Streaming Data Analytics

Professor: Emanuelle Della Valle

Last updated: October 10, 2024

Contents

Chapter 1

Introduction

In streaming data engineering, continuity of data analysis is a key concept. To process large amounts of data, we have two main approaches:

- Batch processing: this is the traditional approach where data is collected and stored, then processed in a batch to obtain insights.

- Stream processing: in this approach, data is processed as it arrives, allowing for real-time insights.

In some cases, batch processing is enough, like when we need to process data that is not time-sensitive, such as historical data. However, in many cases, we need to process data in real-time, like in fraud detection or monitoring systems like fire alarms. In these cases, stream processing is the way to go. But, how do we enable this continuity of data flow?

1.1 Enabling Continuity

To enable continuity of data processing, we need four main components:

- Sensors and actuators

- Connectivity

- Streaming data engineering

- Streaming data science

1.1.1 Sensors and Actuators

Sensors are devices that collect data from the environment, like temperature sensors, cameras, etc. Actuators are devices that can act on the environment, like turning on a light, opening a door, etc. These devices are the first step in the data processing pipeline. They are a way of giving the computers the means to gather data from the environment and act on it.

[...] sensor technology enable computers to observe, identify and understand the world - without the limitations of human-entered data. - **Kevin Ashton, the brander of "Internet of Things"**

1.1.2 Connectivity

This is the enabling and constraninig factor of the Internet of Things. It is the way that sensors and actuators communicate with the rest of the system. This can be done through a variety of means, like WiFi, Bluetooth, etc.

In this case, the distance and the use case are the main factors that determine the type of connectivity that we need. For example, if we need to connect a sensor to a computer that is far away, we might need to use a cellular network, while if we need to connect a sensor to a computer that is close, we might use Bluetooth.

Also, bandwidth is a factor that we need to consider. If we need to send a lot of data, we might need a high bandwidth connection, while if we need to send a small amount of data, we might use a low bandwidth connection. Another important factor is the latency of the connection. If we need to send data in real-time, we need a low latency connection, while if we don't need to send data in real-time, we might use a high latency connection.

Note that most of the times, it comes down to a trade-off between these factors, so we need to choose the right connectivity for our use case.

1.1.3 Streaming Data Engineering

This is the main topic of this chapter. This area focuses on the systems and infrastructure that we need to manage various types of data streams efficiently. We have four main concepts:

- Event-based systems: to tame myriads of tiny flows of data

- Data Stream Management Systems (DSMS): to handle continuos massive flows of unstoppable data

- Complex Event Processing (CEP): to manage continuos numerous flows of data that can turn into a torrent

- Event-driven architecture: to tame the forming of an inmense delta made of myriads of flows of any size and speed.

We will cover these concepts in more detail in the next sections.

1.1.4 Streaming Data Science

This is the area that focuses on the algorithms and techniques that we need to analyze data streams efficiently. Note that changes in the data stream can happen at any time, and they cause ML models to lose accuracy. This is why we need to use techniques to adapt to these changes. The three main components of this section are:

- Time series analytics: to explain the past and forecast the future of a continuos flow of data without assuming data independence

- Streaming machine learning: to learn one data at a time from a continuos flow of data without assuming identically distributed data

- Continual learning: to do a long-life learning from a sequence of experiences without forgetting past knowledge

This is the second topic of this course, and we will cover it in more detail in the next chapters.

1.2 Importance of Streaming Data Analytics

One idea: value is about time. Traditionally, data processing creates value by providing insights that are generated with months or years of data gathering. However, in many cases, we need to process data in real-time to create value.

Continuos streaming analysis creates value by providing insights that are generated within seconds or minutes, allowing for real-time decision making, in other words, reactive applications that can respond to events as they happen.

Streaming data analytics is important, as it can:

- Provide real-time insights

- Reduce costs

- Improve efficiency

- Create innovative products

- Generate new revenue streams

Chapter 2

A Tale of Four Streams

2.1 Types of data streams

Data streams come in many forms and sizes. They can be classified into four main categories for the sake of this course:

- Myriads of tiny flows that you can collect, for example, coming from from physical or software sensors

- Continuos massive flows that you cannot stop, for example, from telecomunications and utilities monitoring

- Continuous numerous flows that can turn into a torrent, like physical or cyber alarms

- Myriads of flows of any size and speed forming an inmense delta, like physical or software actuators

2.2 Time models

There are 3 main type of time models for streaming data analytics:

- Stream-only time model

- Absolute time model

- Interval-based time model

2.2.1 Stream-only time model

In this model, time is defined by the order of the events in the stream. This model is used when we only care about the order of the events, and not about the time that they happened.

