin and atmax restrict the function to the minimum and maximum values of the base types

initial and final (startinstant, endinstant) return the **(instant, value)** pairs for the first and last instant of the definition time

startInstant returns a (value, inst) pair, i.e., an **intime** value.

present: checks whether the temporal function is/has been ever defined at a time instant or in a given set of time intervals

Unary operations on temporal types

Interaction with points and point sets in domain or range

Operation	Signature	
atInstant (atTime)	*temporal(∝) x instant*	→ *intime(∝)*
atPeriods (atTime)	*temporal(∝) x periods*	→ *temporal(∝)*
initial, final (startInstant, endInstant)	*temporal(∝)*	→ *intime(∝)*
present	*temporal(∝) x instant* *temporal(∝) x periods*	→ *bool* → *bool*
at (atValues)	*temporal(∝) x β* *temporal(∝) x β*	→ *temporal(∝)* → *temporal(min((∝, β))*
atmin, atmax	*temporal(∝)*	→ *temporal(∝)*
passes	*temporal(∝)*	→ *bool*

Operations over time and interval types

Time constants constructors

Operation	*Signature*	
year	$\underline{int}$	$\rightarrow \underline{periods}$
month	$\underline{int} \times \underline{int}$	$\rightarrow \underline{periods}$
day	$\underline{int} \times \underline{int} \times \underline{int}$	$\rightarrow \underline{periods}$
hour	$\underline{int} \times \underline{int} \times \underline{int} \times \underline{int}$	$\rightarrow \underline{periods}$
minute	$\underline{int} \times \underline{int} \times \underline{int} \times \underline{int} \times \underline{int}$	$\rightarrow \underline{periods}$
second	$\underline{int} \times \underline{int} \times \underline{int} \times \underline{int} \times \underline{int} \times \underline{int}$	$\rightarrow \underline{periods}$
period	$\underline{periods} \times \underline{periods}$	$\rightarrow \underline{periods}$

Ex.: year denotes the interval comprising the time between the first and last instant of a year

The basic constructors are already given in any programming language
For example in Postgres we have SELECT extract(hour from timestamp '2002-09-17 19:27:45');

period(date1, date2) constructs the time interval ranging from the first instant of date1 to the last instant of date2

Operations over time and interval types

Interval constructors

Operation	Signature	
range	$\propto$ x $\propto$	→ *range($\propto$)*
open, closed, leftclosed, rightclosed	*range($\propto$)*	→ *range($\propto$)*
minint, maxint		→ *int*
minfloat, maxfloat		→ *float*
mininstant, maxinstant		→ *instant*

range: builds the closed interval of the given argument values.
open, closed, leftclosed, and rightclosed: construct open or half-open intervals given a range provided

Lifted binary operations

- Consider an operation **op** with signature

$$\alpha 1 \times \alpha 2 \times \ldots \times \alpha n \rightarrow \beta$$

- Lifting allows any of the argument types to be replaced by the respective temporal type and *return a corresponding temporal type*. **The lifted version of op** has signatures:

$$\alpha 1' \times \alpha 2' \times \ldots \times \alpha n' \rightarrow \beta. \text{ Where } \alpha i' \in \{\alpha i, moving(\alpha i)\}.$$

- Eg.: The **equality** operation on real numbers with signature:

    = float x float → bool

  has lifted versions

    = *temporal(float) x float → temporal(bool)*
    = *float x temporal(float) → temporal(bool)*
    = *temporal(float) x temporal(float) → temporal(bool)*

