Chapter 2
Spatio-Temporal Continuous Queries

Abstract Spatio-temporal stream processing in general refers to a class of software systems for processing of high volume spatio-temporal data streams with very low latency, i.e. in near real-time. Motivated by the limitation of DBMS, the database community developed data stream management systems (DSMSs), as a new class of management systems oriented toward processing large data streams in a near real-time. Despite differences these between these two classes of management systems, DSMSs resemble DBMSs—they process data streams using SQL and operators defined by the relational algebra. This chapter gives an insight into spatio-temporal stream processing at conceptual level, i.e. from the DSMS user perspective.

Keywords Data streaming · Data stream architectures · Spatio-temporal data streams · GeoStreaming · Continuous query processing · Stream windows

2.1 Foundation of Continuous Query Processing

When processing data streams, there are two inherent temporal domains to consider:

- *Event* time,[1] which is the time at which the event itself actually occurred in the real world.
- *Processing* time, which is the current time according to the system clock, at which an event is observed during processing.

Most DSMS are founded on extensions of the relational model and corresponding query languages. Consequently, data stream items could be viewed as relational tuples with one important distinction: they are *time ordered*. The ordering is defined either explicitly by event time (a.k.a valid time) or implicitly by processing time. Similarly to [9], we define *time domain*, *time instant* and *time interval* as follows:

Definition 2.1 (*Time Domain*) A time domain $\mathbb{T}$ is a pair $(T; \leqslant)$ where T is non-empty set of discrete time instants and $\leqslant$ is total order on T.

Definition 2.2 (*Time Instant*) A time instant τ is any value from T, i.e. $\tau \in T$.

[1] The terms *event time* and *valid time* are often used interchangeably.

Z. Galić, *Spatio-Temporal Data Streams*, SpringerBriefs
in Computer Science, DOI 10.1007/978-1-4939-6575-5_2

Definition 2.3 (*Time Period*) A time period represents extend in time defined by the temporal positions of the time instants at which it begins (τ_{begin}) and ends (τ_{end}).

Discrete time domain implies that every time instant has an immediate successor (except the last, if any) and immediate predecessor (except the first, if any).

Definition 2.4 (*Time Interval*) A time interval consists of all distinct time instants $\tau \in T$ and could be open, closed, left-closed or right-closed:
$(\tau_{begin}, \tau_{end}) = \{\tau \in \mathbb{T} \mid \tau_{begin} < \tau < \tau_{end}\}$,
$[\tau_{begin}, \tau_{end}] = \{\tau \in \mathbb{T} \mid \tau_{begin} \leq \tau \leq \tau_{end}\}$,
$[\tau_{begin}, \tau_{end}) = \{\tau \in \mathbb{T} \mid \tau_{begin} \leq \tau < \tau_{end}\}$,
$(\tau_{begin}, \tau_{end}] = \{\tau \in \mathbb{T} \mid \tau_{begin} < \tau \leq \tau_{end}\}$.

Spatio-temporal data streams have two distinct features which differentiate them from conventional data streams based on relational model:

- Event time of data stream tuple is defined by temporal attribute $\mathscr{A}_\theta$.
- Shape and location of an object of interest described by a data stream tuple defined by spatial attribute $\mathscr{A}_\sigma$.

First feature implies that each data stream tuple has *event timestamp* [27] generated by the source, which classifies spatio-temporal streams into a class of *explicitly timestamped data streams* [31].

Spatial domain is a set of homogeneous object structures (values) which provide a fundamental abstraction for modelling the geometric structure of real-world phenomena in space. Points, lines, polygons and surfaces are the most popular and fundamental abstractions for mapping a geometric structure from 3D space into 2D space [44]. In order to locate object in space, the embedding space must be defined as well. The formal treatment of spatial domain requires a definition of mathematical space. Although Euclidean $\mathscr{R}^2$ embedding space seems to be dominant, in some cases other spaces (metric, vector, topological) are more important.

Simple object structures (point, continuous line, simple polygon) are not closed under the geometric set operations (difference, intersection, union). This means that geometric set operations can produce *complex* spatial objects (multipoints, multilines, multipolygons, polygons with holes, etc.). For this reason, spatial domain should include spatial objects with complex structure.

Definition 2.5 (*Spatial Domain*) A spatial domain $\mathscr{D}_\sigma$ is a set of spatial objects with simple or complex structure.

At first glance, complex spatial objects may appear to be overwhelming in DSMS context. However, we want to relay on and explore extensive research on abstract spatial data types in spatial DBMS and GIS. More specifically, we are going to follow *abstract spatial data types* framework, as defined in [24, 25], where the internal structure of a spatial object is hidden from the user, and could only be accessed through a set of predefined operations (Fig. 2.1). Spatial domain, even though consisting of simple object structures only, is obviously not atomic but rather structured,

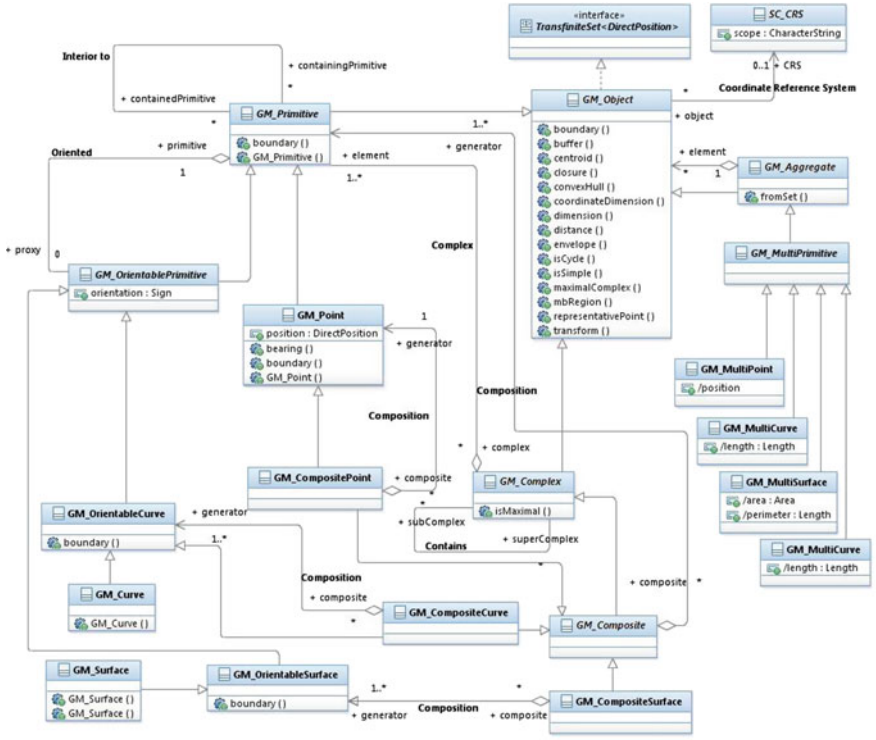

Fig. 2.1 ISO 19107 abstract spatial data types [24]

and as a consequence, spatio-temporal data streams rely on object-relational (or object-oriented) paradigm.

Having defined the time and spatial domains, we define spatio-temporal data stream schema:

Definition 2.6 (*Spatio-Temporal Data Stream Schema*) A spatio-temporal data stream schema $\Sigma_{\sigma\theta}$ is represented as a set of attributes $\langle \mathscr{A}_1, \mathscr{A}_2, \ldots, \mathscr{A}_n \rangle$ of finite arity n. One of the attributes (denoted by $\mathscr{A}_\sigma$) has associated spatial domain $\mathscr{D}_\sigma$, and one of the attributes (denoted by $\mathscr{A}_\theta$) has associated temporal domain $\mathscr{D}_\theta$, i.e. $\mathbb{T}$. The values of other $n-2$ attributes are drawn either from atomic type domain $\mathscr{D}_{\alpha_i}$ or complex type domain $\mathscr{D}_{\chi_i}$.

Finally, spatio-temporal data stream is defined in the following way:

Definition 2.7 (*Spatio-Temporal Data Stream*) A spatio-temporal data stream $\mathscr{S}_{\sigma\theta}$ is a possibly infinite sequence of tuples belonging to the schema of $\Sigma_{\sigma\theta}$ and ordered by the increasing values of $\mathscr{A}_\theta$.

A spatio-temporal stream tuple t represents an *event*, i.e. an instantaneous fact capturing information that occurred in real-world at time instant τ, defined by event

timestamp. Event timestamp offers a unique time indication for each tuple, and therefore cannot be either undefined (i.e., event timestamp cannot have a `null` value) or mutable. In the sequel, we consider explicitly timestamped data streams, ordered by the increasing values of their event timestamps. Analogously to [42], we also define temporal ordering:

Definition 2.8 (*Temporal Order*) A temporal order is surjective (many-to-one) mapping $f_\Omega : \mathscr{D}_{\mathscr{S}_{\sigma\theta}} \twoheadrightarrow \mathbb{T}$ from data type domain $\mathscr{D}_{\mathscr{S}_{\sigma\theta}}$ of the tuples belonging to a data stream $\mathscr{S}_{\sigma\theta}$ to time domain $\mathbb{T}$, such that following holds:

1. Timestamp existence: $\forall s \in \mathscr{S}_{\sigma\theta},\ \exists \tau \in \mathbb{T},\ \textit{such that} f_\Omega(s) = \tau$.
2. Timestamp monotonicity: $\forall s_1, s_2 \in \mathscr{S}_{\sigma\theta},\ \textit{if} s_1.\mathscr{A}_\theta \leq s_2.\mathscr{A}_\theta,\ \textit{then} f_\Omega(s_1) \leq f_\Omega(s_2)$.