Therefore, in this model there is no notion of time, so this limits the type of queries that we can do. This reduces the expresiveness of the model, but it also reduces the latency of the system.

2.2.2 Absolute time model

In this model, time is defined by the time that the events happened. This model is useful when we need to know the exact time that the events happened, not only the order in which they happened.

Therefore, in this model we can do more complex queries, like time-based queries or window-based queries. This increases the expressiveness of the model, but it also increases the latency of the system.

2.2.3 Interval-based time model

In this model, time is defined by intervals. This model is useful when we need to know the time that the events happened, but we also need to group the events in intervals.

Therefore, in this model we can do more complex queries, not only time-based queries, but also group the events in specific intervals. This increases the expressiveness of the model, but it also increases the latency of the system.

2.2.4 Trade-off: latency vs expressiveness

There is a trade-off between latency and expressiveness. Let us see the following graph:
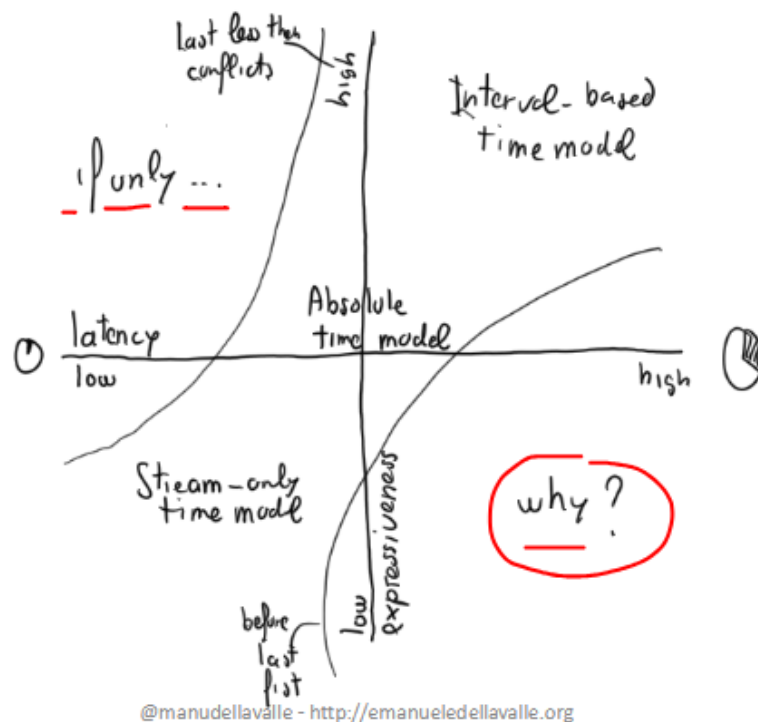
Figure 2.1: Trade-off between latency and expressiveness

10

2.3　Architectures for stream processing

There are four main architectures for stream processing:

- Event-based systems

- Data Stream Management Systems

- Complex Event Processing

- Event-driven architecture

2.3.1　Event Based Systems (EBS)

An Event-Based System is a software architecture paradigm in which data flows and operations are driven by events rather than following a pre-defined control flow. In other words, in an EBS, actions are executed in response to specific events that occur, rather than as a part of a predetermined logical sequence.

Two main concepts:

- Event: an immutable and append-only stream of "business facts" that are generated by sensors, applications, or users.

- Decoupling: means decentralizing the freedom to act, adapt and change

There are two main types of event-based systems:

- Content-based systems

- Topic-based systems

Content-based systems

Content-based systems filter and route messages based on the actual content within the messages. In this model, subscribers specify their interest through content attributes, and the system delivers only messages that match these attributes.

This type of systems are mainly used as an academic effort, as they are hard to implement in practice.

Topic-based systems

Topic-based systems organize messages by predefined topics. In this model, publishers send messages to a specific topic, and subscribers receive all messages sent to the topic they have subscribed to.

This type of systems are the most common in practice, and they are widely available for industrial applications. Some examples are Apache Kafka, MQTT, among others.

Kafka: "the" event-based system

Apache Kafka is a distributed event streaming platform that is capable of handling trillions of events a day. It is used by thousands of companies to build real-time streaming data pipelines and applications.

In this system, topics are partitioned into kafka brokers. Producers share messages over the partitions on a certain topic via hash partitioning or round-robin partitioning. Different consumers can read messages from the same topic, and partitions guarantee parallel reads.

2.3.2 Data Stream Management Systems (DSMS)

Data Stream Management Systems (DSMS) are systems that are designed to handle continuous massive flows of unstoppable data. They are used to process and analyze data streams in real-time.