Lifted binary operations

- The **intersection** operation with signature

  intersection *point x region → point*
  has **lifted** versions

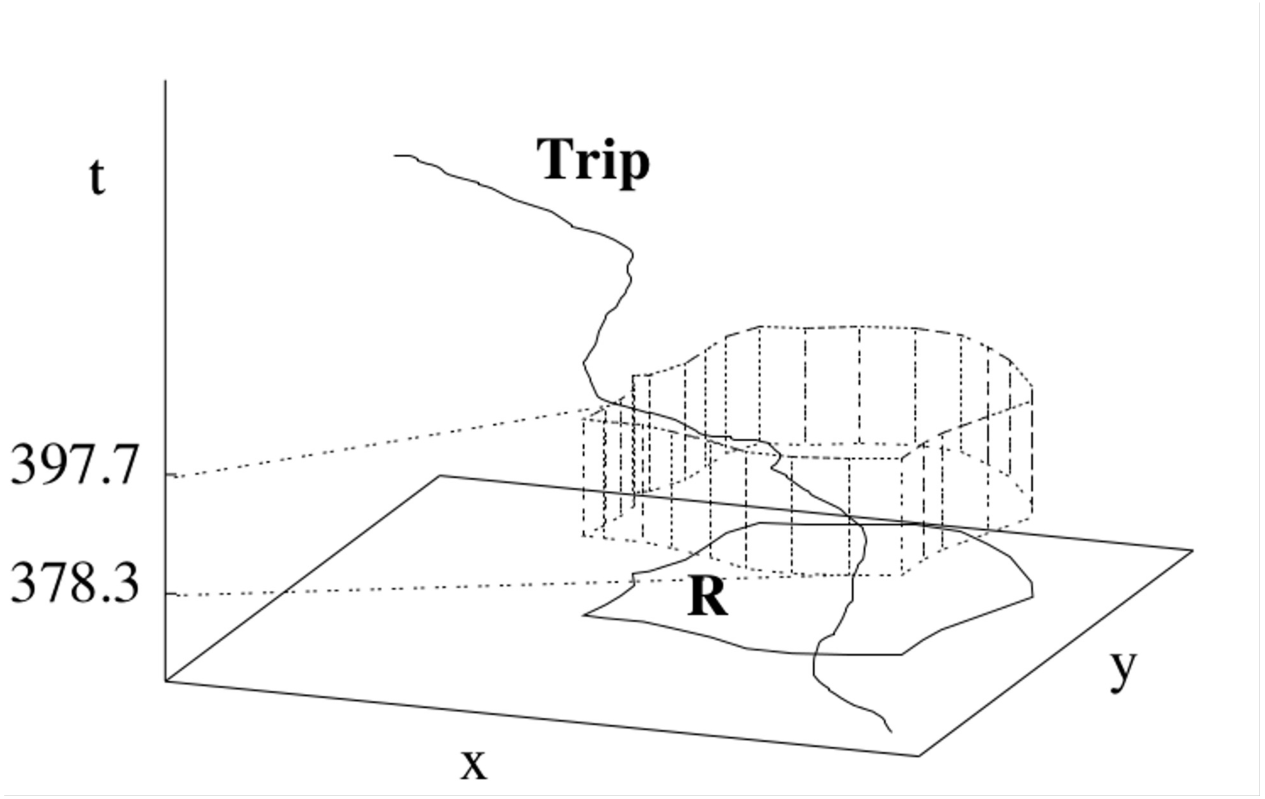
tintersection *temporal(point) x region → temporal(point)*
tintersection *point x temporal(region) → temporal(point)*
tintersection *temporal(point) x temporal (region) → temporal(point)*

- The **distance** operation with signature

  distance *point x point → float*
  has **lifted** versions

tdistance (<-->) *temporal(point)* x *point -> temporal(float)*
tdistance (<-->) *point* x *temporal(point) -> temporal(float)*
tdistance (<-->) *temporal(point)* x *temporal(point) -> temporal(float)*

