Querying of Data Streams for Spatial Information Management

Leif Harald Karlsen
`leifhka@ifi.uio.no`

University of Oslo

Overview

1. Data Streams
2. Data Stream Management Systems and Continuous Queries
3. Spatial Data Streams
4. Two use cases
5. Spatial Information Management

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity

⋮

```
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
```

⋮

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time t

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time *t*

```
t = 12:00:07

      ⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
      ⋮
```

Data Streams

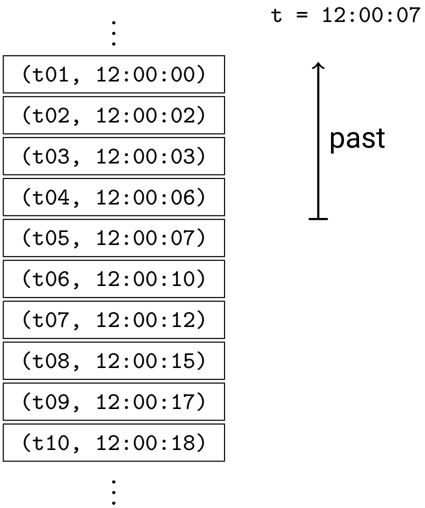
- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time t, tuples with timestamp t'
  - reflect the past if t after t'

```
t = 12:00:07

        ⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
        ⋮
```

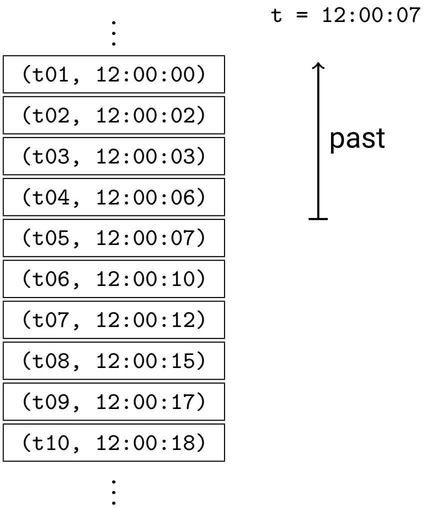
Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time t, tuples with timestamp t'
  - reflect the past if t after t'



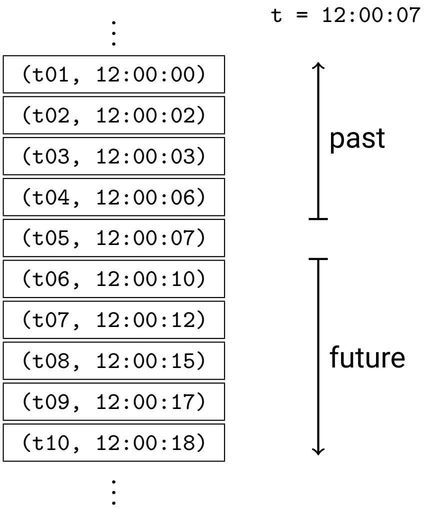
Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time t, tuples with timestamp t'
  - reflect the past if t after t'
  - reflect the (unseen) future if t before t',



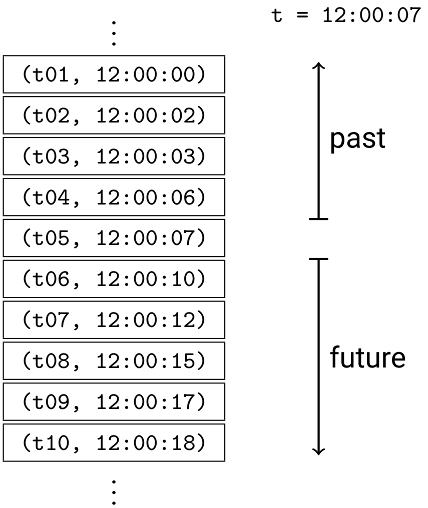
Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time t, tuples with timestamp t'
  - reflect the past if t after t'
  - reflect the (unseen) future if t before t',



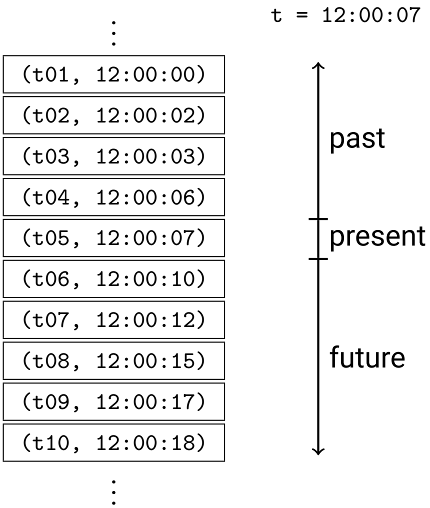
Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time t, tuples with timestamp t'
  - reflect the past if t after t'
  - reflect the (unseen) future if t before t',
  - and reflect the present if $t = t'$



Data Streams

- A *data stream* is a (possibly infinite) sequence of timestamped tuples
- Lots of data produced as streams
  - Sensor values
  - Stock prices
  - User activity
  - GPS coordinates for moving objects
- Ordered by their timestamp
  - Explicit: part of data
  - Implicit: added when entering stream
- At time t, tuples with timestamp t'
  - reflect the past if t after t'
  - reflect the (unseen) future if t before t',
  - and reflect the present if $t = t'$



Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach

⋮

(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)

⋮

Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query

⋮

```
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
```

⋮

Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query
- Data streams updated continuously,

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

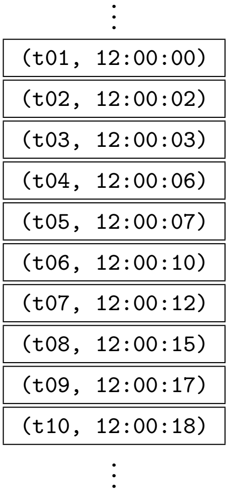
Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query
- Data streams updated continuously,
- at high speed, and

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query
- Data streams updated continuously,
- at high speed, and
- are normally unbounded (always growing),



Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query
- Data streams updated continuously,
- at high speed, and
- are normally unbounded (always growing),
- thus, impossible to store the entire stream

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query
- Data streams updated continuously,
- at high speed, and
- are normally unbounded (always growing),
- thus, impossible to store the entire stream
- or compute answers depending on all tuples

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query
- Data streams updated continuously,
- at high speed, and
- are normally unbounded (always growing),
- thus, impossible to store the entire stream
- or compute answers depending on all tuples
- Data needs to be processed in real-time

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Data Streams vs. Static Data

- With static data, normally use *store-then-query* approach
- All data always available to each query
- Data streams updated continuously,
- at high speed, and
- are normally unbounded (always growing),
- thus, impossible to store the entire stream
- or compute answers depending on all tuples
- Data needs to be processed in real-time
- Two main ways of doing this: *synopses* or *windows*

```
⋮
(t01, 12:00:00)
(t02, 12:00:02)
(t03, 12:00:03)
(t04, 12:00:06)
(t05, 12:00:07)
(t06, 12:00:10)
(t07, 12:00:12)
(t08, 12:00:15)
(t09, 12:00:17)
(t10, 12:00:18)
⋮
```

Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*

```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data

```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.

⋮

(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)