DSMS could tag each stream tuple with its arrival timestamp, using system's local clock. A stream with system timestamps can be processed like a regular stream with application event timestamps, but we should be aware that application event time and system time are not necessarily synchronized [29].

We distinguish between *raw* streams produced by the sources and *derived* streams produced by continuous queries and their operators. In either case, we model individual stream elements as object-relational tuples with a fixed spatio-temporal schema.

The raw streams are an essential input for a broad range of applications such as traffic management and control, routing, and navigation. To become useful, the raw streams have to be related to the underlying transportation network by means of *map-matching* algorithms. For example, the map-matching is one of the key operations found in Intelligent Transportation Systems—a map-matching UDF could be applied on raw stream to produce derived stream.

2.1.1 Running Example

As a running example,[2] let us consider the raw stream generated by GPS and speed sensors embedded into a moving object:

```
CREATE STREAM gpsStream (
   id          VARCHAR(8),
   lat         REAL,       // Latitude
   lon         REAL,       // Longitude
   elevation   SMALLINT,   // Ellipsoidal height
   speed       REAL,       // Speed [km/h]
   timestamp   TIMESTAMP VALIDTIME
)
```

[2]The notation used in this book is close to the notation used in [15, 16], which is itself based on TelegraphCQ [14], PostGIS [43] and SQL/MM—Spatial [25]. Due to the simplicity and clarity of the syntax, as well as to avoid possible confusion with spatio-temporal data types, prefix `ST_` has been omitted.

```
ORDERED BY timestamp;
ALTER STREAM gpsStream ADD WRAPPER gpsWrapper;
```

Stream might have multiple attributes of `TIMESTAMP` type, but only one should have `VALIDTIME` constraint. This constraint implicitly determines the read-only attribute according to which the stream is ordered. Wrappers are user-defined data acquisition functions that transform the sequence of bytes into a raw stream, and `ALTER STREAM` associates the raw stream with a wrapper.

An example of a sequence of `gpsStream` tuples (Fig. 2.2):

```
. . .
"W-45084A" 48.20781333 16.43832500 221 73.2 2015-10-19 11:50:30
"W-45084A" 48.20795500 16.43853167 221 79.2 2015-10-19 11:50:31
"W-45084A" 48.20809167 16.43873667 220 77.4 2015-10-19 11:50:32
"W-45084A" 48.20823500 16.43894667 220 80.3 2015-10-19 11:50:33
"W-45084A" 48.20838167 16.43916333 220 82.5 2015-10-19 11:50:34
"W-45084A" 48.20851667 16.43936000 220 75.4 2015-10-19 11:50:35
"W-45084A" 48.20865833 16.43957167 220 80.1 2015-10-19 11:50:36
"W-45084A" 48.20881000 16.43978667 219 83.6 2015-10-19 11:50:37
"W-45084A" 48.20894667 16.43997667 219 74.7 2015-10-19 11:50:38
"W-45084A" 48.20908667 16.44017833 219 77.8 2015-10-19 11:50:39
"W-45084A" 48.20923167 16.44038333 218 79.8 2015-10-19 11:50:40
"W-45084A" 48.20937667 16.44059167 218 80.5 2015-10-19 11:50:41
. . .
```

Derived stream with position modelled as a point on WGS84 ellipsoid, can be quite natural for applications involving spatio-temporal objects whose moving is related to the Earth's surface: airplanes, tankers, combat aircrafts, cruise missiles, drones, etc.:

```
CREATE STREAM movingObjectWGS84 AS
SELECT id,
       SetSRID(Point(lon,lat,elevation),4326)::GEOGRAPHY
       AS wgsPosition,
       speed,
       timestamp
FROM gpsStream;
```

Parameter 4326 is EPSG[3] identifier of WGS 84 spatial reference system, and `::` is shorthand for type casting.

Here is a derived sequence of `movingObjectWGS84` tuples:

```
. . .
"W-45084A" POINT(48.20781333 16.43832500 221) 73.2 2015-10-19 11:50:30
"W-45084A" POINT(48.20795500 16.43853167 221) 79.2 2015-10-19 11:50:31
"W-45084A" POINT(48.20809167 16.43873667 220) 77.4 2015-10-19 11:50:32
"W-45084A" POINT(48.20823500 16.43894667 220) 80.3 2015-10-19 11:50:33
"W-45084A" POINT(48.20838167 16.43916333 220) 82.5 2015-10-19 11:50:34
"W-45084A" POINT(48.20851667 16.43936000 220) 75.4 2015-10-19 11:50:35
```

[3] http://spatialreference.org/ref/epsg/4326/.

```
"W-45084A" POINT(48.20865833 16.43957167 220) 80.1 2015-10-19 11:50:36
"W-45084A" POINT(48.20881000 16.43978667 219) 83.6 2015-10-19 11:50:37
"W-45084A" POINT(48.20894667 16.43997667 219) 74.7 2015-10-19 11:50:38
"W-45084A" POINT(48.20908667 16.44017833 219) 77.8 2015-10-19 11:50:39
"W-45084A" POINT(48.20923167 16.44038333 218) 79.8 2015-10-19 11:50:40
"W-45084A" POINT(48.20937667 16.44059167 218) 80.5 2015-10-19 11:50:41
. . .
```

We may define another derived data stream with a position modelled as a point in two-dimensional Euclidean space as:

```
CREATE STREAM movingObject AS
SELECT id,
       Force_2D(Transform(wgsPosition::GEOMETRY,3416))
       AS position,
       speed,
       timestamp
FROM movingObjectWGS84;
```

Function `Transform` transforms a point on WGS84 ellipsoid (`wgsPoint`) to the specified spatial reference system. Parameter 3416 is a unique identifier (SRID) used to unambiguously identify EPSG:3416[4] spatial reference system, which incorporates European Terrestrial Reference System 1989 (ETRS89) and Lambert projection.

Finally, here is a tuple sequence of derived `movingObject` stream:

```
. . .
"W-45084A" POINT(630656.02 483284.08) 73.2 2015-10-19 11:50:30
"W-45084A" POINT(630670.74 483300.43) 79.2 2015-10-19 11:50:31
"W-45084A" POINT(630685.35 483316.22) 77.4 2015-10-19 11:50:32
"W-45084A" POINT(630700.30 483332.76) 80.3 2015-10-19 11:50:33
"W-45084A" POINT(630715.74 483349.70) 82.5 2015-10-19 11:50:34
"W-45084A" POINT(630729.74 483365.28) 75.4 2015-10-19 11:50:35
"W-45084A" POINT(630744.82 483381.64) 80.1 2015-10-19 11:50:36
"W-45084A" POINT(630760.17 483399.13) 83.6 2015-10-19 11:50:37
"W-45084A" POINT(630773.62 483414.87) 74.7 2015-10-19 11:50:38
"W-45084A" POINT(630787.97 483431.02) 77.8 2015-10-19 11:50:39
"W-45084A" POINT(630802.54 483447.74) 79.8 2015-10-19 11:50:40
"W-45084A" POINT(630817.36 483464.46) 80.5 2015-10-19 11:50:41
. . .
```

Previous two examples illustrate the concept of derived streams. Of course, it would be possible to define corresponding spatio-temporal raw streams in straight way as:

[4] http://spatialreference.org/ref/epsg/3416/.

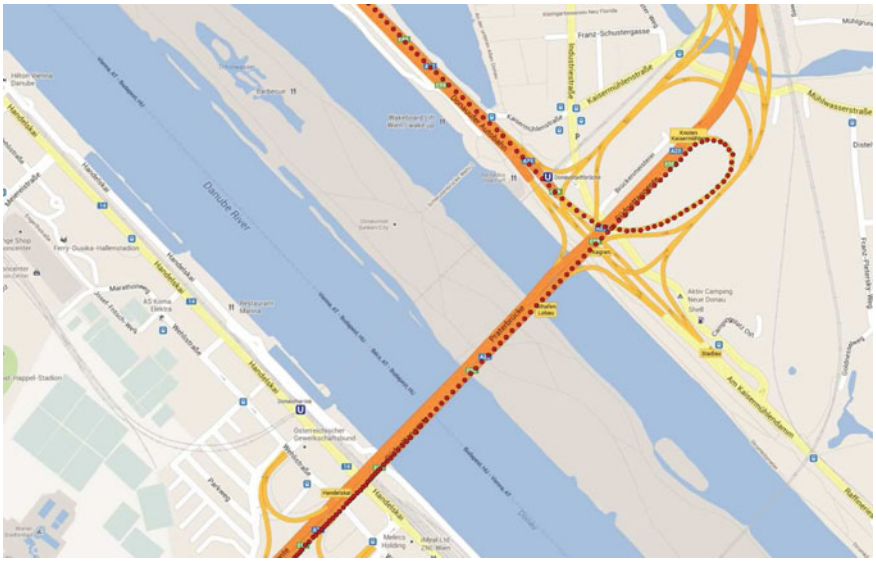

Fig. 2.2 Visualisation of a spatio-temporal stream

```
CREATE STREAM movingObjectWGS84 (
   id          VARCHAR(8),
   wgsPosition GEOGRAPHY(POINT,4326)
   speed       REAL,
   timestamp   TIMESTAMP VALIDTIME
)
ORDERED BY timestamp;
ALTER STREAM movingObjectWGS84 ADD WRAPPER moWrapperWGS84;
```

and

```
CREATE STREAM movingObject (
   id        VARCHAR(8),
   position  GEOMETRY(POINT, 3416)
   speed     REAL,
   timestamp TIMESTAMP VALIDTIME
)
ORDERED BY timestamp;
ALTER STREAM movingObject ADD WRAPPER moWrapper;
```

In both cases, complete logic of data acquisition and transformation of bytes into a row streams is encapsulated into corresponding wrappers.