Spatiotemporal predicates

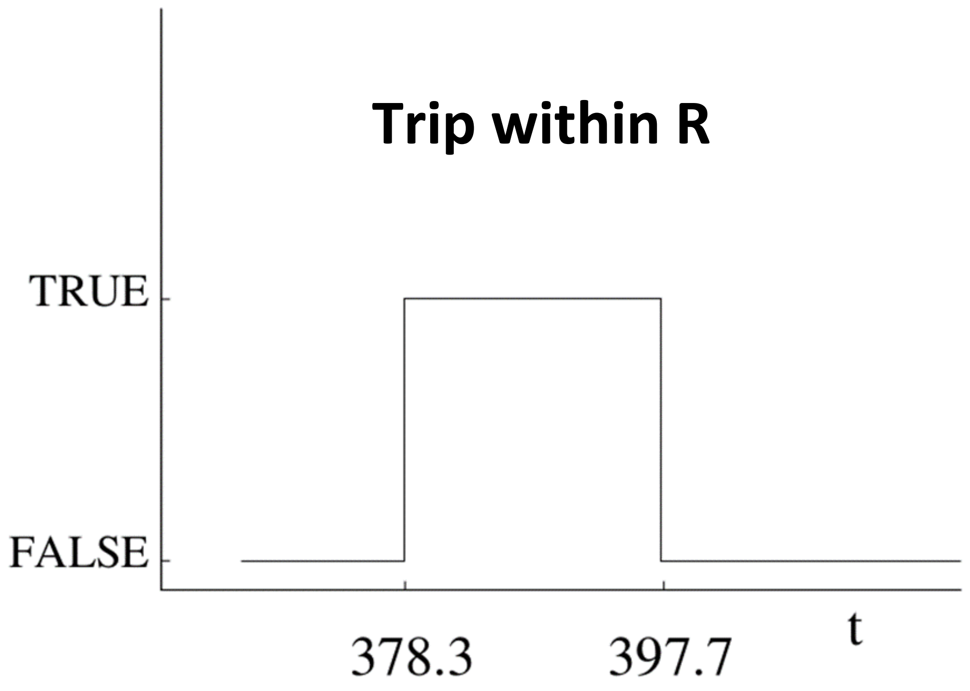


Lifted topological predicates

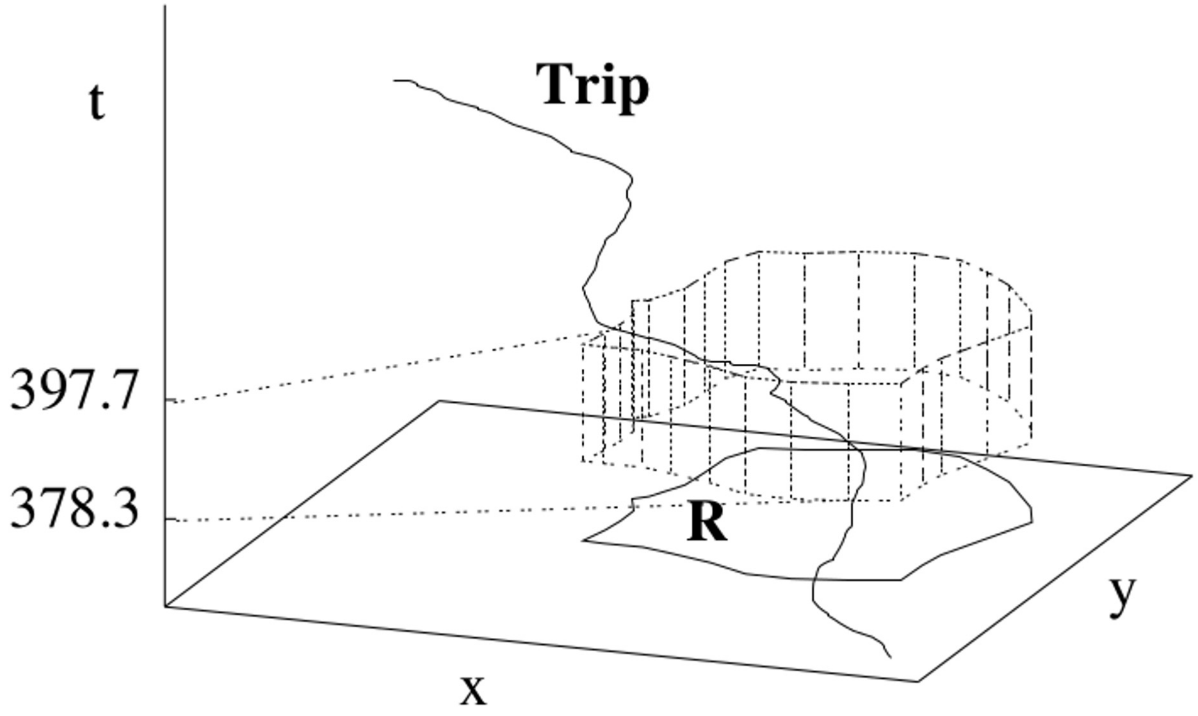
- **A temporally lifted** topological predicate is a function from two spatio-temporal data types to *temporal(bool)*
- Example **within:** *point x region → bool*