⋮

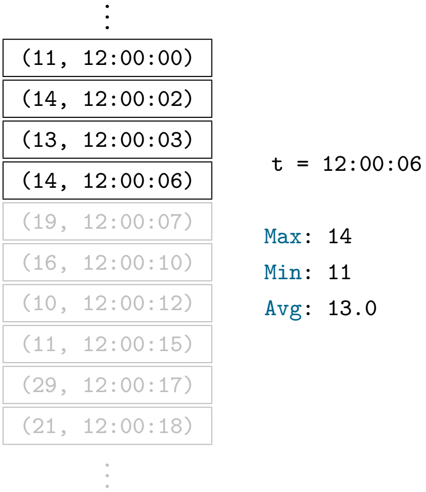
Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen

```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen



Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen

⋮

(11, 12:00:00)

(14, 12:00:02)

(13, 12:00:03)

(14, 12:00:06)

(19, 12:00:07)

(16, 12:00:10)

(10, 12:00:12)

(11, 12:00:15)

(29, 12:00:17)

(21, 12:00:18)

⋮

t = 12:00:07

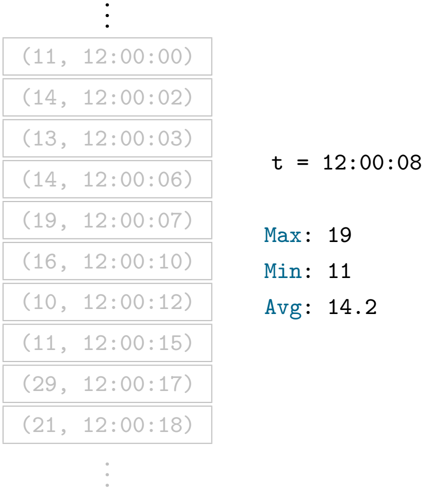
Max: 19

Min: 11

Avg: 14.2

Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen



Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen

⋮

(11, 12:00:00)

(14, 12:00:02)

(13, 12:00:03)

(14, 12:00:06)

(19, 12:00:07)

(16, 12:00:10)

(10, 12:00:12)

(11, 12:00:15)

(29, 12:00:17)

(21, 12:00:18)

⋮

t = 12:00:09

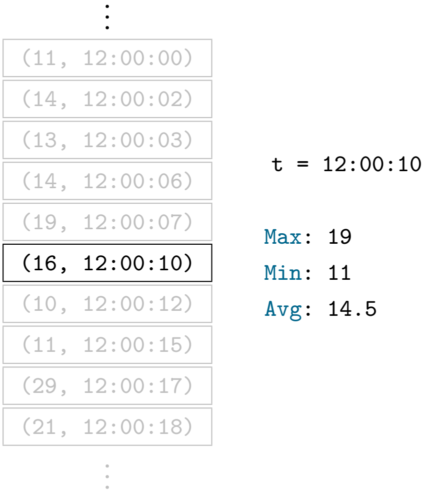
Max: 19

Min: 11

Avg: 14.2

Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen



Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen

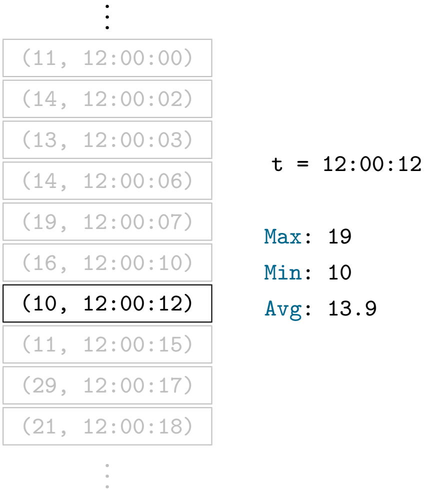
```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

```
t = 12:00:11

Max: 19
Min: 11
Avg: 14.5
```

Synopses

- A *synopsis* is an aggregate or summary of *all data seen until now*
- Is a form of compression of the data
- For instance, keeping track of the total maximum, minimum, average, sum, etc.
- Updated for every new tuple seen



Windows

- A *window* is a finite part of *the data seen so far*

```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window

```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows

```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
  - number of tuples,
  - timestamp,
  - or general predicate

```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
    - number of tuples,
    - timestamp,
    - or general predicate

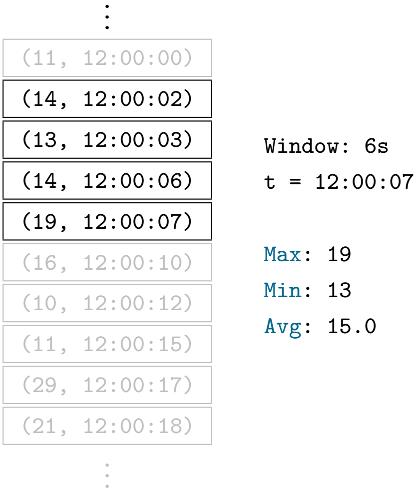
```
⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
⋮
```

```
Window: 6s
t = 12:00:06

Max: 14
Min: 11
Avg: 13.0
```

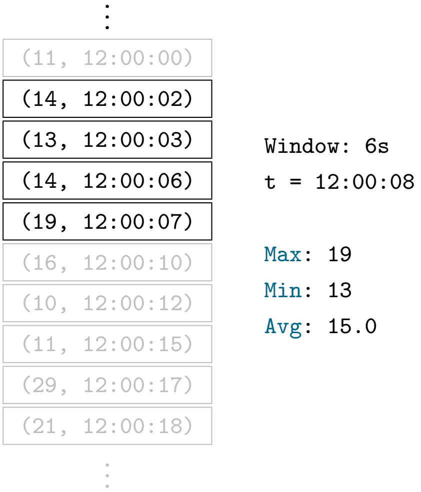
Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
  - number of tuples,
  - timestamp,
  - or general predicate



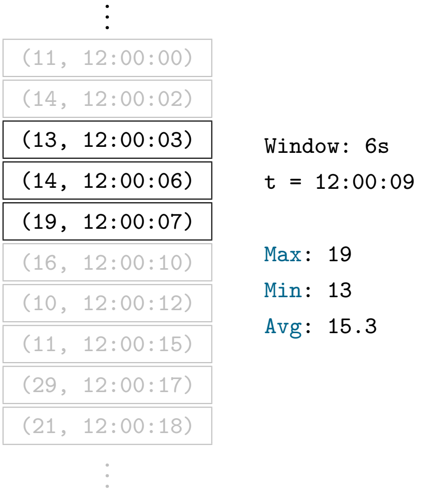
Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
  - number of tuples,
  - timestamp,
  - or general predicate



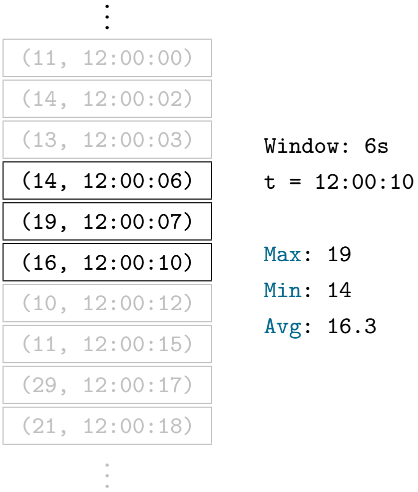
Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
  - number of tuples,
  - timestamp,
  - or general predicate



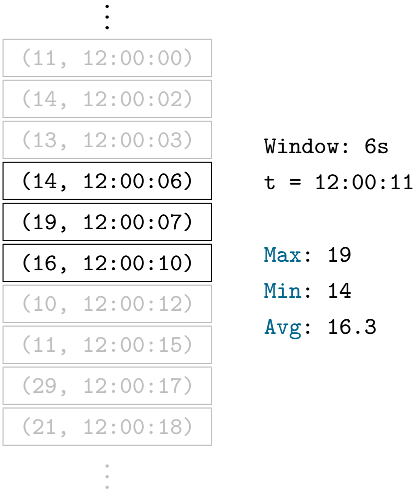
Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
  - number of tuples,
  - timestamp,
  - or general predicate



Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
  - number of tuples,
  - timestamp,
  - or general predicate



Windows

- A *window* is a finite part of *the data seen so far*
- Updated for every window
- Can have sliding or tumbling windows
- Selection can be based on
  - number of tuples,
  - timestamp,
  - or general predicate

```
      ⋮
(11, 12:00:00)
(14, 12:00:02)
(13, 12:00:03)
(14, 12:00:06)
(19, 12:00:07)
(16, 12:00:10)
(10, 12:00:12)
(11, 12:00:15)
(29, 12:00:17)
(21, 12:00:18)
      ⋮
```

```
Window: 6s
t = 12:00:12

Max: 19
Min: 10
Avg: 14.8
```

Data Stream Management Systems

- To extract information from streams, we can either use

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)
  - Data Stream Management Systems (DSMS) (e.g. *PipelineDB*, *TelegraphCQ*, *Aurora/Borealis*)

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)
  - Data Stream Management Systems (DSMS) (e.g. *PipelineDB*, *TelegraphCQ*, *Aurora/Borealis*)
- Benefits of using DSMS:

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)
  - Data Stream Management Systems (DSMS) (e.g. *PipelineDB*, *TelegraphCQ*, *Aurora/Borealis*)
- Benefits of using DSMS:
  - Similar benefits as DBMS (separate data from code, tailored for data management, etc.)