2.2 Stream Windows

Most of DSMS extend and modify database query language (such as SQL) to support efficient continuous queries on data streams. As stated before, queries in DSMS run continuously and incrementally produce new results over time. The operators in continuous queries (selection, projection, join, aggregation, etc.) compute on tuples as they arrive and do not suppose the data stream is finite, which has a significant negative implications. Some operators (Cartesian product, join, union, set difference, spatial aggregation, etc.) require the entire input sets to be completed. These *blocking* operators will produce no results until the data streams end (if ever), which is obviously a serious obstacle and limitation. To output results continuously and not wait until the data streams end, blocking operators must be transformed into *non-blocking* operators. In fact, queries expressible by non-blocking operators are monotonic queries.

Definition 2.9 (*Monotonic query*) A continuous operator or continuous query Q is monotonic if $Q(\tau) \subseteq Q(\tau')$ for all $\tau \leq \tau'$.

Simple selection over a single stream is an example of monotonic query—at any point in time τ' when a new tuple arrives, it either satisfies selection predicate or it does not, and all the previously returned results (tuples) remain in $Q(\tau')$ [19]. Both standard and spatial aggregate operators always return a stream of length one—they are non-monotonic, and thus blocking.

Definition 2.10 (*Non-blocking query*) A non-blocking continuous operator or continuous query $Q_{\mathscr{NB}}$ is one that produces results (all the tuples of output) before it has detected the end of the input [31].

The problem of transforming the blocking queries into their non-blocking counterpart has long been recognized by data stream researchers. A dominant technique that overcomes this problem is to restrict the operator range to a finite *window* over input streams. Windows were introduced into standard SQL as part of SQL:1999 OLAP functions. In SQL:1999, window is a user-specified selection of rows within a query that determines the set of rows with respect to the current row under examination [33]. The motivation for having the window concept in DSMS is quite different. Window limits and focuses the scope of an operator or a query to a manageable portion of the data stream. A window is *stream-to-relation* operator [6]—it specifies a snapshot of a finite portion of a stream at any time point as a temporary *relation*. In other words, window transforms blocking operators and queries to compute in non-blocking manner. At the same time, the most recent data is emphasized, which is more relevant than the older data in the majority of data stream applications. There are several window types, though two basic types are being extensively used in conventional DSMS: logical, *time-based* windows and physical, *tuple-based* windows. By default, a time-based is refreshed at every time tick and tuple-based window is refreshed when a new tuple arrives. The tuples enter and expire from the window in a *first-in-first-expire* pattern: whenever a tuple becomes old enough, it is expired (i.e.,

deleted) from memory leaving its space to a more recent tuple. As a result, traditional window queries can support only (recent) historical queries, making them not suitable for spatio-temporal queries concerned with the current state of data rather than the recent history [37]. However, these two windows type are not useful in answering an interesting and important class of queries over spatio-temporal data stream, and the *predicate-based* window has been proposed [17], in which an arbitrary logical predicate specifies the window content.

2.2.1 Time-Based Window

The time-based window $\mathscr{W}_{\omega}^{time}$ is defined in terms of the window size ω represented as a time interval $\mathscr{T}_{\omega}$. Formally, it takes a stream $\mathscr{S}$ and the time interval $\mathscr{T}_{\omega}$ as parameters and returns finite, bounded stream, i.e. a temporary finite relation[5]:

$\mathscr{W}_{\omega}^{time} : \mathscr{S} \times \mathscr{T}_{\omega} \rightarrow \mathscr{R}$

The scope of time-based window $\mathscr{W}_{\omega}^{time}$ denotes the most recent time interval, i.e. it consists of the tuples whose timestamp is between the current time τ and $\tau - \omega$. Let τ_0 denotes the time instant that a continuous query specifying a sliding window has effectively started, and τ denotes time instant of the current time. The scope of time interval $\mathscr{T}_{\omega}$ may be formally specified as follows:

$$\mathscr{T}_{\omega}(\tau) = \begin{cases} [\tau_0, \tau] & \text{if } \tau_0 \leq \tau < \tau_0 + \omega \\ [\tau - \omega + 1, \tau] & \text{if } \tau \geq \tau_0 + \omega \end{cases}$$

The qualifying tuples are included in the window on the basis of their timestamps, by appending any newly tuples and discarding older ones. It is worth to note that time-based window is refreshed at every time instant, i.e., with constant refresh time granularity. We will use the following basic syntax to specify time-based window $\mathscr{W}_{\omega}^{time}$ on stream $\mathscr{S}$:

$\mathscr{S}$ [RANGE ω]

Example 2.1 Time-based window `gpsStream [RANGE 20 seconds]` defines a window over input stream `gpsStream` with the size of 20 s. At any time instant, the window output (relation) contains the bag of tuples from the previous 20 s.

There are two important subclasses of time-based window: *now* window $\mathscr{W}_{now}^{time}$ and *unbounded* window $\mathscr{W}_{\infty}^{time}$. Now window $\mathscr{W}_{now}^{time}$, defined by setting τ = NOW and $\omega = 1$, returns tuples of stream (relation) with timestamp equals to NOW. Unbounded window $\mathscr{W}_{\infty}^{time}$, defined by setting $\omega = \infty$ consists of tuples obtained from all tuples from stream $\mathscr{S}$ up to time instant τ. These two special windows are specified using the following syntax:

[5]The *instantaneous relation* in [6].

$\mathscr{S}$ [NOW]
$\mathscr{S}$ [RANGE UNBOUNDED]

Example 2.2 Suppose that Austrian rivers are stored in spatial database,[6] in `river` table created as:

```
CREATE TABLE river (
name :    VARCHAR(16),
geometry: GEOMETRY(POLYGON,3416)
)
```

Next query returns a position (type of `Point`) of moving objects that cross the Danube River at each time instant of the current time:

```
WITH danube AS (
     SELECT geometry FROM river WHERE name = 'Danube'
     )
SELECT position
FROM   movingObject [NOW]
WHERE  Crosses(movingObject.position, danube.geometry)
```

Example 2.3 Consider the following query:

```
WITH danube AS (
     SELECT geometry FROM river WHERE name = 'Danube'
     )
SELECT position
FROM   movingObject [RANGE UNBOUNDED]
WHERE  Crosses(movingObject.position, danube.geometry)
```

This query is monotonic, and produces relation that at time τ contains position of all moving objects that have crossed the Danube River up to τ.

When a stream is referenced in a query without a window specification, an unbounded window $\mathscr{W}_{\infty}^{time}$ is applied by default. Therefore, the next query is equivalent to the previous one:

```
WITH danube AS (
     SELECT geometry FROM river WHERE name = 'Danube'
     )
SELECT position
FROM   movingObject
WHERE  Crosses(movingObject.position, danube.geometry)
```

[6] We suppose that DSMS is coupled with spatial DBMS, but to preclude any potential transaction-processing issues that might occur concurrently with data stream processing, we will assume that the content of persistent relations involved in continuous query remain static [7].

Time-based window can optionally contain a *slide* parameter λ, indicating the granularity at which window slides, i.e. how frequently the window should be refreshed. Accordingly, we define the scope of time interval $\mathscr{T}_\omega$ as

$$\mathscr{T}_\omega(\tau) = \begin{cases} [\tau_0, \tau] & \text{if } \tau_0 \leq \tau < \tau_0 + \omega \wedge (\tau - \tau_0)\, mod\, \lambda = 0 \\ [\tau - \omega + 1, \tau] & \text{if } \tau \geq \tau_0 + \omega \wedge (\tau - \tau_0)\, mod\, \lambda = 0 \\ \mathscr{T}_\omega(\tau - 1) & \text{if } (\tau - \tau_0)\, mod\, \lambda \neq 0, \end{cases}$$

and use the following syntax construction for sliding time-based window $\mathscr{W}^{time}_{\omega,\lambda}$:

$\mathscr{S}$ [RANGE ω SLIDE λ]

Example 2.4 The following query returns every minute the position of moving objects that have crossed the Danube River in the last 5 min:

```
WITH danube AS (
     SELECT geometry FROM river WHERE name = 'Danube'
     )
SELECT position
FROM   movingObject [RANGE '5 minutes' SLIDE '1 minute']
WHERE  Crosses(movingObject.position, danube.geometry)
```

2.2.2 Tuple-Based Window

The tuple-based window $\mathscr{W}^{tuple}$ defines its output stream over time by a window of the last N elements over its input stream. Formally, it takes a stream $\mathscr{S}$ and the natural number $N \in \mathbb{N}^*$ as parameters, and returns temporary finite relation:

$\mathscr{W}^{tuple}_N : \mathscr{S} \times \mathbb{N}^* \rightarrow \mathscr{R}$

At any time instant τ, the output relation $\mathscr{R}$ consists of the N tuples of $\mathscr{S}$ with the largest timestamps $\leq\tau$ (or all tuples if the length of up to τ is $\leq$N. If more than one tuple has the same timestamp, we must choose one tuple in a non-deterministic way to ensure N tuples are returned. For this reason, tuple-based windows may be non-deterministic, and therefore may not be appropriate for streams in which timestamps are not unique.

We will use the following basic syntax to specify tuple-based window $\mathscr{W}^{tuple}$ on stream $\mathscr{S}$:

$\mathscr{S}$ [ROWS N]

The special case of $N = \infty$ is specified by

$\mathscr{S}$ [ROWS UNBOUNDED]

and is equivalent to time-based window

$\mathscr{S}$ [RANGE UNBOUNDED]

Example 2.5 Tuple-based window `gpsStream [ROWS 1]` denotes the "latest" tuple in our `gpsStream`, which is very simple compared to reality. In reality, we

will have a number of moving objects with the same timestamp, and the result of `gpsStream [ROWS 1]` will be ambiguous. As a result, the usability of tuple-based window in spatio-temporal applications is very limited.