Data streams are sequences (sometimes unbounded) of time-varying data elements. They represent an almost continuos flow of information, with the most recent information being more relevant, as it describes the current state of a dynamic system.

The nature of streams require a paradigmatic change, from persistent data (one-time semantics) to transient data (continuous semantics).

DSMS semantics

This syem uses continuos queries registered over streams that are observed through a window, like the following picture:
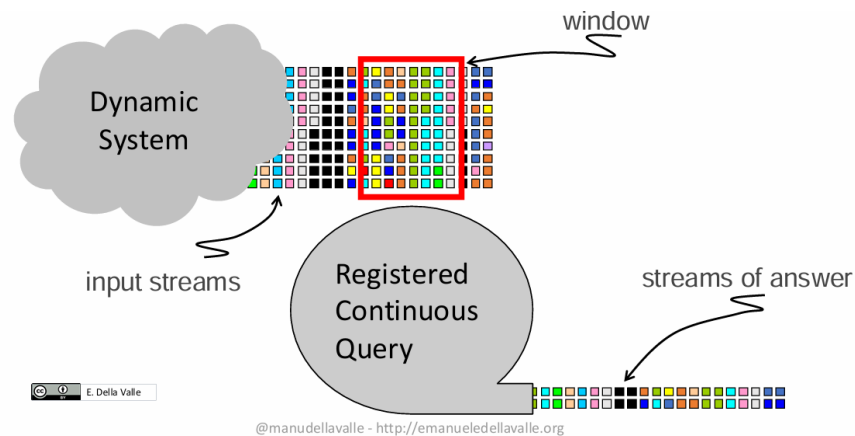

Figure 2.2: DSMS semantics

We can do different operations over the data stream. For example, we can do incremental operations, like:

→ Filtering (more generally, mapping)

$\rightarrow$ Grouping

$\rightarrow$ Count, sum, min, max, means

$\rightarrow$ Variance and standard deviation

Specially, for variance calculation, we can use the Welford's algorithm, which is an online algorithm for calculating the variance of a sample. It is more numerically stable than the standard formula for variance, and it is also more efficient. The algorithm is as follows:

Algorithm 1 Welford's algorithm

1: $n \leftarrow 0$
2: $\mu \leftarrow 0$
3: $M2 \leftarrow 0$
4:
5: **procedure** UPDATE(x)
6:      $n \leftarrow n + 1$
7:      $\delta \leftarrow x - \mu$
8:      $\mu \leftarrow \mu + \frac{\delta}{n}$
9:      $\delta_2 \leftarrow x - \mu$
10:      $M2 \leftarrow M2 + \delta \cdot \delta_2$
11: **end procedure**

However, there are some operations that we are not able to exactly calculate with continuos semantics, like the median, mode, top-k, distinct count, etc. For some of these operations, we can use approximations.

Window types

There are three main types of windows in DSMS:

- Sliding window: a window that slides over the stream

- Tumbling window: a window that is fixed in time

- Session window: a window that groups events that are close in time

Note that we are not specifying the time model. This is because DSMS can work with any time model, as long as it is specified in the query. For example, if we are working with the stream-only time model, we define physical windows. E.g.: for a tumbling window, every 10 events; for a sliding window, consider the last 10 events every 5 events; Note that session windows cannot be specified.

If we consider an absolute time model, we define logical windows. E.g.: for a tumbling window, every 10 seconds; for a sliding window, consider the last 10 seconds every 5 seconds; for a session window, every 10 seconds of inactivity.

An example of a windowed aggregation query in DSMS is the following:

```
1  SELECT STREAM
2      HOP_END(rowtime, INTERVAL '1' HOUR, INTERVAL '3' HOUR),
3      COUNT(*),
4      SUM(units)
5  FROM Orders
6  GROUP BY
7      HOP(rowtime, INTERVAL '1' HOUR, INTERVAL '3' HOUR);
```

Streaming joins

A streaming join is an operation that combines two or more streams based on a common key or specific attribute.

In traditional databases, combining rows from different tables is a basic operation, but in stream processing is hard because the data is not static but in constant motion.

Streams are data that flows over time. Therefore, joins consider the temporal sycronization of the data via time windows to limit the join to events that occur within a certain time interval.