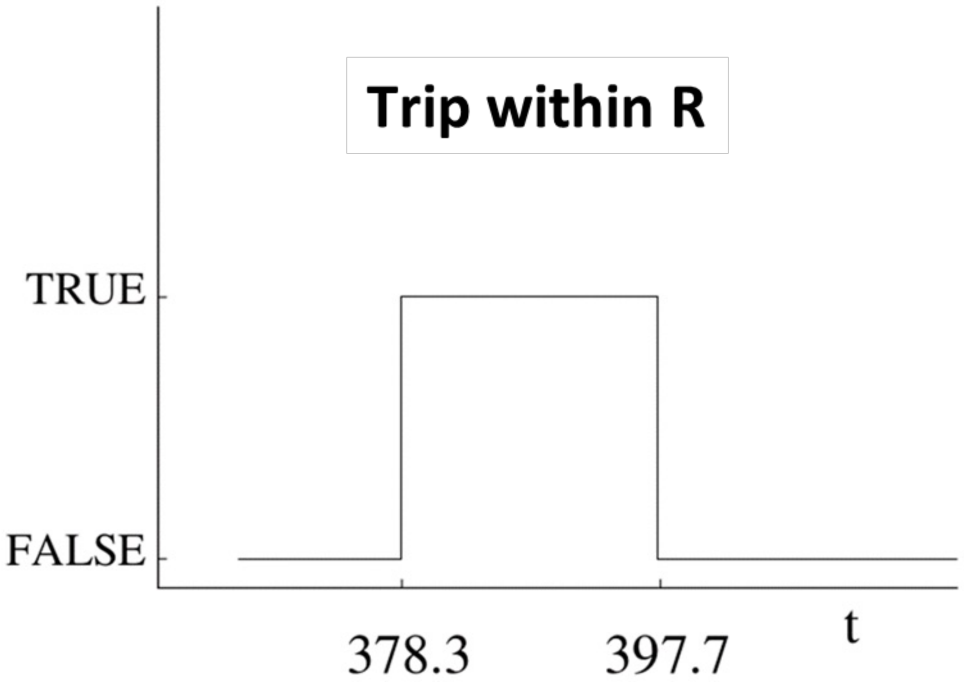
  within: *temporal(point) x temporal(region) → temporal(bool)*

- **within** yields *true* for each time point at which the moving point is inside the evolving region, undefined (⊥) when the point or region is undefined, *false* in all other cases.



Spatiotemporal predicates





Lifting operations

We now have available in our query language all the operations of the algebra in their lifted versions:

- predicates
- set operations
- aggregation operations
- numeric operations
- distance and direction operations
- Boolean operations

We now give examples of queries that can be expressed over this abstract model

Queries over the abstract model (in red temporal types)

- Example Schema:

```
Trips (TripId: int, TripDate: date, VehId: int, Trip: tgeompoint)
```

- Query 1: Trips longer than 50 kms

```
SELECT TripId
FROM Trips
WHERE length(trajectory(Trip)) > 50
```

- Query 2: Which are the trips that took less than two hours?

```
SELECT TripId
FROM   Trips
WHERE  duration(Trip) <= 2
```

Queries (extended schema)

- Extended Schema:

```
Trips    (TripId: int, TripDate: date, VehId: int, Trip: tgeompoint)
Vehicles(VehId int, Licence text, Type text, Model text);
Points  (PointId int, PosX float, PosY  float, Type text, Geom Geometry(Point))
Communes(CommuneId int, Name text, Area float,  Geom Geometry(Polygon),…)
Weather (Kind: string, Area: tregion)
```

- Query 3: Find all pairs of vehicles that during their trip came closer to each other than 500 meters

```
SELECT A.TripId, B.TripId
FROM   Trips A, Trips B
WHERE  A.TripId < B.TripId and minValue(tdistance(A.Trip, B.Trip)) < 0.5
```

- Query 4: Which trips passed through the commune of Ixelles?

```
SELECT id
FROM Trips, Communes
WHERE Name = ’Ixelles’ and duration(tintersection(Trip,Area)) > 0
```

Queries (cont.)

```
WITH Brussels AS (Select ST_UNION (Area) as Area FROM Communes)
WITH Storm AS    (SELECT Area FROM weather WHERE Kind = 'Snow Storm'),
WITH TripLic AS  (SELECT Trip FROM Trips T JOIN  Vehicles V on T.VehId = V. VehId
                  WHERE Licenseid = 'ABC393')
```

- Query 5: *Duration of snowstorms over Brussels*

```
SELECT duration(getTime(atGeometry(Storm.Area,(SELECT Brussels.Area FROM Brussels)))
FROM Storm
```

- Query 6: *Where was vehicle with license ABC393 during snowstorms in Brussels?*

```
SELECT trajectory(atTime(Trip, getTime(atGeometry(Storm.Area,(SELECT Brussels.Area
       FROM Brussels))))
FROM TripsLic, Storm
```

Queries (cont.)

- Query 7: *At what times did the area of a storm shrink?*

```
SELECT getTime(atValues(derivative(Storm.Area), open(range(minreal, 0))))
FROM Storm
```

`open(range(minreal, 0))` means that the derivative is negative.