2.2.3 Predicate-Based Window

An important issue in data stream query languages is the frequency by which the answer gets refreshed as well as the conditions that trigger the refresh. Coarser periodic refresh requirements are typically expressed as windows, but users in spatio-temporal applications may not be interested only in refreshing the query answer (i.e. window) in response to every tuple arrival. Consequently, a data stream query language should allow a user to express more general refresh conditions based on an arbitrary conditions: temporal, spatial, event, etc. For example, consider the following query:

Q^{π}*: Continuously, report the position of moving objects that cross the Danube River.*

At any time point τ, the *window-of-interest* for query Q^{π} includes only the moving objects that qualify predicate *"cross the Danube River"*. If a moving object O reports a position that crosses (i.e. is within) the Danube River, then it should be in Q^{π}'s window. Whenever O reports position that disqualifies the predicate *"cross the Danube River"*, O expires from Q^{π}'s window. It's important to note that objects enter and expire from Q^{π}'s window in an *out-of-order* pattern. An object is expires (and hence is deleted) from Q^{π}'s window only when the object reports another position that disqualifies the window predicate.

The semantics of time-based window query model reads as follows:

Q^{time}*: Continuously, report the position of moving objects that cross the Danube River in the last* Ω *time units.*

Where Ω is the size of time-based window. The query Q^{time} is semantically different from query Q^{π}: the window-of-interest in Q^{π} includes objects *"crossing the Danube River"* while the window-of-interest in Q^{time} includes objects that *"have crossed the Danube Rriver in the last* Ω *time units"*.

Definition 2.11 (*Predicate-based window query*) A predicate-based window query Q^{π} is defined over data stream $\mathscr{S}$ and window predicate Π over the tuples in $\mathscr{S}$. At any point in time τ the answer to Q^{π} equals the answer to snap-shot query Q^{σ}_{τ}, where Q^{σ}_{τ} is issued at time τ and the inputs to Q^{σ}_{τ} are the tuples in stream $\mathscr{S}$ that qualify the predicate Π at time τ.

We will use the following basic syntax to specify predicate-based window $\mathscr{W}^{\pi}$ on stream $\mathscr{S}$:

$\mathscr{S}$ [Π]

where Π is the predicate that qualifies and disqualifies tuples into (and out of) the window.

Time-based and tuple-based window queries fail to answer some of predicate-based window queries. Let Q_{∞}^{time} denote the query in Example 2.3, which is *de facto* implementation of query Q^{time} with $\Omega = \infty$. The main difference between predicate-based window query Q^{π}

```
WITH danube AS (
     SELECT geometry FROM river WHERE name = 'Danube'
     )
SELECT position
FROM   movingObject AS mo
       [Crosses(mo.position, danube.geometry)]
```

and the query Q_{∞}^{time} in Example 2.3 is that a disqualified tuple in the predicate-based window may result in a negative tuple[7] as an output while a disqualified tuple in the `WHERE` clause predicate of Q_{∞}^{time} does not result in any output tuples. When a tuple t qualifies the `WHERE` predicate of Q_{∞}^{time} and is reported in answer, t will remain in the Q_{∞}^{time} query answer for ever. On the contrary, in the predicate-based window query model, when a tuple t qualifies the predicate Π and is reported in the Q^{π} answer, later, t may be deleted from the query answer if t receives an update so that t does not qualify the window predicate any more [17].

Similarly, the *now* window is semantically different from the predicate-based window. Thus semantics for the query with *now* window in Example 2.2 read as follows:

***Q^{now}**: Report the positions of moving objects that cross the Danube River NOW.*

At any time, the answer of continuously running query Q^{now} will include only position of moving objects that cross the Danube River at time instant $\tau = NOW$. On the other hand, at any time instant τ the predicate-based window query Q^{π} may include positions of moving objects that have crossed the Danube River before τ.

2.3 OCEANUS—A Prototype of Spatio-Temporal DSMS

Most of the relevant research in the area of data streams has been done within several projects, each producing a prototype DSMS. Within the STREAM project [5, 6], streams are transformed into relations using window operators and then queried using standard relational operators. Results can then be transformed back into streams. The Aurora [2, 10] supports continuous queries (real-time processing), views, and ad hoc queries all using substantially the same mechanisms. The Aurora has been superseded by Borealis [1], a distributed multi-processor version of Aurora. The Stream Mill system [8, 30] developed at UCLA with the emphasis on data mining streaming data, enables the user to define custom aggregates which are then used to process streaming data. The Telegraph project at UC Berkley [11] explores adaptive

[7] In the pipelined query execution model with the negative tuples approach, a negative tuple is interpreted as a deletion of a previously produced positive tuple [6, 18].

dataflow architecture which enables it to make scheduling decisions for each tuple. The TelegraphCQ DSMS is based on the Telegraph framework and implemented as an extension of the PostgreSQL DBMS, with very limited spatial support. Work described in [26] represents a unification of two different SQL extensions for data streams and their associated semantics. A time-based execution provides a way to model simultaneity, while a tuple-based execution provides a way to react to primitive events as soon as they are seen by the system. The result is a new model that gives the user control over the granularity at which one can express simultaneity.

Research works and corresponding DSMS prototypes mentioned above have one thing in common: spatial and spatio-temporal data types and operations have been completely neglected.

The PLACE server [38] supports continuous query processing of spatio-temporal streams. It views spatial data streams as automatically changing relations incrementally calculating results and producing positive and negative updates of the result. It allows the user to construct complex queries out of simpler operators such as *inside* or *kNN* (k-nearest-neighbor). The scalable on-line execution (SOLE) algorithm [37] for continuous and on-line evaluation of concurrent continuous spatio-temporal queries over data streams, is implemented in the PLACE. This is one of the first attempts to furnish query processors in data stream management systems with the required operators and algorithms to support a scalable execution of concurrent continuous spatio-temporal queries over spatio-temporal data streams. The SOLE is a unified framework that deals with range queries as well as kNN queries. Furthermore, the SOLE implements a *predicate-based* window [17] especially suitable for spatio-temporal queries, because most of them are concerned with the current state of data rather than the recent history. Although the PLACE server, supported by the SOLE, allows the user to construct complex queries by encapsulating the spatio-temporal query algorithms into a very limited set of primitive spatio-temporal pipelined operators (e.g., *inside* and *kNN*), the native support for spatio-temporal data types and operations on them is missing. Opposed to that, our formally grounded framework natively supports a rich set of spatio-temporal data types and operations and is a significant step towards a full-fledged geospatial DSMS.

Research works presented in [40, 41] try to merge the moving objects and the data streams fields of research. TelegraphCQ is used to manage moving objects trajectories. The emphasis is placed on design and implementation of a powerful map application for managing, querying and visualizing the evolving locations of moving objects.

The GeoInsight framework [28] extends a commercial DSMS processing engine [35] in two directions. It integrates a spatial library to support the online processing of geospatial streaming data and implements an online analytical refinement and prediction (OARP) layer that enables querying of historical streaming data. These extensions allow users to issue various spatio-temporal queries about continuous events and enable the analysis of historical data, together with geospatial streaming data, to refine the answer of real-time queries and predict the answer in the near future. The extensibility infrastructure of the stream processing engine [35] provides user-defined aggregate (UDA), user-defined operator (UDO) and user-defined function

(UDF) facilities, which are used to seamlessly integrate spatial library and OARP modules into the query pipeline of the DSMS processing engine [3].

The work presented in [23] defines a geo-stream as a data stream carrying information about geometry or geometries changing over time by adding new data types called *stream* and *window*. It also defines their abstract semantics and operations on them. This work resolves two important issues: definition of windows semantics through a data type based approach and design of streaming data types, operations and predicates. It presents a novel approach in handling streaming data by encapsulating it inside a single attribute in a relation. This is one of the first works that formally and successfully merges geo-streams and moving objects.

OCEANUS formal framework [15, 16] may be most similar to that of Patroumpas and Sellis [41], Huang and Zhang [23], and Ali et al. [4, 28, 36] However, it overcomes limitations of [41], and is complementary to [4, 23, 28, 36] in the following way:

- Spatial data types in [41] are restricted to point objects only and do not include spatio-temporal operations. The scope of the model is limited to the movement of spatial point entities (i.e. spatial entities are considered to have no extent). The model also includes a restricted set of spatial predicates and operators proposed in an ad-hoc fashion, thus lacking in generality. The TelegraphCQ stream engine is used as a basis for the system, but with PostgreSQL built-in spatial operators and, therefore, with a rather limited and non-standard spatial support.
- OCEANUS is formally grounded in the framework of abstract spatio-temporal data types, it relies on relevant GIS standards [25] and full standardized spatial support for data types and operations, including a full set of spatio-temporal operations.
- The formal methodology taken in [23] is also based on many-sorted algebra, but considers *streams* and *windows* as data types. It includes a proof of the concept [46] as an extension of a moving objects database.
- OCEANUS strategy fully incorporates the data stream paradigm, viewing data streams as unbounded, explicitly timestamped and time-ordered sequences of tuples. It includes a prototype system implemented as a spatio-temporal extension of an object-relational DSMS.
- The approach taken in [4, 28, 36] and OCEANUS share a common feature: both rely on (albeit different) standard spatial library and support native approach to deal with spatial attributes as first-class citizens, to reason about the spatial properties of incoming events and to provide consistency guarantees over space and time. The work presented in [4, 28, 36] utilizes a generic approach to extend a streaming system for a particular application domain. This approach is focused on integrating user-defined modules (UDM) within the query pipeline of a DSMS [3]. Opposed to that, OCEANUS formal framework primarily focuses on formally extending and enriching a DSMS type system to support not only spatial, but also spatio-temporal attributes as first-class citizens. Additionally, it also includes a rich set of spatial, temporal and spatio-temporal operators.