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)
  - Data Stream Management Systems (DSMS) (e.g. *PipelineDB*, *TelegraphCQ*, *Aurora/Borealis*)
- Benefits of using DSMS:
  - Similar benefits as DBMS (separate data from code, tailored for data management, etc.)
  - Declarative query languages

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)
  - Data Stream Management Systems (DSMS) (e.g. *PipelineDB*, *TelegraphCQ*, *Aurora/Borealis*)
- Benefits of using DSMS:
  - Similar benefits as DBMS (separate data from code, tailored for data management, etc.)
  - Declarative query languages
  - Many DSMS based on RDBMS, giving mature foundation

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)
  - Data Stream Management Systems (DSMS) (e.g. *PipelineDB*, *TelegraphCQ*, *Aurora/Borealis*)
- Benefits of using DSMS:
  - Similar benefits as DBMS (separate data from code, tailored for data management, etc.)
  - Declarative query languages
  - Many DSMS based on RDBMS, giving mature foundation
  - Combine continuous and static data in one query

Data Stream Management Systems

- To extract information from streams, we can either use
  - Libraries for programming languages (e.g. *Apache Fink*, *Apache Kafka*)
  - Data Stream Management Systems (DSMS) (e.g. *PipelineDB*, *TelegraphCQ*, *Aurora/Borealis*)
- Benefits of using DSMS:
  - Similar benefits as DBMS (separate data from code, tailored for data management, etc.)
  - Declarative query languages
  - Many DSMS based on RDBMS, giving mature foundation
  - Combine continuous and static data in one query
  - Easy to store interesting parts of stream as traditional relations

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS
- Tuples tagged and ordered by arrival time

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS
- Tuples tagged and ordered by arrival time
- These treat data streams similarly as traditional relations

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS
- Tuples tagged and ordered by arrival time
- These treat data streams similarly as traditional relations
- But, distinguishes between the following four types of relations:

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS
- Tuples tagged and ordered by arrival time
- These treat data streams similarly as traditional relations
- But, distinguishes between the following four types of relations:
  - Static relations (Traditional relation)

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS
- Tuples tagged and ordered by arrival time
- These treat data streams similarly as traditional relations
- But, distinguishes between the following four types of relations:
  - Static relations (Traditional relation)
  - Streams

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS
- Tuples tagged and ordered by arrival time
- These treat data streams similarly as traditional relations
- But, distinguishes between the following four types of relations:
  - Static relations (Traditional relation)
  - Streams
  - Continuous transforms

PipelineDB

- We will use *PipelineDB*, extension to *PostgreSQL*
- Approach similar to many other DSMS
- Tuples tagged and ordered by arrival time
- These treat data streams similarly as traditional relations
- But, distinguishes between the following four types of relations:
  - Static relations (Traditional relation)
  - Streams
  - Continuous transforms
  - Continuous views

Streams

- Data streams come from foreign source

Streams

- Data streams come from foreign source

```
CREATE FOREIGN TABLE sensors (
    sid int,
    temp float,
    wind float,
    humidity float
)
SERVER pipelinedb;
```

Streams

- Data streams come from foreign source
- Continuous transforms can alter streams

```
CREATE FOREIGN TABLE sensors (
    sid int,
    temp float,
    wind float,
    humidity float
)
SERVER pipelinedb;
```

Streams

- Data streams come from foreign source
- Continuous transforms can alter streams

```
CREATE FOREIGN TABLE sensors (
    sid int,
    temp float,
    wind float,
    humidity float
)
SERVER pipelinedb;

CREATE VIEW temps
WITH (action=transform) AS
SELECT sid,
       (temp - 32)/1.8 AS tempC
FROM sensors
```

Streams

- Data streams come from foreign source
- Continuous transforms can alter streams
- Resulting stream accessed with `output_of('temps')`

```
CREATE FOREIGN TABLE sensors (
    sid int,
    temp float,
    wind float,
    humidity float
)
SERVER pipelinedb;

CREATE VIEW temps
WITH (action=transform) AS
SELECT sid,
       (temp - 32)/1.8 AS tempC
FROM sensors
```

Streams

- Data streams come from foreign source
- Continuous transforms can alter streams
- Resulting stream accessed with `output_of('temps')`
- No tuples stored for either

```
CREATE FOREIGN TABLE sensors (
    sid int,
    temp float,
    wind float,
    humidity float
)
SERVER pipelinedb;

CREATE VIEW temps
WITH (action=transform) AS
SELECT sid,
       (temp - 32)/1.8 AS tempC
FROM sensors
```

Streams

- Data streams come from foreign source
- Continuous transforms can alter streams
- Resulting stream accessed with `output_of('temps')`
- No tuples stored for either
- Must be read by continuous views

```
CREATE FOREIGN TABLE sensors (
    sid int,
    temp float,
    wind float,
    humidity float
)
SERVER pipelinedb;

CREATE VIEW temps
WITH (action=transform) AS
SELECT sid,
       (temp - 32)/1.8 AS tempC
FROM sensors
```

Streams

- Data streams come from foreign source
- Continuous transforms can alter streams
- Resulting stream accessed with `output_of('temps')`
- No tuples stored for either
- Must be read by continuous views
- Push based

```
CREATE FOREIGN TABLE sensors (
    sid int,
    temp float,
    wind float,
    humidity float
)
SERVER pipelinedb;

CREATE VIEW temps
WITH (action=transform) AS
SELECT sid,
       (temp - 32)/1.8 AS tempC
FROM sensors
```

Continuous views

- Sometimes called time-varying relation

Continuous views

- Sometimes called time-varying relation
- Finite relations, but varies with time

Continuous views

- Sometimes called time-varying relation
- Finite relations, but varies with time
- Views constructed as queries over streams

Continuous views

- Sometimes called time-varying relation
- Finite relations, but varies with time
- Views constructed as queries over streams

```
CREATE VIEW strongwind AS
SELECT sid, humidity, temp
FROM sensors
WHERE wind > 40
```

Continuous views

- Sometimes called time-varying relation
- Finite relations, but varies with time
- Views constructed as queries over streams
- or over transforms (using `output_of`)

```
CREATE VIEW strongwind AS
SELECT sid, humidity, temp
FROM sensors
WHERE wind > 40
```

Continuous views

- Sometimes called time-varying relation
- Finite relations, but varies with time
- Views constructed as queries over streams
- or over transforms (using `output_of`)

```
CREATE VIEW strongwind AS
SELECT sid, humidity, temp
FROM sensors
WHERE wind > 40

CREATE VIEW hightemps AS
SELECT sid, tempC
FROM output_of('temps')
WHERE tempC > 50
```

Continuous views

- Sometimes called time-varying relation
- Finite relations, but varies with time
- Views constructed as queries over streams
- or over transforms (using `output_of`)
- Materialized

```
CREATE VIEW strongwind AS
SELECT sid, humidity, temp
FROM sensors
WHERE wind > 40

CREATE VIEW hightemps AS
SELECT sid, tempC
FROM output_of('temps')
WHERE tempC > 50
```

Continuous views

- Sometimes called time-varying relation
- Finite relations, but varies with time
- Views constructed as queries over streams
- or over transforms (using `output_of`)
- Materialized
- Can do synopsis or window-based

```
CREATE VIEW strongwind AS
SELECT sid, humidity, temp
FROM sensors
WHERE wind > 40

CREATE VIEW hightemps AS
SELECT sid, tempC
FROM output_of('temps')
WHERE tempC > 50
```

Stream Queries: Synopses

- A synopsis is made by aggregation

`output_of('temps')`

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate

`output_of('temps')`

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally

```
output_of('temps')
⋮
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
⋮
```

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

```
output_of('temps')
        ⋮
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
        ⋮
```

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

```
output_of('temps')
⋮
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
⋮
```

```
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

```
output_of('temps')
⋮
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
⋮
```

```
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:06)

sid	mn	mx	ag
0	11	11	11.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

output_of('temps')

⋮

```
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
```

⋮

```sql
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:07)

sid	mn	mx	ag
0	11	19	15.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

output_of('temps')

⋮

```
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
```

⋮

```sql
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:08)

sid	mn	mx	ag
0	11	19	15.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```sql
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:09)

sid	mn	mx	ag
0	11	19	15.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```sql
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:10)

sid	mn	mx	ag
0	11	19	15.0
1	13	16	14.3
2	14	14	14.0

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:11)

sid	mn	mx	ag
0	11	19	15.0
1	13	16	14.3
2	14	14	14.0

Stream Queries: Synopses

- A synopsis is made by aggregation
- Can use any common SQL aggregate
- Computed incrementally
- Result is a (materialized) view with typically few rows

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```sql
CREATE VIEW tempagg AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:12)

sid	mn	mx	ag
0	11	19	15.0
1	13	17	15.0
2	14	14	14.0

Stream Queries: Windows

- A window-query is evaluated over windows

`output_of('temps')`

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation

`output_of('temps')`

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates

`output_of('temps')`

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

```
output_of('temps')
⋮
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
⋮
```

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

```
output_of('temps')
⋮
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
⋮
```