- Query 8: *Show the parts of the route of the trips of vehicle with licence 'ABC393' when the vehicle's speed was at least 30 km/h.*

```
SELECT trajectory(atTime(Trip, getTime(atValues(speed(Trip), range(30, maxreal))))
FROM TripLic
```

Lifting operations

- Query 9: *At what times did storms split into two separate parts?*

```
SELECT  inst(initial(atValues(no_components(Storm.Area), range(2,2))))
FROM Storm
```

Here `no_components` returns a *temporal(int)* which by the **`atValues`** operation is reduced to the times when it has value "2"; **`initial`** is a (value,time) pair and **`inst`** takes the time instant of the pair.

- Query 10: *Compute for each airport the minimal distance ever from the center of a storm*

```
WITH Airport AS (SELECT Geom FROM Points WHERE  Type = 'Airport')

SELECT id, val(initial(atmin(tdistance(center(Storm.Area), pos(PosX, PosY))))))
FROM Airport, Storm
```

`atmin` returns the *moving real* (tdistance) when Lizzy was closest to the airport across time, `initial`(**`startInstant`** in MobilityDB) returns the initial **intime** of this moving real, and **`val`**(**`getValue`** in MobilityDB) obtains the value of the (value, instant) pair. The instant would be obtained with **`getTimestamp`**.

The **When** operation

- An operation that restricts a temporal value (or moving object) to the times when a condition is fulfilled.

Example:

- Query 11: *List the trips when their speed was over 80 km/h*

```
SELECT Trip from Trips when Speed(Trip) > 80;
```

```
The semantics is:

SELECT atperiods(Trip, getperiods(atValues(speed(Trip)#>80), true)
FROM Trips
WHERE Speed(Trip) > 80
```

- The semantics of this operator can be implemented using lifted functions

```
SELECT atperiods(Trip, getTime(atValues(speed(Trip) > 80, true)))
FROM Trips
```

Lifted predicates semantics

- For each lifted predicate there is a preferred temporal semantics
- Example:
  - “A plane route did not encounter a storm” -> disjointedness **during the whole common lifetime** ($\forall_{\cap}$disjoint)
  - “species living in a climatic region” -> “insideness” **using the species lifetime** ($\forall_{\pi 1}$within)
- For some spatio-temporal predicates the default expected behavior is existential

- For **two moving regions** there are 8 possible predicates
- For a **temporal point and a temporal region** there are three basic predicates **Disjoint**, **Meet**, and **Contains/Inside**
- For **two moving points** (only possibility in MobilityDB) we have two predicates: **Disjoint** and **Meet**

Basic spatiotemporal predicates

For two **moving regions** we have the following predicates (lowercase for non-temporal, uppercase temporal):

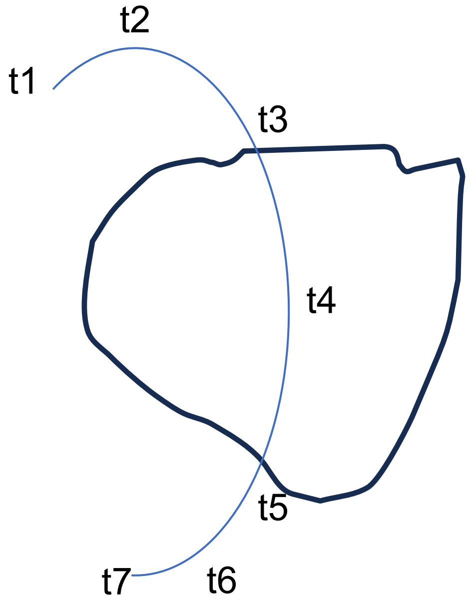
Disjoint	:=	$\forall_{\cap}$ **disjoint**
Meet	:=	$\forall_{\cup}$ **meet**
Overlap	:=	$\forall_{\cup}$ **overlap**
Equal	:=	$\forall_{\cup}$ **equal**
Covers	:=	$\forall_{\pi_2}$ **covers**
Contains	:=	$\forall_{\pi_2}$ **contains**
CoveredBy	:=	$\forall_{\pi_1}$ **coveredBy**
Inside	:=	$\forall_{\pi_1}$ **inside**

For example, Disjoint means disjointness throughout the whole intersection of the lifespans.
For a **temporal point and a temporal region** we have the three basic predicates **Disjoint**, **Meet**, and **Contains/Inside.**
For **two moving points** (only possibility in MobilityDB) we have the predicates **Disjoint** and **Meet**

Composition of spatiotemporal predicates

- We can combine basic spatiotemporal predicates to capture change of spatial situations t1 over time
- Relationships between two spatio-temporal objects modeled by sequences of spatial and (basic) spatio-temporal predicates
- Example: a moving point enters a (temporal) region, then exits it, and finally reenters the region (that is, P *crosses* R). At t2 P is outside R. At t4 P is inside R (P *enters* R). At t6 it is outside R again.