2.3.1 The Type System

There are three kinds of different temporal predicates in spatio-temporal queries. Accordingly, the queries can be classified into three classes: historical, current and future queries [34]. In this section we define a system of data types on the abstract level supporting historical and current queries, using many-sorted algebra and second-order signature, building upon work in [12, 21, 23].

Modeling on the abstract level allows us to make definitions in terms of infinite sets, without fixing any finite representations of these sets. In the spatio-temporal context, a moving point is a continuous curve in 3D (2D + time) or 4D (3D + time) space, i.e. mapping from an infinite time domain into an infinite 2D (or 3D) space domain. The abstract level is the conceptual model that we are interested in, but we should keep in mind that this level of abstraction is not directly implementable and an additional step of choosing a concrete, finite representation is needed.

A many-sorted algebra [32] consists of sets and functions. A *signature* is a pair of sets (S, Ω), the elements of which are called *sorts* and *operations* respectively. Each operation consists of a (k+2)-tuple

$n : s_1 \times \cdots \times s_k \rightarrow s$, with $s_1, \ldots s_k, s \in S$ and $k \geq 0$.

Operation names (*operators*) are denoted by n and $s_1, \ldots, s_k, s \in S$ are sorts. In the case $k = 0$, the operation is called a *constant of sort s*. Informally, a sort denotes the name of a type, and an operation denotes a function [32]. A second-order signature [20] consists of two coupled many-sorted signatures:

1. The first signature defines a type system. In this context the sorts are called *kinds* and denote collections of types, and the *type constructors* have the role of operators. This signature generates a set of *terms*, which are exactly the *types* (Table 2.1).
2. The second signature defines a collection of polymorphic operations over the types of the type system.

The definitions of the *tuple* and *srelation* type constructors already use some extensions to the basic concepts of many-sorted algebra. The first extension is that a type constructor may use as sorts not only kinds (type collections), but also individual types. The second extension makes it possible to define operators taking a variable number of operands. The notation s^+ denotes a list of one or more operands of sort s. The third extension is that if $s_1, \ldots, s_n$ are sorts, then $(s_1 \times \cdots \times s_n)$ is also a sort.

The concepts of temporal types require additional consideration and will be described briefly. To model values of a spatial type σ that change over time, we introduce the notion of a temporal function which is an element of type

$\theta(\sigma) = \mathbb{T} \nrightarrow \sigma$

The temporal functions are the basis of an algebraic model for the spatio-temporal data types, where σ is assigned one of the spatial data types *point, multipoint, linestring* or *polygon*, resulting in *tpoint* as values of type $\theta(point)$, *tmultipoint* as values of type $\theta(multipoint)$, *tlinestring* as values of type $\theta(linestring)$, *tmultilinestring*

Table 2.1 Type system—abstract model

Type constructor	Signature	Target sort
integer, real *string, boolean* *point, multipoint,* *linestring, multilinestring,*		→ BASE
polygon		→ SPATIAL
instant		→ TIME
range	BASE ∪ TIME	→ RANGE
temporal, intime	(BASE ∪ SPATIAL) × WINDOW	→ TEMPORAL
tuple	((BASE ∪ SPATIAL ∪ TIME)$^+$ × TIME)	→ TUPLE
stream	TUPLE	→ STREAM
now, unbounded,		
range		→ WINDOW
srelation	STREAM × WINDOW	→ IRELATION

as values of type θ(*multilinestring*), and *tpolygon* as values of type θ(*polygon*). Consequently, we define base temporal types: *tinteger, tfloat, tboolean*, and *tinstant* which are relevant in the spatio-temporal context.

If we consider a situation where cars equipped with GNSS sensors are moving in a city, each towards a certain goal, *tinteger* could describe the number of cars within a certain area, *tboolean* could describe whether a certain area contains any cars at all, *tfloat* could describe the distance between a car and its goal, and *tinstant* could describe the estimated time of arrival of a car at its goal (which could change in time as traffic conditions vary).

Abstract semantics of BASE, SPATIAL, TIME, RANGE and TEMPORAL sorts have been precisely defined in [12, 13, 20–22]. BASE contains the atomic data types *int, real, bool* and *string*, while SPATIAL contains spatial data types *point, multipoint, linestring* and *polygon*. When applied to a compatible simple data type, temporal constructors produce a new data type e.g. *range(int)*, *temporal(linestring)*, etc.

Abstract semantics of data type α are defined by its carrier set denoted by $\mathscr{C}_\alpha$. Each carrier set must contain an undefined value denoted by $\perp$. Type *instant* represents a point in time and is isomorphic to real numbers. The *range* constructor can be applied to BASE and TIME data types, has a starting and ending value and contains all values in between. The *temporal* constructors can be applied to BASE and SPATIAL data types (Table 2.1).

Table 2.2 Projection operations to domain and range

Operation	Signature	
deftime	*temporal*(α)	$\rightarrow$ *periods*
rangevalues	*temporal*(α)	$\rightarrow$ *range*(α)
locations	*temporal(point)*	$\rightarrow$ *multipoint*
	temporal(multipoint)	$\rightarrow$ *multipoint*
trajectory	*temporal(point)*	$\rightarrow$ *linestring*
	temporal(multipoint)	$\rightarrow$ *multilinestring*
traversed	*temporal(line)*	$\rightarrow$ *polygon*
	temporal(polygon)	$\rightarrow$ *polygon*
instant	*intime*(α)	$\rightarrow$ *instant*
value	*temporal*(α)	$\rightarrow \alpha$

2.4 Operators

As already stated in the previous section, the second signature defines a collection of polymorphic operations over the types of the type system. The complete list of polymorphic operations is rather extensive; a complete list of operators on the static data types (*BASE*, *SPATIAL*, *TIME* and *RANGE*) is defined in [21, 22].[8]

All these static data types are made uniformly time dependent by introducing a type constructor ***temporal***. For a given static data type α, *temporal* returns the type whose values are partial functions from the time domain $\mathbb{T}$ into α.

Let $\mathscr{D}_\alpha$ denote the domain of type α. The domain for type *temporal*(α) is

$\mathscr{D}_{temporal(\alpha)} := \{f | f : \mathscr{D}_{instant} \nrightarrow \mathscr{D}_\alpha\}$

The temporal types realized through the *temporal* type constructor are infinite sets of pairs (instant, value), whereas the *intime* type constructor returns types representing single elements of such sets.

There are two classes of operation on temporal types. Projection class of operations (Table 2.2) deals with projections of temporal values into domain and range, whereas interaction class of operations (Table 2.3) relates the functional values of with values in either their time or their range [22].

2.4.1 Lifting Operations to Spatio-Temporal Streaming Data Types

Operations on non-temporal types are then applied to *TEMPORAL* data types using *temporal*. Temporal lifting is the key concept for achieving consistency of operations

[8] However, OCEANUS prototype, including operations on non-temporal types relies on the existing operations and functions of TelegraphCQ [14] and PostGIS [43].

Table 2.3 Interaction operations with domain and range

Operation	Signature		Space dimension
atinstant	$temporal(\alpha) \times instant$	$\rightarrow intime(\alpha)$	
atperiods	$temporal(\alpha) \times periods$	$\rightarrow temporal(\alpha)$	
initial, final	$temporal(\alpha)$	$\rightarrow intime(\alpha)$	
present	$temporal(\alpha) \times instant$	$\rightarrow boolean$	
	$temporal(\alpha) \times periods$	$\rightarrow boolean$	
at	$temporal(\alpha) \times \beta$	$\rightarrow temporal(\alpha)$	1D
	$temporal(\alpha) \times \beta$	$\rightarrow temporal(min(\alpha, \beta))$	2D
atmin, atmax	$temporal(\alpha)$	$\rightarrow temporal(\alpha)$	1D
passes	$temporal(\alpha) \times \beta$	$\rightarrow boolean$	

on non-temporal and temporal types. All operations defined on static, non-temporal data types are systematically extended to corresponding temporal data types. The idea is to allow argument sorts and the target sort of a signature (the arguments and the result of the operation) to be of the temporal type. Consider the binary topological predicate ***inside*** for points and polygons that determines whether a point lies within a polygon:

inside : $point \times polygon \rightarrow boolean$

It is reasonable to expect that a similar operation can be calculated for a temporal point and a temporal polygon. However, since a temporal point can move in and out of a temporal polygon with time, the result of this lifted operation ***inside*** will also change with time. Thus, a lifted operation ***inside*** will have the following signatures:

inside : $temporal(point) \times polygon \rightarrow temporal(boolean)$
inside : $point \times temporal(polygon) \rightarrow temporal(boolean)$
inside : $temporal(point) \times temporal(polygon) \rightarrow temporal(boolean)$

If we abbreviate the formally defined notation $temporal(\alpha)$ with $t\alpha$, a lifted operation ***inside*** has the following signatures:

inside : $tpoint \times polygon \rightarrow tboolean$
inside : $point \times tpolygon \rightarrow tboolean$
inside : $tpoint \times tpolygon \rightarrow tboolean$

The predicate yields true for each time instant τ at which the temporal point is inside the polygon, undefined whenever the point or the polygon is undefined, and false otherwise.

Formally, for each operation with a signature

$o : s_1 \times \cdots \times s_n \rightarrow s$

its corresponding temporally lifted version is defined by:

$\Uparrow o : \theta(s_1) \times \cdots \times \theta(s_n) \rightarrow \theta(s)$

More specifically, OCEANUS collection of spatio-temporal polymorphic operations consists of temporally lifted versions of spatial operations defined in [25, 43].