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:06)

sid	mn	mx	ag
0	11	11	11.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:07)

sid	mn	mx	ag
0	19	19	19.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:08)

sid	mn	mx	ag
0	19	19	19.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:09)

sid	mn	mx	ag
0	19	19	19.0
1	13	14	13.5
2	14	14	14.0

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:10)

sid	mn	mx	ag
0	19	19	19.0
1	16	16	16.0
2	14	14	14.0

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

```
output_of('temps')
⋮
(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)
⋮
```

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:11)

sid	mn	mx	ag
0	19	19	19.0
1	16	16	16.0
2	14	14	14.0

Stream Queries: Windows

- A window-query is evaluated over windows
- Each window is treated as a regular relation
- Results updates whenever window updates
- Computed incrementally based on difference

output_of('temps')

⋮

(0, 11, 12:00:00)
(1, 14, 12:00:02)
(1, 13, 12:00:03)
(2, 14, 12:00:06)
(0, 19, 12:00:07)
(1, 16, 12:00:10)
(1, 17, 12:00:12)
(0, 11, 12:00:15)
(2, 29, 12:00:17)
(2, 21, 12:00:18)

⋮

```
CREATE VIEW tempagg
WITH (sw='6 seconds') AS
SELECT
  sid,
  min(tempC) AS mn,
  max(tempC) AS mx,
  avg(tempC) AS ag
FROM output_of('temps')
GROUP BY sid
```

tempagg (t=12:00:12)

sid	mn	mx	ag
0	19	19	19.0
1	16	17	16.5
2	14	14	14.0

Joins

- Streams and continuous views can be joined

Joins

- Streams and continuous views can be joined
- both with other streams and continuous views

Joins

- Streams and continuous views can be joined
- both with other streams and continuous views
- and with static relations

Joins

- Streams and continuous views can be joined
- both with other streams and continuous views
- and with static relations
- But, restrictions apply

Spatial Streams and Views

- Streams or continuous views can either:

Spatial Streams and Views

- Streams or continuous views can either:
  - be joined with static spatial data,

Spatial Streams and Views

- Streams or continuous views can either:
  - be joined with static spatial data,
  - contain spatial data directly,

Spatial Streams and Views

- Streams or continuous views can either:
  - be joined with static spatial data,
  - contain spatial data directly,
  - or be interpreted as spatial objects

Spatial Streams and Views

- Streams or continuous views can either:
  - be joined with static spatial data,
  - contain spatial data directly,
  - or be interpreted as spatial objects

  to become a spatial data stream or a continuous view

Spatial Queries over Streams

- Spatial streams and continuous views can be queried like any other stream or view

Spatial Queries over Streams

- Spatial streams and continuous views can be queried like any other stream or view
- Can filter, transform, and derive spatial data with spatial predicates and functions

Spatial Queries over Streams

- Spatial streams and continuous views can be queried like any other stream or view
- Can filter, transform, and derive spatial data with spatial predicates and functions
- Can use spatial aggregates to form complex spatial objects from simpler objects

Spatial Queries over Streams

- Spatial streams and continuous views can be queried like any other stream or view
- Can filter, transform, and derive spatial data with spatial predicates and functions
- Can use spatial aggregates to form complex spatial objects from simpler objects
- *PipelineDB* supports all functions, predicate and aggregates from *PostGIS*

Use case 1: Storms

- Stream of weather data from sensors

`sensors(sid, wind, temp, humid, time)`

⋮

0	11	10	61	12:00:00
1	4	12	82	12:00:02
4	13	14	74	12:00:03
22	2	19	53	12:00:06
8	19	21	60	12:00:07
7	6	11	63	12:00:10
17	2	18	59	12:00:12
9	11	11	71	12:00:15
24	29	13	84	12:00:17

⋮

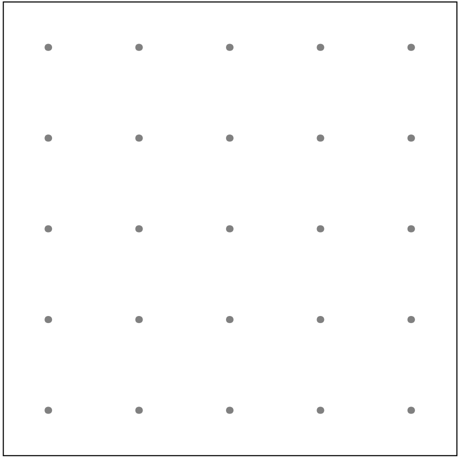
Use case 1: Storms

- Stream of weather data from sensors
- Each sensor has a (static) location associated with it

```
sensors(sid, wind, temp, humid, time)
                  ⋮
( 0, 11, 10, 61, 12:00:00)
( 1,  4, 12, 82, 12:00:02)
( 4, 13, 14, 74, 12:00:03)
(22,  2, 19, 53, 12:00:06)
( 8, 19, 21, 60, 12:00:07)
( 7,  6, 11, 63, 12:00:10)
(17,  2, 18, 59, 12:00:12)
( 9, 11, 11, 71, 12:00:15)
(24, 29, 13, 84, 12:00:17)
                  ⋮
```

Use case 1: Storms

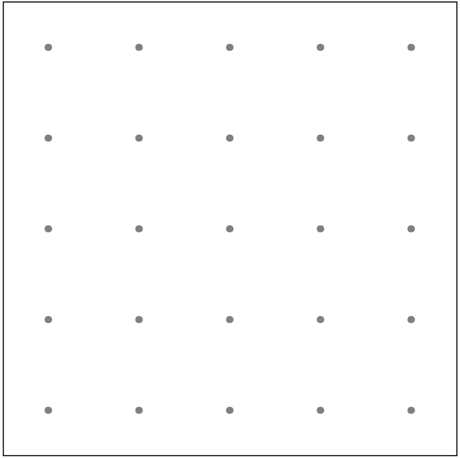
- Stream of weather data from sensors
- Each sensor has a (static) location associated with it



Use case 1: Storms

- Stream of weather data from sensors
- Each sensor has a (static) location associated with it
- Join the stream with static table to get location of wind speeds:

```
CREATE VIEW wind_loc
WITH (action=transform) AS
SELECT sid, l.location, s.wind
FROM sensors AS s, sensor_loc AS l
WHERE s.sid = l.sid
```



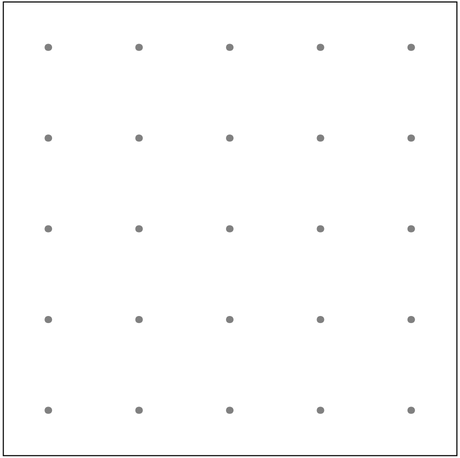
Use case 1: Storms

- Stream of weather data from sensors
- Each sensor has a (static) location associated with it
- Join the stream with static table to get location of wind speeds:

```
CREATE VIEW wind_loc
WITH (action=transform) AS
SELECT sid, l.location, s.wind
FROM sensors AS s, sensor_loc AS l
WHERE s.sid = l.sid
```

- Use average over sliding window to get current picture

```
CREATE VIEW wind_avg
WITH (sw='1 minute') AS
SELECT l.location AS loc,
       avg(s.wind) AS wind
FROM output_of('obs_loc')
GROUP BY sid
```



Use case 1: Storms

- Stream of weather data from sensors
- Each sensor has a (static) location associated with it
- Join the stream with static table to get location of wind speeds:

```
CREATE VIEW wind_loc
WITH (action=transform) AS
SELECT sid, l.location, s.wind
FROM sensors AS s, sensor_loc AS l
WHERE s.sid = l.sid
```

- Use average over sliding window to get current picture

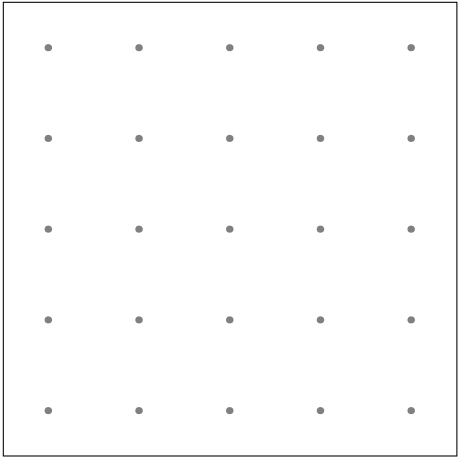
```
CREATE VIEW wind_avg
WITH (sw='1 minute') AS
SELECT l.location AS loc,
       avg(s.wind) AS wind
FROM output_of('obs_loc')
GROUP BY sid
```

wind_avg

location	wind
Point(11,59)	11.7
Point(12,60)	4.6
Point(13,59)	13.1
Point(10,61)	2.0
Point(12,59)	19.7
Point(11,61)	6.1
Point(13,60)	2.2
Point(10,58)	11.8
Point(11,60)	29.9
⋮	⋮