Predicate	**Holds**	**Observation**
Disjoint(P, R)	During I1 = [*t1, t3*)	*t2* ∈ I1
meet(P, R)	at *t3*	
Inside(P, R)	During I2 = (*t3, t5*)	*t4* ∈ I2
meet(P, R)	at t5	
Disjoint(P, R)	During I3 = (*t5, t7*]	*t6* ∈ I3



Composition of spatiotemporal predicates

- Compositions are denoted by a sequence of predicates
- Example:
  - **Disjoint** $\triangleright$ **meet** $\triangleright$ **Inside** $\triangleright$ **meet** $\triangleright$ **Disjoint**
  - This is an abbreviation for: **Disjoint** until **meet** then (**Inside** until **meet** then **Disjoint**)
- Notation: **meet** is a spatial predicate, **Inside**, **Disjoint**, **Meet** (capitalized) are spatiotemporal predicates
- We can define other predicates in a natural way.
- For a *tpoint* and a *tregion*:

$$\begin{array}{lll} \textbf{Enter} & := & \textbf{Disjoint} \triangleright \textbf{meet} \triangleright \textbf{Inside} \\ \textbf{Cross} & := & \textbf{Disjoint} \triangleright \textbf{meet} \triangleright \textbf{Inside} \triangleright \textbf{meet} \triangleright \textbf{Disjoint} \end{array}$$

- For two *tregions*:

$$\begin{array}{lll} \textbf{Cross} & := & \textbf{Disjoint} \triangleright \textbf{meet} \triangleright \textbf{Overlap} \triangleright \textbf{coveredBy} \triangleright \textbf{Inside} \triangleright \\ & & \textbf{coveredBy} \triangleright \textbf{Overlap} \triangleright \textbf{meet} \triangleright \textbf{Disjoint} \end{array}$$

Composition of spatiotemporal predicates

- Consider again the tables Trips, and Weather

```
Trips (TripId: int, TripDate: date, VehId: int, Trip: tgeompoint)
Weather(Kind: string, Extent: tregion)
```

Query 12: “Determine the trips entering a snowstorm in the road”.

- *Problem: For each car/snowstorm combination check whether the car and the storm were disjoint for a while, then they met at one point in time, and the car was inside the storm for a while*
- *The development of entering a storm is only true* ***if each of the three subqueries are true*** *and if they* ***have occurred one after the other***

- *This can be written as the next slide shows*

Composition of spatiotemporal predicates

Query 12: “Determine the trips entering a snowstorm in the road and ending during the storm.”

Answer **without using the spatiotemporal predicates**

```
SELECT id
FROM Trips, Weather
WHERE kind = ”snowstorm" AND
             not(getValue(atTime(Trip, startTimestamp(getTime(Trip)))) within
                 getValue(atTime(Extent, startTimestamp(getTime(Trip))))) AND
                 getValue(atTime(Trip, endTimestamp(getTime(Trip)))) within
                 getValue(atTime(Extent, endTimestamp(getTime(Trip))))
```

- First compute the starting time of the trip (***startTimestamp(getTime(Trip))***), then the (time, point)-pair is computed at this instant by the operation ***atTime.*** The **extent** of the storm at that instant is determined as a region.
- If **within** is false, the car was outside of the storm when starting the trip
- Similarly, we compute the end of the trip (**endTimestamp(getTime(Trip))**). If **within** is true, then the car must have entered the storm and ended within the latter.

Credits - Readings

These slides were based on material from:

Modeling and Querying History of Movement. Ch.4, Moving Objects Databases, Ralf Hartmut Güting, Markus Schneider, 2005

Güting, Ralf & Böhlen, Michael & Erwig, Martin & Jensen, Christian & Lorentzos, Nikos & Schneider, Markus & Vazirgiannis, Michalis. A Foundation for Representing and Querying Moving Objects. ACM Transactions on Database Systems (TODS). 25. 1-42, 2000

https://dl.acm.org/doi/pdf/10.1145/352958.352963