2.5 Implementation

A discrete model, as a finite instantiation of an abstract model, is needed for implementation. The discrete model uses the so-called sliced representation: temporal changes of a value along the time dimension are decomposed into fragment intervals called slices. The temporal changes for temporal points and temporal polygons can be represented within each slice by a simple linear function. Figure 2.3 shows a single slice of a polygon changing in time (*tpolygon*) using linear interpolation between subsequent timestamps of two data stream tuples. The design of efficient algorithms for the operations of the abstract model is based on the premise that data structures for different data types are intended to be used within a DSMS, which implies that the data will be placed into the main memory under the control of the DSMS.

In the discrete model, the constructor *temporal* is replaced by type constructors that enable the so-called sliced representation of temporal types (Table 2.4).

Definition 2.12 (*Generic Unit Type*) Let S be a set.

$Unit(S) = Interval(Instant) \times S$

A pair (i, v) from $Unit(S)$ is called a *unit*, i is its *unit interval*, and v is called its *unit function*.

The semantics of a unit type are a function of time, defined during the unit interval:

$\iota : S_\alpha \times Instant \rightarrow \mathscr{D}_\alpha$

where α is the corresponding non-temporal type.

Definition 2.13 (*Mapping Type Constructor*) Let S be a set and $Unit(S)$ a unit type.

$$Mapping(S) = \{U \subset Unit(S) \mid \forall (i_1, v_1) \in U, (i_2, v_2) \in U : i_1 = i_2 \Rightarrow v_1 = v_2, i_1 \neq i_2 \Rightarrow (disjoint(i_1, i_2) \wedge adjacent(i_1, i_2) \Rightarrow v_1 \neq v_2)\}$$

The type constructor *mapping* represents a set of unit types whose values are pairs consisting of a time interval and a simple function, describing temporal development

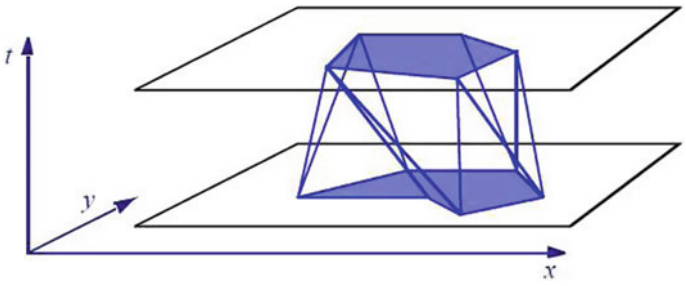

Fig. 2.3 A temporal polygon unit (*source* [22])

Table 2.4 Type system—discrete model

Type constructor	Signature	Target sort
...	...	...
const	(BASE ∪ SPATIAL) × WINDOW	→ UNIT
upoint, umultipoint,		
ulinestring, umultilinestring,		
upolygon		→ UNIT
mapping	UNIT	→ TEMPORAL
...	...	...

of a value during that time interval. A mapping is a set of units whose time intervals are pairwise disjoint.

The spatial attribute $\mathscr{A}_\sigma$ and temporal attribute $\mathscr{A}_\theta$ represent two related but separate attribute values. Together they represent a unique *spatio-temporal reference* of a data stream element. They are implemented with composite temporal data type, whose values are pairs of time instant τ (the value of $\mathscr{A}_\theta$) and a shape/location (the value of $\mathscr{A}_\sigma$). This data type is derived using the type constructor *intime* in a data acquisition process that manages connections with external data sources and the functions which read data from these data sources. The time instant[9] component implicitly used in the type constructor *intime* represents valid time.

Definition 2.14 (*Spatio-Temporal Reference*) A spatio-temporal reference of a data stream element is obtained by applying the type constructor *intime* to an appropriate spatial data type, i.e. *intime* (σ).

The carrier set for *intime*(σ) at the discrete level is
$\mathscr{D}_{intime(\sigma)} := \mathscr{D}_{instant} \times \mathscr{D}_\sigma$

2.5.1 User-Defined Aggregate Functions

The set of aggregate functions available to users of a data stream system is usually limited to five standard functions: *min, max, count, sum, and avg*. Over time it became apparent that users often want to aggregate data in additional ways. Therefore, many data stream systems allow the user to extend the set of available aggregate functions by defining user-defined aggregate functions (UDAF).

[9] In OCEANUS prototype, the *instant* is implemented as PostgreSQL `timestamp` data type.

To create a UDAF, the user must implement at least three functions[10]:

Init	This function is used to initialize any variables needed for the computation later on. Intuitively, it is similar to constructor.
Accumulate	This function is called once for each value aggregated. Generally, this function "adds" the value to the running total computed so far.
Terminate	This function is used to end the calculation and return the final value of the aggregate function. It may involve some calculations on the variables which were defined for use with the aggregate function.

These required functions can be implemented in a external procedural language, but they can also be specified natively using a query language supported by the DSMS. The second approach is less expressive, but enables these three computations to be coded in one single procedure written in SQL, rather than in three separate procedures using an external procedural language. This approach is used in the Expressive Stream Language (ESL), an application language of the Stream Mill system [45].

Similar to continuous queries, a UDAF is considered as *blocking* [19] if it requires its entire input before it can return a result (i.e. it must have a terminate operation), or if, for computing the aggregate result, the data first has to be sorted according to GROUP BY attributes. Such blocking aggregates can only be applied over data streams using a window operator and are termed *windowed aggregates*, while UDAFs invoked without a window operator are termed *base aggregates*. UDAFs in which terminate operation is not defined are called non-blocking and can be applied to data streams freely.

The proposed temporal data type for presenting a spatio-temporal reference of a data stream element enables the implementation of a sliced representation of spatio-temporal objects. A spatio-temporal object is formed of unit types, each consisting of a time interval and a value describing a simple function on the unit time interval. If we consider, for example, two successive spatio-temporal data stream elements with spatio-temporal references consisting of successive point locations and time instants, it is possible to define the unit type that describes the movement of the corresponding point object as a linear function of time.

The complete sliced representation presents a temporal object as a set of temporal units whose time intervals are disjoint, and if adjacent, their values are different. The assembling of the unit types into a temporal object based on the spatio-temporal references of data stream elements can be achieved using UDAF. We have already seen that aggregation can be defined as a state that is modified with every input data. In terms of spatio-temporal data streams, such a state represents a temporal object in a form of described sliced representation (*mapping(upoint)* data type), while a spatio-temporal reference of a data stream element represents new input data (*intime(point)*

[10]The exact names of the functions (and the class to which they must belong) differs depending on the concrete DSMS.

data type). Function `accumulate` that is used in UDAFs, performs the creation of a unit type and adds it to the temporal object. Since this function computes a new state (temporal object) from the old one and the value of the spatio-temporal reference of a new data stream element, the UDAF does not need a terminate function and is thus considered non-blocking. This concept can be implemented using the following PostgreSQL-like syntax:

```
CREATE AGGREGATE tpoint(intime(point))
       SFUNC = accumulate,
       STYPE = mapping(upoint),
       INITCOND = NULL
);
```

An aggregate function is made from one or two ordinary functions: a state transition (*accumulate*) function `SFUNC`, and an optional *terminate* function `FINALFUNC`. Since UDAF for extracting a moving object from a data stream is non-blocking, the terminate function `FINALFUNC` is omitted. A temporary variable of data type `STYPE` holds the current internal state of the aggregate. At each input row, the aggregate argument value(s) are calculated and the state transition function is invoked with the current state value and the new argument value(s) in order to calculate a new internal state value. In the UDAF definition given above, the name of the function is `tpoint`, the data type for storing internal state and the result of the function is `mapping(upoint)` (discrete data type for temporal points) and the data type for the new input data is `intime(point)` (given as an argument of the defined UDAF). An aggregate function provides an initial condition (`INITCOND`), that is, an initial value for the internal state. A spatio-temporal object is not defined before the first input stream tuple has arrived, and the initial value for the internal state is NULL. After all the stream tuples rows have been processed, the final function is invoked once to calculate the aggregates return value. If there is no final function, then the ending state value is returned unchanged [39].

Algorithms 1 and 2 describe an abstract implementation of UDAF `tpoint` and `accumulate` functions respectively.

Algorithm 1 *tpoint(IP)*

input: set *IP* of spatio-temporal references of type *ipoint*
output: temporal point object *tp* of STYPE *tpoint*
summary: A set of spatio-temporal references is aggregated into a temporal point object.

1: INITCOND ← NULL
2: **for all** $ip \in IP$ **do**
3: $tp \leftarrow accumulate(tp, ip)$
4: **end for**
5: **return** *tp*

Algorithm 2 *accumulate(tp,ip)*

input: temporal point object *tp* of type *tpoint*, ST reference *ip* of type *ipoint*
output: temporal point object *tp* of type *tpoint*
summary: Temporal point object *tp* consists of units. Each unit consists of a time interval and a linear function valid during that time interval. The algorithm takes a new ST reference *ip* and tries to add it to the last unit in the moving object *mo*. If that is not possible, it creates a new unit using the new ST reference *ip* and the ending ST reference of the last unit object ip_l. It then adds the new unit to the moving object.