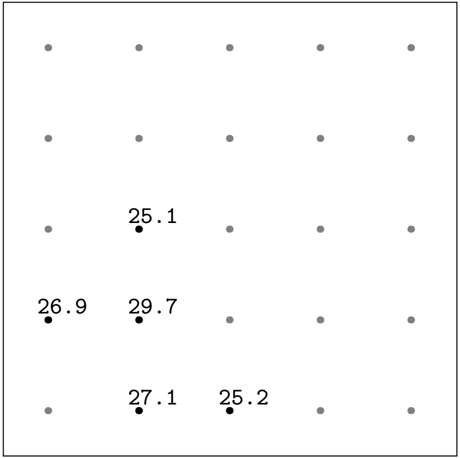
Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)



Use case 1: Detecting storms

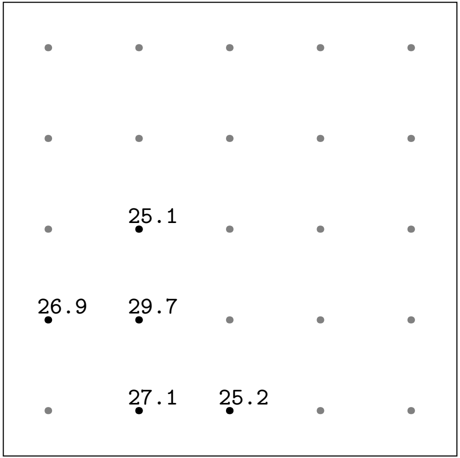
- Might now have multiple sensors forming a storm (`wind > 25`)



Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)

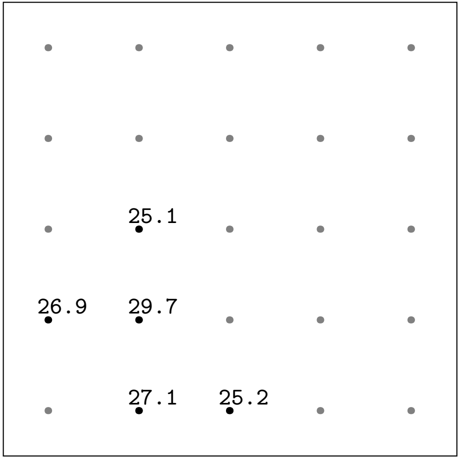
```
CREATE VIEW storm AS
SELECT loc
FROM wind_avg
WHERE wind > 25
```



Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)
- However, a storm has a spatial extent

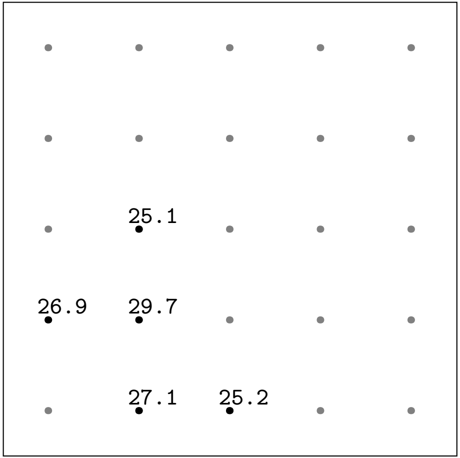
```
CREATE VIEW storm AS
SELECT loc
FROM wind_avg
WHERE wind > 25
```



Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)
- However, a storm has a spatial extent
- Let the storm equal convex hull of locations of sensors observing storm

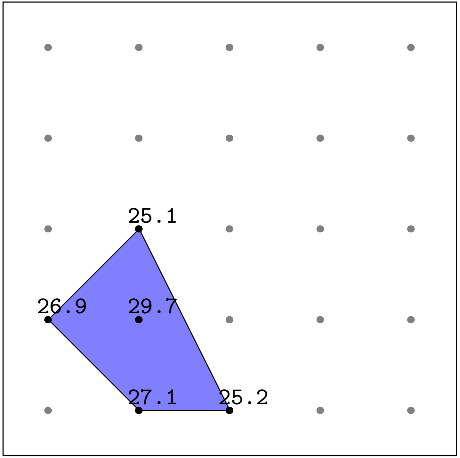
```
CREATE VIEW storm AS
SELECT loc
FROM wind_avg
WHERE wind > 25
```



Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)
- However, a storm has a spatial extent
- Let the storm equal convex hull of locations of sensors observing storm

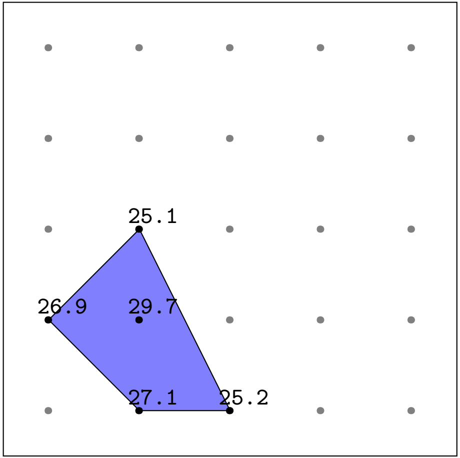
```
CREATE VIEW storm AS
SELECT loc
FROM wind_avg
WHERE wind > 25
```



Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)
- However, a storm has a spatial extent
- Let the storm equal convex hull of locations of sensors observing storm

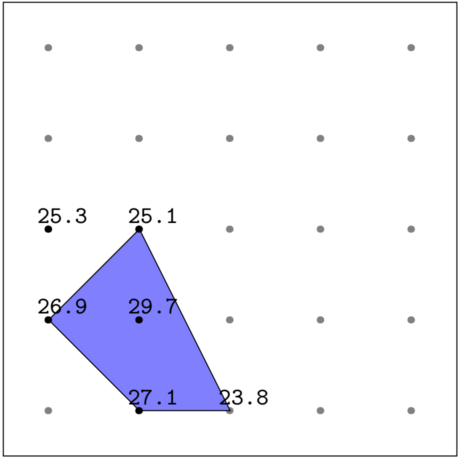
```
CREATE VIEW storm AS
SELECT ST_ConvexHull(loc) AS extent
FROM wind_avg
WHERE wind > 25
```



Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)
- However, a storm has a spatial extent
- Let the storm equal convex hull of locations of sensors observing storm

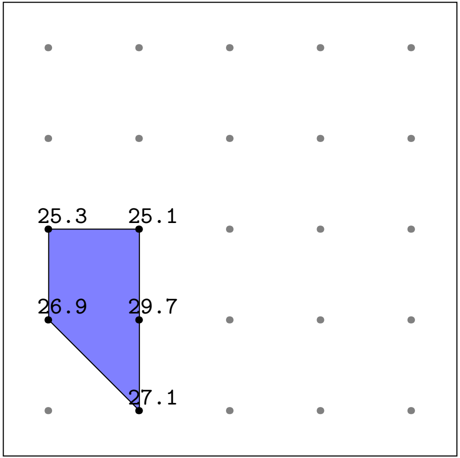
```
CREATE VIEW storm AS
SELECT ST_ConvexHull(loc) AS extent
FROM wind_avg
WHERE wind > 25
```



Use case 1: Detecting storms

- Might now have multiple sensors forming a storm (`wind > 25`)
- However, a storm has a spatial extent
- Let the storm equal convex hull of locations of sensors observing storm

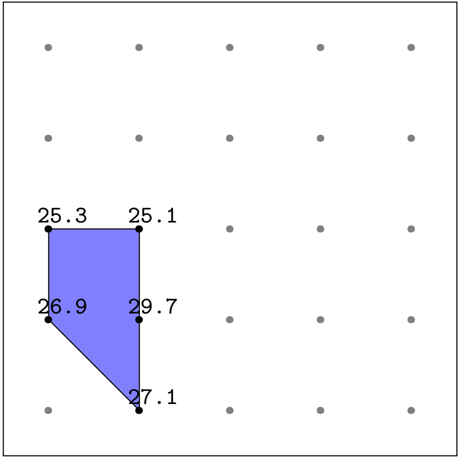
```
CREATE VIEW storm AS
SELECT ST_ConvexHull(loc) AS extent
FROM wind_avg
WHERE wind > 25
```