1: **if** $tp = \emptyset$ **then**
2: $fun.x_0 \leftarrow ip.\sigma.x, fun.x_1 \leftarrow 0$ ▷ *fun* - linear function valid during time interval
3: $fun.y_0 \leftarrow ip.\sigma.y, fun.y_1 \leftarrow 0$
4: $ti.start \leftarrow ti.end \leftarrow ip.\theta$ ▷ *tp* - time interval
5: **else**
6: $up \leftarrow tp.units.last$
7: $ip_l \leftarrow up.endpoint$
▷ linear movement function $= f((x_0, x_1, y_0, y_1), t) = (x_0 + x_1 t, y_0 + y_1 t)$
8: $fun.x_0 \leftarrow (-ip.\theta) * (ip_l.\sigma.x - ip.\sigma.x)/(ip_l.\theta - ip.\theta) + ip_l.\sigma.x$
9: $fun.x_1 \leftarrow (ip_l.\sigma.x - ip.\sigma.x)/(ip_l.\theta - ip.\theta)$
10: $fun.y_0 \leftarrow (-ip.\theta) * (ip_l.\sigma.y - ip.\sigma.y)/(ip_l.\theta - ip.\theta) + ip_l.\sigma.y$
11: $fun.y_1 \leftarrow (ip_l.\sigma.y - ip.\sigma.y)/(ip_l.\theta - ip.\theta)$
12: **if** $fun = up.function$ **then**
13: $up.timeinterval.end \leftarrow ip.\theta$
14: **return** *tp*
15: **else**
16: $ti.start \leftarrow ip_l.\theta$
17: $ti.end \leftarrow ip.\theta$
18: **end if**
19: **end if**
20: create new unit object up_{new}
21: $up_{new}.timeinterval \leftarrow ti$
22: $up_{new}.function \leftarrow fun$
23: $tp.add(up_{new})$
24: **return** *tp*

In combination with different window operators, i.e. stream-to-relation operators [6] on spatio-temporal data streams, the proposed UDAF constructs temporal point object in a form of a sliced representation as intermediate results of a continuous query.

2.5.2 SQL-Like Language Embedding: CSQL

The proposed concept is illustrated through CSQL, an SQL-like query language that supports continuous queries over spatio-temporal data streams. In the following examples we show how the query language is extended with spatio-temporal data types and a set of operations over those data types such as *passes, deftime, _value_,*

trajectory, etc. Our examples are based on the following spatio-temporal data stream schema:

```
carStream (
    id:          INTEGER,
    carType:     VARCHAR(8),
    driver:      SMALLINT,
    geometry:    intime(point),
    speed:       REAL,
    tcqtime:     TIMESTAMP TIMESTAMPCOLUMN
)
```

Attribute `tcqtime` represents valid time instant of a stream tuple which obeys the `timestampcolumn` constraint, which is assumed to be monotonically increasing.

The following examples illustrate spatio-temporal continuous queries and their visualization (Fig. 2.4) enabled within OCEANUS prototype:

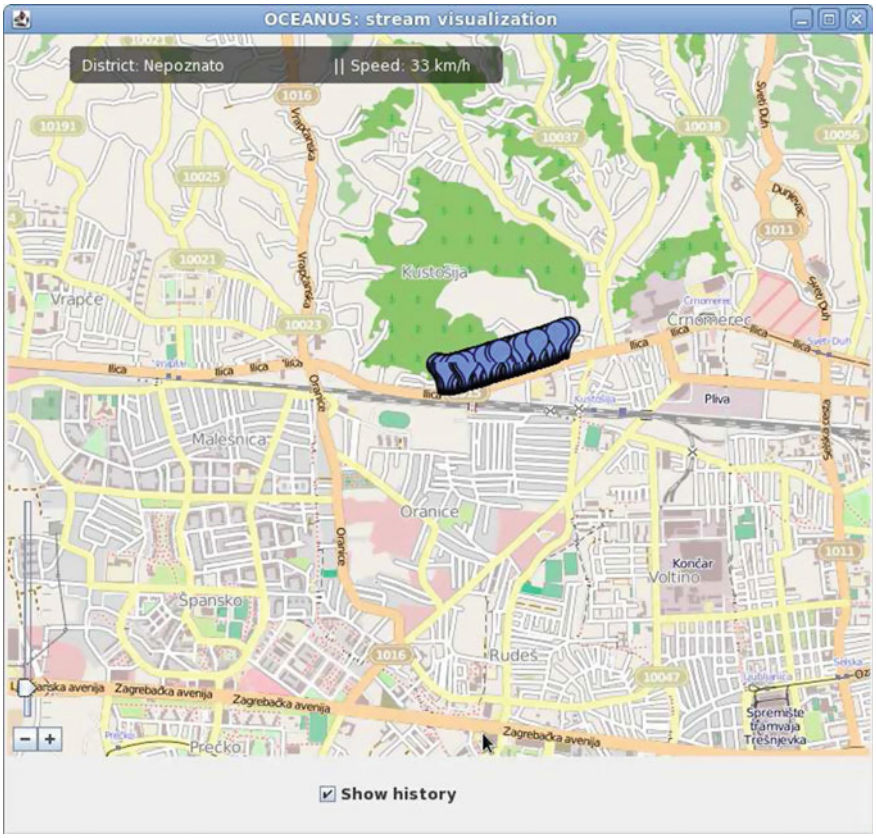

Fig. 2.4 OCEANUS interface—visualisation of ***Q8***

Q1: *Find all cars within the area of interest (e.g. a rectangle).*

```
SELECT id
FROM   carStream [NOW]
GROUP  BY id
HAVING passes(tpoint(geometry),
              GeometryFromText
              ('POLYGON(575000.0 322000.0, 557000.0 323000.0)',
                4326)
              )
```

Q2: *For each car, find its minimal distance from the point of interest during the last half an hour.*

```
SELECT id, _value_(initial(atmin(distance(tpoint(geometry),
               SetSRID(POINT(557234, 322701), 4326))))
FROM   carStream [RANGE 30 minutes]
GROUP BY id
```

Q3: *Find each time instant when car 1007 was heading north (let us say, within* $10°$ *of the exact north direction) during the last ten minutes.*

```
SELECT deftime(at(tdirection(tpoint(geometry)), range(80, 100)))
FROM   carStream [RANGE 10 minutes]
WHERE  car = 1007
```

Q4: *Find all cars/drivers that have travelled more than* $50\,\mathrm{km}$ *during the last hour.*

```
SELECT id, driver
FROM   carStream [RANGE 1 hour]
GROUP  BY id, driver
HAVING length(trajectory(tpoint(geometry))) > 50000
```

Q5: *Report the trajectories of cars with id 1, 2, 3 and 4 in the last* $2\,\mathrm{h}$.

```
SELECT id, trajectory(tpoint(geometry))
FROM   carStream [RANGE 2 hours]
WHERE  id IN (1, 2, 3, 4)
GROUP  BY id
```

Q6: *Find trajectories of the cars that were within* $100\,\mathrm{m}$ *from a point of interest within the last* $30\,\mathrm{min}$.

```
SELECT id, trajectory(tpoint(geometry))
FROM   carStream [RANGE 30 minutes]
GROUP  BY id
HAVING _value_(initial(atmin(distance(tpoint(geometry),
       SetSRID(POINT(557234, 322701), 4326))))) <= 100
```

Q7: *Report the movement of all cars that have crossed the street number 10 in the last* 2 h.

```
SELECT id, trajectory(tpoint(carStream.geometry))
FROM   carStream [RANGE 2 hours], streetTable
WHERE  streetid = 10
GROUP  BY id
HAVING intersects(trajectory(tpoint(carStream.geometry)),
                  streetTable.geometry)
```

Q8: *Continuously report the location of all cars within a particular city district.*

```
WITH district AS (
     SELECT geometry FROM cityDistrict WHERE name = 'Crnomerec'
     )
SELECT id, speed, carStream.geometry AS location
FROM   carStream [NOW]
WHERE  Within(carStream.geometry, district.geometry)
```

References

1. Abadi, D.J., Ahmad, Y., Balazinska, M., Çetintemel, U., Cherniack, M., Hwang, J., Lindner, W., Maskey, A., Rasin, A., Ryvkina, E., Tatbul, N., Xing, Y., Zdonik, S.B.: The design of the Borealis stream processing engine. In: CIDR, pp. 277–289 (2005). http://www.cidrdb.org/cidr2005/papers/P23.pdf
2. Abadi, D.J., Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Aurora: a new model and architecture for data stream management. Int. J. Very Large Databases **12**(2), 120–139 (2003)
3. Ali, M.H., Chandramouli, B., Goldstein, J., Schindlauer, R.: The extensibility framework in Microsoft StreamInsight. In: Abiteboul, S., Böhm, K., Koch, C., Tan, K. (eds.) Proceedings of the 27th International Conference on Data Engineering, ICDE 2011, pp. 1242–1253. IEEE Computer Society (2011)
4. Ali, M.H., Chandramouli, B., Raman, B.S., Katibah, E.: Spatio-temporal stream processing in Microsoft StreamInsight. IEEE Data Eng. Bull. **33**(2), 69–74 (2010)
5. Arasu, A., Babcock, B., Babu, S., Datar, M., Ito, K., Motwani, R., Nishizawa, I., Srivastava, U., Thomas, D., Varma, R., Widom, J.: STREAM: the stanford stream data manager. IEEE Data Eng. Bull. **26**(1), 19–26 (2003)
6. Arasu, A., Babu, S., Widom, J.: The CQL continuous query language: semantic foundations and query execution. Int. J. Very Large Databases **15**(2), 121–142 (2006)
7. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Popa, L., Abiteboul, S., Kolaitis, P.G. (eds.) PODS, pp. 1–16. ACM (2002)
8. Bai, Y., Thakkar, H., Wang, H., Luo, C., Zaniolo, C.: A data stream language and system designed for power and extensibility. In: Yu, P.S., Tsotras, V.J., Fox, E.A., Liu, B. (eds.) CIKM, pp. 337–346. ACM (2006)
9. Bettini, C., Dyreson, C.E., Evans, W.S., Snodgrass, R.T., Wang, X.S.: A glossary of time granularity concepts. In: Temporal Databases, Dagstuhl, pp. 406–413 (1997)
10. Carney, D., Çetintemel, U., Cherniack, M., Convey, C., Lee, S., Seidman, G., Stonebraker, M., Tatbul, N., Zdonik, S.B.: Monitoring streams - a new class of data management applications. In: VLDB, pp. 215–226. Morgan Kaufmann (2002)