Use case 1: Complication – Multiple storms

- Can have more than one storm

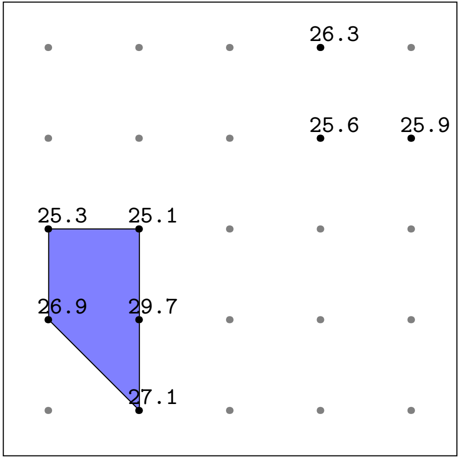
```
CREATE VIEW storm AS
SELECT ST_ConvexHull(loc) AS extent
FROM wind_avg
WHERE wind > 25
```



Use case 1: Complication – Multiple storms

- Can have more than one storm

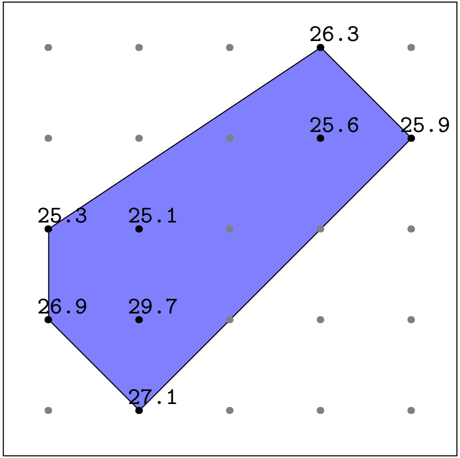
```
CREATE VIEW storm AS
SELECT ST_ConvexHull(loc) AS extent
FROM wind_avg
WHERE wind > 25
```



Use case 1: Complication – Multiple storms

- Can have more than one storm
- Current approach fails

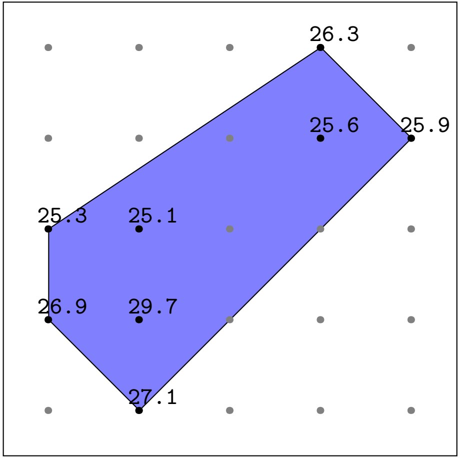
```
CREATE VIEW storm AS
SELECT ST_ConvexHull(loc) AS extent
FROM wind_avg
WHERE wind > 25
```



Use case 1: Complication – Multiple storms

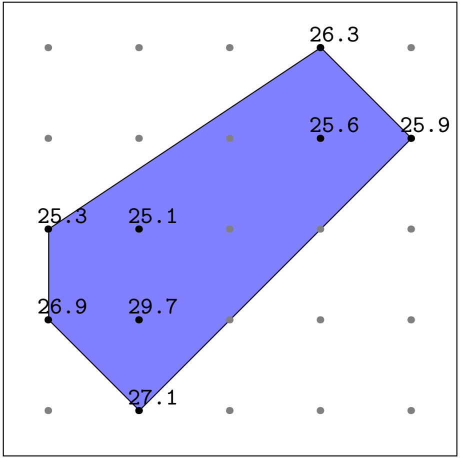
- Can have more than one storm
- Current approach fails
- Use clustering to group close points together

```
CREATE VIEW storm AS
SELECT ST_ConvexHull(loc) AS extent
FROM wind_avg
WHERE wind > 25
```



Use case 1: Complication – Multiple storms

- Can have more than one storm
- Current approach fails
- Use clustering to group close points together

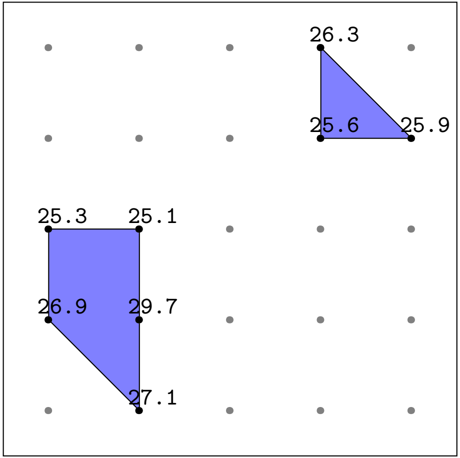



```sql
CREATE VIEW storms AS
SELECT ST_ConvexHull(c.cl) AS extent
FROM (
 SELECT
  unnest(ST_ClusterWithin(loc, 10000)) AS cl
 FROM wind_avg
 WHERE wind > 25
) AS c
```

Use case 1: Complication – Multiple storms

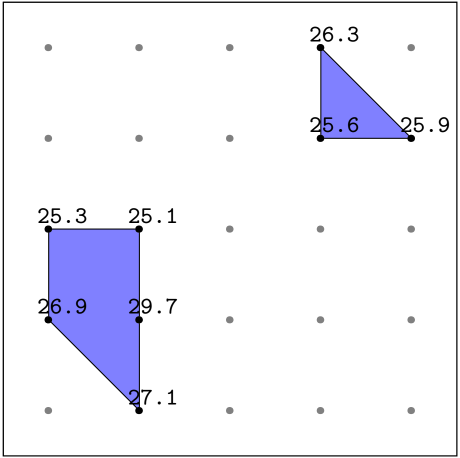
- Can have more than one storm
- Current approach fails
- Use clustering to group close points together

```
CREATE VIEW storms AS
SELECT ST_ConvexHull(c.cl) AS extent
FROM (
 SELECT
  unnest(ST_ClusterWithin(loc, 10000)) AS cl
 FROM wind_avg
 WHERE wind > 25
) AS c
```



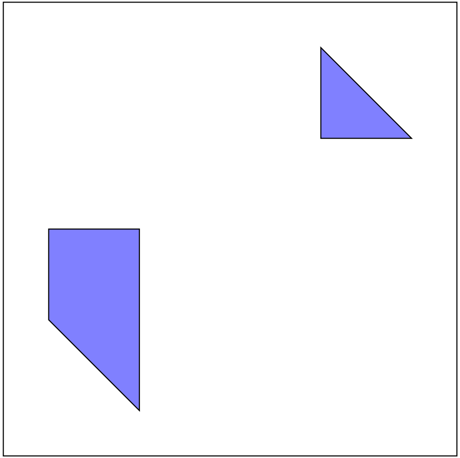
Use case 1: Creating a storm warning system

- Can now use the storms as spatial objects without thinking of sensors



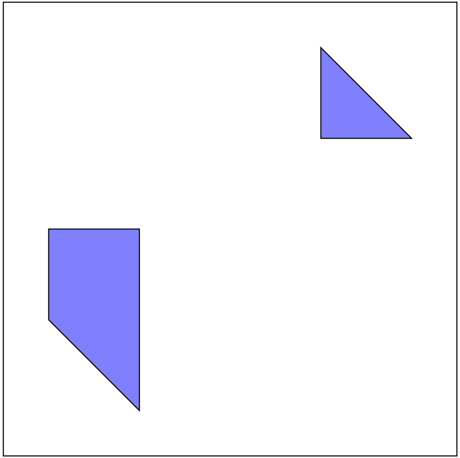
Use case 1: Creating a storm warning system

- Can now use the storms as spatial objects without thinking of sensors



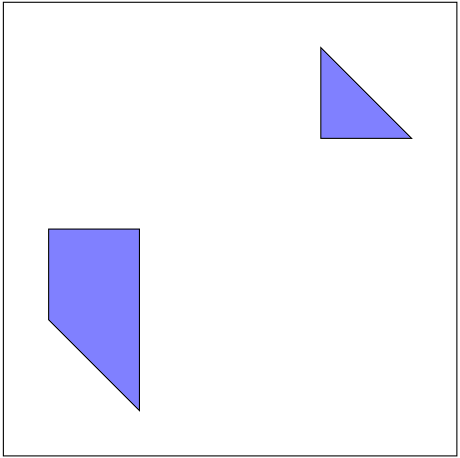
Use case 1: Creating a storm warning system

- Can now use the storms as spatial objects without thinking of sensors
- For example, make a storm warning system



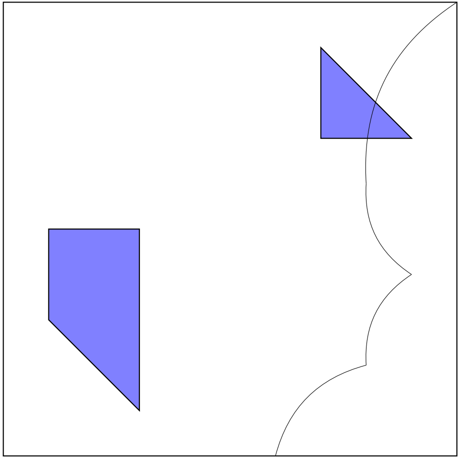
Use case 1: Creating a storm warning system

- Can now use the storms as spatial objects without thinking of sensors
- For example, make a storm warning system
- Given a table `sensitive(name, extent)` of storm sensitive objects



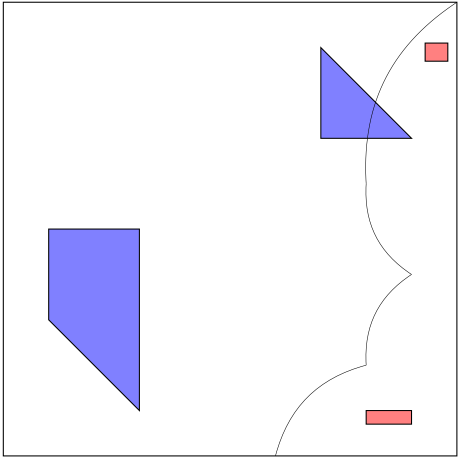
Use case 1: Creating a storm warning system

- Can now use the storms as spatial objects without thinking of sensors
- For example, make a storm warning system
- Given a table `sensitive(name, extent)` of storm sensitive objects



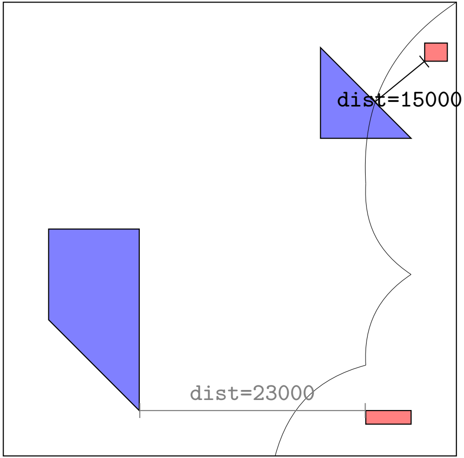
Use case 1: Creating a storm warning system

- Can now use the storms as spatial objects without thinking of sensors
- For example, make a storm warning system
- Given a table `sensitive(name, extent)` of storm sensitive objects



Use case 1: Creating a storm warning system

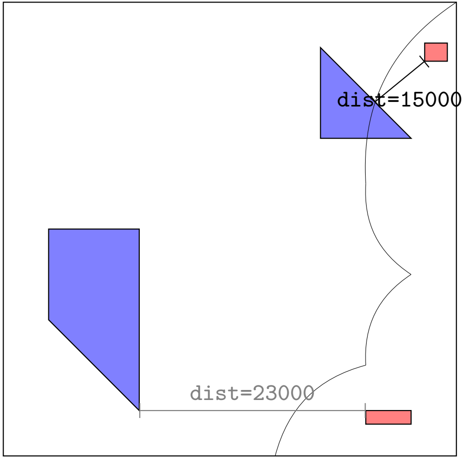
- Can now use the storms as spatial objects without thinking of sensors
- For example, make a storm warning system
- Given a table `sensitive(name, extent)` of storm sensitive objects
- Want to give warning if storm distance `< 20000`



Use case 1: Creating a storm warning system

- Can now use the storms as spatial objects without thinking of sensors
- For example, make a storm warning system
- Given a table `sensitive(name, extent)` of storm sensitive objects
- Want to give warning if storm distance < 20000

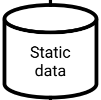
```
CREATE VIEW warnings AS
SELECT
  sen.name,
  min(ST_Distance(sen.extent, sto.extent)) AS dist
FROM storms AS sto, sensitive AS sen
GROUP BY sen.name
HAVING dist < 20000
```



Use case 1: Pipeline

```
CREATE VIEW wind_loc
WITH (action=transform) AS
SELECT sid, l.location, s.wind
FROM sensors AS s, sensor_loc AS l
WHERE s.sid = l.sid
```

```
CREATE VIEW wind_avg
WITH (sw='1 minute') AS
SELECT l.location AS loc,
       avg(s.wind) AS wind
FROM output_of('obs_loc')
GROUP BY sid
```

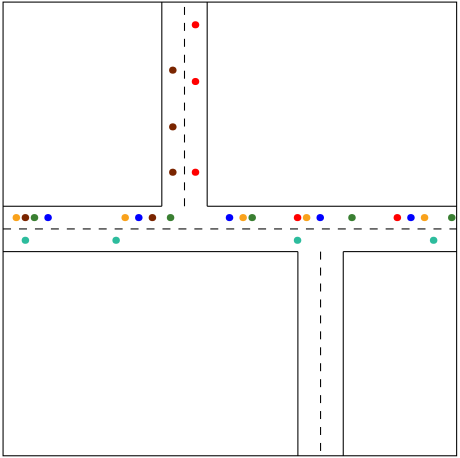



```
CREATE VIEW storms AS
SELECT ST_ConvexHull(c.cl) AS extent
FROM (
 SELECT
  unnest(ST_ClusterWithin(loc, 10000)) AS cl
 FROM wind_avg
 WHERE wind > 25
) AS c
```

```
CREATE VIEW warnings AS
SELECT
  sen.name,
  min(ST_Distance(sen.extent, sto.extent)) AS dist
FROM storms AS sto, sensitive AS sen
GROUP BY sen.name
HAVING dist < 20000
```

Use case 2: Traffic

- Stream of vehicle locations based on GPS, as strings



Use case 2: Traffic

- Stream of vehicle locations based on GPS, as strings

`gps_stream`

⋮

''0 12:00:00 11 59''
''1 12:00:02 12 58''
''2 12:00:06 14 60''
''1 12:00:03 13 57''
''0 12:00:07 11 59''
''1 12:00:10 12 58''
''0 12:00:15 11 59''
''1 12:00:12 13 61''
''2 12:00:17 14 62''
''2 12:00:18 11 61''

⋮

Use case 2: Traffic

- Stream of vehicle locations based on GPS, as strings
- Use query to parse location to spatial entities

gps_stream

```
⋮
''0 12:00:00 11 59''
''1 12:00:02 12 58''
''2 12:00:06 14 60''
''1 12:00:03 13 57''
''0 12:00:07 11 59''
''1 12:00:10 12 58''
''0 12:00:15 11 59''
''1 12:00:12 13 61''
''2 12:00:17 14 62''
''2 12:00:18 11 61''
⋮
```

Use case 2: Traffic

- Stream of vehicle locations based on GPS, as strings
- Use query to parse location to spatial entities

```
CREATE VIEW vlocations
WITH (action=transform) AS
SELECT
  CAST(raw[0] AS integer) AS vid,
  CAST(raw[1] AS datetime) AS gps_time,
  ST_MakePoint(CAST(raw[2] AS double),
               CAST(raw[3] AS double)) AS loc
FROM (SELECT regexp_split_to_array(data, ' ')
      FROM gps_stream) AS t(raw)
```

gps_stream

⋮

''0 12:00:00 11 59''
''1 12:00:02 12 58''
''2 12:00:06 14 60''
''1 12:00:03 13 57''
''0 12:00:07 11 59''
''1 12:00:10 12 58''
''0 12:00:15 11 59''
''1 12:00:12 13 61''
''2 12:00:17 14 62''
''2 12:00:18 11 61''

⋮

Use case 2: Traffic

- Stream of vehicle locations based on GPS, as strings
- Use query to parse location to spatial entities

```
CREATE VIEW vlocations
WITH (action=transform) AS
SELECT
  CAST(raw[0] AS integer) AS vid,
  CAST(raw[1] AS datetime) AS gps_time,
  ST_MakePoint(CAST(raw[2] AS double),
               CAST(raw[3] AS double)) AS loc
FROM (SELECT regexp_split_to_array(data, ' ')
      FROM gps_stream) AS t(raw)
```

vlocations

⋮

(0, 12:00:00, Point(11, 59))
(1, 12:00:02, Point(12, 58))
(2, 12:00:06, Point(14, 60))
(1, 12:00:03, Point(13, 57))
(0, 12:00:07, Point(11, 59))
(1, 12:00:10, Point(12, 58))
(0, 12:00:15, Point(11, 59))
(1, 12:00:12, Point(13, 61))
(2, 12:00:17, Point(14, 62))
(2, 12:00:18, Point(11, 61))