11. Chandrasekaran, S., Cooper, O., Deshpande, A., Franklin, M.J., Hellerstein, J.M., Hong, W., Krishnamurthy, S., Madden, S., Reiss, F., Shah, M.A.: TelegraphCQ: continuous dataflow processing. In: Halevy, A.Y., Ives, Z.G., Doan, A. (eds.) SIGMOD Conference, p. 668. ACM (2003)
12. Erwig, M., Güting, R.H., Schneider, M., Vazirgiannis, M.: Abstract and discrete modeling of spatio-temporal data types. In: Laurini, R., Makki, K., Pissinou, N. (eds.) ACM-GIS '98, Proceedings of the 6th International Symposium on Advances in Geographic Information Systems, November 6–7, 1998, Washington, DC, USA, pp. 131–136. ACM (1998). http://doi.acm.org/10.1145/288692.288716
13. Forlizzi, L., Güting, R.H., Nardelli, E., Schneider, M.: A data model and data structures for moving objects databases. In: Chen, W., Naughton, J.F., Bernstein, P.A. (eds.) Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16–18, 2000, Dallas, Texas, USA, pp. 319–330. ACM (2000). http://doi.acm.org/10.1145/342009.335426
14. Franklin, M.J., Krishnamurthy, S., Conway, N., Li, A., Russakovsky, A., Thombre, N.: Continuous analytics: Rethinking query processing in a network-effect world. In: CIDR (2009). www.cidrdb.org
15. Galić, Z., Baranović, M., Križanović, K., Mešković, E.: Geospatial data streams: Formal framework and implementation. Data Knowl. Eng. **91**, 1–16 (2014). http://dx.doi.org/10.1016/j.datak.2014.02.002
16. Galić, Z., Mešković, E., Križanović, K., Baranović, M.: Oceanus: a spatio-temporal data stream system prototype. In: Proceedings of the Third ACM SIGSPATIAL International Workshop on GeoStreaming, pp. 109–115. IWGS '12, ACM, New York, NY, USA (2012). http://doi.acm.org/10.1145/2442968.2442982
17. Ghanem, T.M., Aref, W.G., Elmagarmid, A.K.: Exploiting predicate-window semantics over data streams. SIGMOD Rec. **35**(1), 3–8 (2006)
18. Ghanem, T.M., Hammad, M.A., Mokbel, M.F., Aref, W.G., Elmagarmid, A.K.: Incremental evaluation of sliding-window queries over data streams. IEEE Trans. Knowl. Data Eng. **19**(1), 57–72 (2007)
19. Golab, L., Özsu, M.T.: Data Stream Management. Synthesis Lectures on Data Management. Morgan Claypool Publishers, San Rafael (2010)
20. Güting, R.H.: Second-order signature: a tool for specifying data models, query processing, and optimization. In: Buneman, P., Jajodia, S. (eds.) SIGMOD Conference, pp. 277–286. ACM Press (1993)
21. Güting, R.H., Böhlen, M.H., Erwig, M., Jensen, C.S., Lorentzos, N.A., Schneider, M., Vazirgiannis, M.: A foundation for representing and quering moving objects. ACM Trans. Database Syst. **25**(1), 1–42 (2000)
22. Güting, R.H., Schneider, M.: Moving Objects Databases. Morgan Kaufmann, Amsterdam (2005)
23. Huang, Y., Zhang, C.: New data types and operations to support geo-streams. In: Cova, T.J., Miller, H.J., Beard, K., Frank, A.U., Goodchild, M.F. (eds.) GIScience. Lecture Notes in Computer Science, vol. 5266, pp. 106–118. Springer (2008)
24. ISO 19107:2003: Geographic information – Spatial schema (2008)
25. ISO/IEC 13249-3:2011: Information technology – Database languages – SQL multimedia and application packages – Part 3: Spatial (2011)
26. Jain, N., Mishra, S., Srinivasan, A., Gehrke, J., Widom, J., Balakrishnan, H., Çetintemel, U., Cherniack, M., Tibbetts, R., Zdonik, S.B.: Towards a streaming SQL standard. Proc. VLDB Endow. **1**(2), 1379–1390 (2008)
27. Jensen, C.S., Dyreson, C.E., Böhlen, M.H., Clifford, J., Elmasri, R., Gadia, S.K., Grandi, F., Hayes, P.J., Jajodia, S., Käfer, W., Kline, N., Lorentzos, N.A., Mitsopoulos, Y.G., Montanari, A., Nonen, D.A., Peressi, E., Pernici, B., Roddick, J.F., Sarda, N.L., Scalas, M.R., Segev, A., Snodgrass, R.T., Soo, M.D., Tansel, A.U., Tiberio, P., Wiederhold, G.: The consensus glossary of temporal database concepts - february 1998 version. In: Temporal Databases, Dagstuhl, pp. 367–405 (1997)

28. Kazemitabar, S.J., Demiryurek, U., Ali, M.H., Akdogan, A., Shahabi, C.: Geospatial stream query processing using Microsoft SQL Server StreamInsight. PVLDB **3**(2), 1537–1540 (2010)
29. Krämer, J., Seeger, B.: Semantics and implementation of continuous sliding window queries over data streams. ACM Trans. Database Syst. 34(1) (2009)
30. Law, Y.N., Wang, H., Zaniolo, C.: Query languages and data models for database sequences and data streams. In: Nascimento, M.A., Özsu, M.T., Kossmann, D., Miller, R.J., Blakeley, J.A., Schiefer, K.B. (eds.) VLDB, pp. 492–503. Morgan Kaufmann (2004)
31. Law, Y.N., Wang, H., Zaniolo, C.: Relational languages and data models for continuous queries on sequences and data streams. ACM Trans. Database Syst. **36**(2), 8:1–8:32 (2011)
32. Loeckx, J., Ehrich, H.D., Wolf, M.: Specification of Abstract Data Types. Wiley and B. G Teubner (1996)
33. Melton, J.: Advanced SQL 1999: Understanding Object-Relational, and Other Advanced Features. Elsevier Science Inc., New York (2003)
34. Meng, X., Chen, J.: Moving Objects Management: Models, Techniques and Applications. Tsinghua University Press and Springer (2010)
35. Microsoft: Microsoft StreamInsight (2013). http://msdn.microsoft.com/en-us/library/hh750618(v=sql.10).aspx
36. Miller, J., Raymond, M., Archer, J., Adem, S., Hansel, L., Konda, S., Luti, M., Zhao, Y., Teredesai, A., Ali, M.H.: An extensibility approach for spatio-temporal stream processing using Microsoft StreamInsight. In: Pfoser, D., Tao, Y., Mouratidis, K., Nascimento, M.A., Mokbel, M.F., Shekhar, S., Huang, Y. (eds.) SSTD. Lecture Notes in Computer Science, vol. 6849, pp. 496–501. Springer (2011)
37. Mokbel, M.F., Aref, W.G.: SOLE: scalable on-line execution of continuous queries on spatio-temporal data streams. Int. J. Very Large Databases **17**(5), 971–995 (2008)
38. Mokbel, M.F., Xiong, X., Hammad, M.A., Aref, W.G.: Continuous query processing of spatio-temporal data streams in PLACE. GeoInformatica **9**(4), 343–365 (2005)
39. Obe, R., Hsu, L.: PostgreSQL - Up and Running: a Practical Guide to the Advanced Open Source Database. O'Reilly (2012)
40. Patroumpas, K., Kefallinou, E., Sellis, T.K.: Monitoring continuous queries over streaming locations. In: Aref, W.G., Mokbel, M.F., Schneider, M. (eds.) GIS, p. 81. ACM (2008)
41. Patroumpas, K., Sellis, T.K.: Managing trajectories of moving objects as data streams. In: Sander, J., Nascimento, M.A. (eds.) STDBM, pp. 41–48 (2004)
42. Patroumpas, K., Sellis, T.K.: Maintaining consistent results of continuous queries under diverse window specifications. Inf. Syst. **36**(1), 42–61 (2011)
43. Refractions Research Inc.: PostGIS Manual (2015)
44. Schneider, M.: Spatial Data Types for Database Systems, Finite Resolution Geometry for Geographic Information Systems, Lecture Notes in Computer Science, vol. 1288. Springer (1997)
45. Thakkar, H., Zaniolo, C.: Introducing Stream Mill: User-Guide to the Data Stream Management System, its Expressive Stream Language ESL, and the Data Stream Mining Workbench SMM. Computer Science Department, UCLA (2010)
46. Zhang, C., Huang, Y., Griffin, T.: Querying geospatial data streams in SECONDO. In: Agrawal, D., Aref, W.G., Lu, C., Mokbel, M.F., Scheuermann, P., Shahabi, C., Wolfson, O. (eds.) 17th ACM SIGSPATIAL International Symposium on Advances in Geographic Information Systems, ACM-GIS 2009, November 4–6, 2009, Seattle, Washington, USA, Proceedings, pp. 544–545. ACM (2009). http://doi.acm.org/10.1145/1653771.1653868

http://www.springer.com/978-1-4939-6573-1

Spatio-Temporal Data Streams
Galic, Z.
2016, XIV, 107 p. 28 illus., Softcover
ISBN: 978-1-4939-6573-1