⋮

Use case 2: Traffic

- Stream of vehicle locations based on GPS, as strings
- Use query to parse location to spatial entities
- Could here also remove noise and other preprocessing steps

```
CREATE VIEW vlocations
WITH (action=transform) AS
SELECT
  CAST(raw[0] AS integer) AS vid,
  CAST(raw[1] AS datetime) AS gps_time,
  ST_MakePoint(CAST(raw[2] AS double),
               CAST(raw[3] AS double)) AS loc
FROM (SELECT regexp_split_to_array(data, ' ')
      FROM gps_stream) AS t(raw)
```

vlocations

⋮

(0, 12:00:00, Point(11, 59))
(1, 12:00:02, Point(12, 58))
(2, 12:00:06, Point(14, 60))
(1, 12:00:03, Point(13, 57))
(0, 12:00:07, Point(11, 59))
(1, 12:00:10, Point(12, 58))
(0, 12:00:15, Point(11, 59))
(1, 12:00:12, Point(13, 61))
(2, 12:00:17, Point(14, 62))
(2, 12:00:18, Point(11, 61))

⋮

Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end,
FROM output_of('vlocations')
GROUP BY vid
```

Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```

Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```

```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```

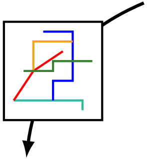
Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```

```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```

Use case 2: Traffic analysis

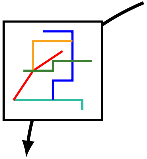
```sql
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```



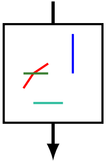
```sql
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```

Use case 2: Traffic analysis

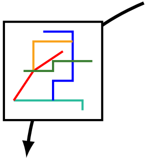
```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```



```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```

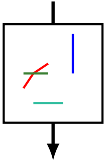


Use case 2: Traffic analysis



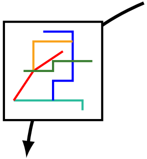
```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```

```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```



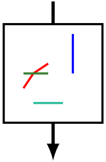
```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

Use case 2: Traffic analysis



```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```

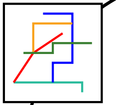
```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```



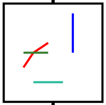
```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

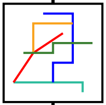
Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```



```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```

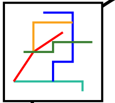


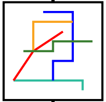


```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

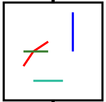
Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```





```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```

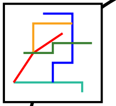


```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

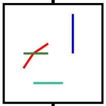
vid	speed
2	60
5	80

Use case 2: Traffic analysis

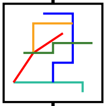
```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```



```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```



```
CREATE VIEW tolls AS
SELECT p.vid,
       t.cost * count(p.vid) AS due
FROM paths AS p, tolls AS t
WHERE ST_contains(p.path, t.loc)
GROUP BY p.vid, t.cost
```

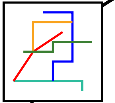


```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

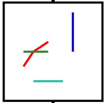
vid	speed
2	60
5	80

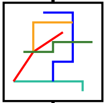
Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```



```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```





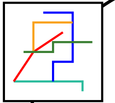
```
CREATE VIEW tolls AS
SELECT p.vid,
       t.cost * count(p.vid) AS due
FROM paths AS p, tolls AS t
WHERE ST_contains(p.path, t.loc)
GROUP BY p.vid, t.cost
```

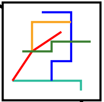
```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

vid	speed
2	60
5	80

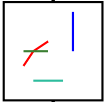
Use case 2: Traffic analysis

```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```

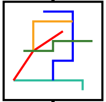




```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```



```
CREATE VIEW tolls AS
SELECT p.vid,
       t.cost * count(p.vid) AS due
FROM paths AS p, tolls AS t
WHERE ST_contains(p.path, t.loc)
GROUP BY p.vid, t.cost
```

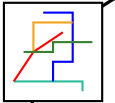


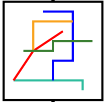
```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

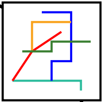
vid	speed
2	60
5	80

Use case 2: Traffic analysis

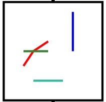
```
CREATE VIEW paths WITH (sw='1 minute') AS
SELECT vid,
       ST_MakeLine(loc ORDER BY gps_time) AS path,
       min(gps_time) AS start,
       max(gps_time) AS end
FROM vlocations
GROUP BY vid
```







```
CREATE VIEW jams AS
SELECT unnest(ST_ClusterIntersect(path))
FROM paths
WHERE ST_Length(path) /
        seconds(end - start) < 2
```



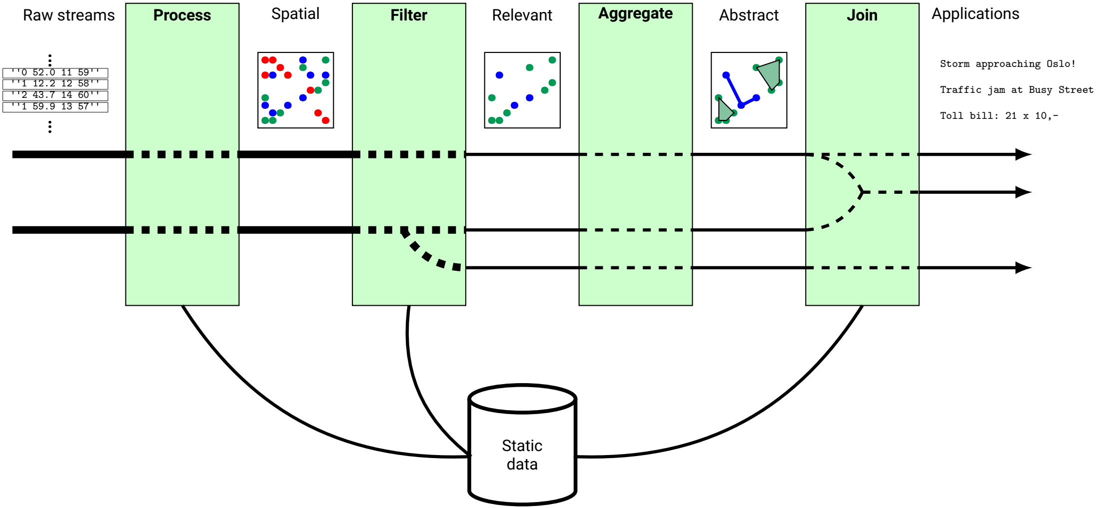
```
CREATE VIEW speedings AS
SELECT p.vid, s.speed
FROM paths AS p, speedlimits AS s
WHERE ST_contains(s.extent, p.path) AND
      ST_Length(p.path) /
        seconds(p.end - p.start) > s.speed
```

vid	speed
2	60
5	80

```
CREATE VIEW tolls AS
SELECT p.vid,
       t.cost * count(p.vid) AS due
FROM paths AS p, tolls AS t
WHERE ST_contains(p.path, t.loc)
GROUP BY p.vid, t.cost
```

vid	due
3	6
7	4
9	10

Spatial information management using continuous queries



Summary

- Queries over data streams allow us to parse and build abstractions over the raw data streams

Summary

- Queries over data streams allow us to parse and build abstractions over the raw data streams
- Different applications can use different abstraction levels

Summary

- Queries over data streams allow us to parse and build abstractions over the raw data streams
- Different applications can use different abstraction levels
- Windows takes care of keeping data fresh and relevant

Summary

- Queries over data streams allow us to parse and build abstractions over the raw data streams
- Different applications can use different abstraction levels
- Windows takes care of keeping data fresh and relevant
- Can still store longer history of abstract objects (e.g. storms)

More information:

- *Data Stream Management*, Minos Garofalakis, Johannes Gehrke, Rajeev Rastogi (Editors), Springer 2016
- `docs.pipelinedb.org`
- *Spatio-Temporal Data Streams*, Zdravko Galić, Springer 2016

Thank you for listening!

Actual queries: single storm

```
CREATE VIEW storm
WITH (action=materialize, sw='30 seconds') AS
SELECT ST_ConvexHull(ST_Collect(loc)) AS location
FROM locations
WHERE wind > 20
```

Actual queries: jams

```
CREATE VIEW jams AS
SELECT ST_LineMerge(ST_SnapToGrid(c.cluster, 0.0001))
FROM (SELECT unnest(ST_ClusterIntersect(path)) AS cluster
      FROM paths
      WHERE ST_Length(path) /
            extract('epoch' from (end - start)::interval)
            < 3 -- ft/s
     ) AS c
```