An example:

```
1  SELECT STREAM o.rowtime o.productId, o.orderId,
2              s.rowtime AS shipTime
3  FROM Orders AS o JOIN Shipments AS s
4      ON o.orderId = s.orderId AND
5          s.rowtime BETWEEN
6              o.rowtime AND
7              o.rowtime + INTERVAL '1' HOUR;
```
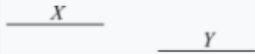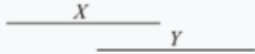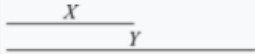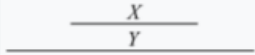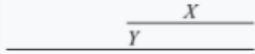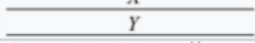
2.3.3 Complex Event Processing (CEP)

Complex Event Processing (CEP) is a method of tracking and analyzing streams of information from different sources to identify patterns and relationships. It is used to process and analyze data streams in real-time.

CEPs add the ability to deploy rules that describe how composite events can be generated from primitive ones. Typical CEP rules search for sequences of events that match a pattern, for example: `Raise C if A → B`. Time is a key factor in CEP, as it is used to determine the order of events and the time window in which the events must occur.

CEP semantics

Complex Event Processing (CEP) uses subsets of Allen's interval algebra to define the temporal relationships between events. The main operators are:

Relation	Illustration	Interpretation
$X < Y$ $Y > X$	X Y	X takes place before Y
$X \mathbf{m} Y$ $Y \mathbf{mi} X$	X Y	X meets Y (*i* stands for *inverse*)
$X \mathbf{o} Y$ $Y \mathbf{oi} X$	X Y	X overlaps with Y
$X \mathbf{s} Y$ $Y \mathbf{si} X$	X Y	X starts Y
$X \mathbf{d} Y$ $Y \mathbf{di} X$	X Y	X during Y
$X \mathbf{f} Y$ $Y \mathbf{fi} X$	X Y	X finishes Y
$X = Y$	X Y	X is equal to Y

@manudellavalle - http://emanueledellavalle.org

Figure 2.3: Allen's interval algebra

Event Processing Language (EPL)

Event Processing Language (EPL) is a language that is used to define rules in CEP. It has the capability to describe complex event patterns and relationships, to create queries that monitor real-time data streams.

It is supported by many CEP engines. The main engine we are going to use in this course is Esper, although there are others like Oracle CEP.

An example of a query in EPL is the following:

```
1   insert into alertIBM
2   select *
3   from pattern [
4       every (
5           StockTick(name="IBM", price > 100)
6           ->
7           (StockTick(name="IBM", price < 100)
8               where timer:within(60 seconds))
9       )
10  ];
```

This query alerts on each IBM stock tick with a price gretaer than 100, followed by a tick with a price less than 100 within 60 seconds.

2.3.4 Event-driven Architecture

Event-driven architecture is a software architecture paradigm promoting the production, detection, consumption of, and reaction to events. An event can be defined as a significant change in state.

In this paradigm, the system only worries about writing the events to the event log, and the rest of the system can react to these events. This allows for a more decoupled system, where each consumer can read react to the events asyncronously.
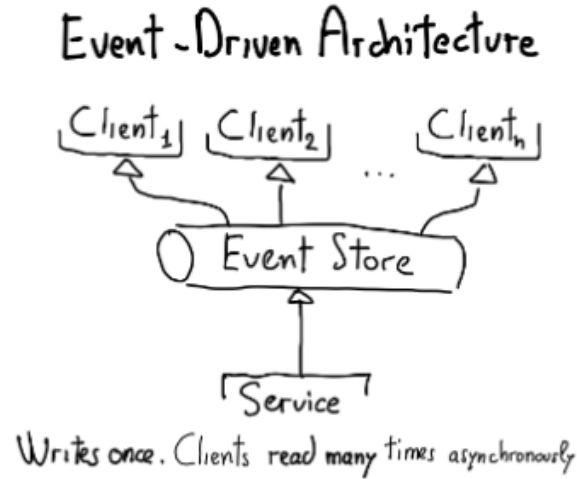
Figure 2.4: Event-driven architecture

EDAs have two main functionalities:

- Notifications, which is just a stream of observations

- Data replication, which is a stream of changes

16

Chapter 3

EPL and Esper

3.1 What is Esper?

Esper is an open-source software product for Complex Event Processing (CEP) and Stream Analytics, developed by EsperTech. It turns a large amount of incoming event series or streams into actionable insights in real-time. It is a lightweight, high-performance, and scalable engine that can process millions of events per second.

Esper provides a SQL-like language called EPL (Event Processing Language) to define queries and patterns over event streams. It also provides a Java API to integrate with Java applications.

Figure 3.1: Esper Architecture

3.1.1 Esper features

- It is implemented as a Java library, so it can be easily integrated with Java applications.

- It provides a SQL-like language called EPL to define queries and patterns over event streams.

- it is designed for performance and scalability, and it can process millions of events per second (high throughput and low latency).

17

- It can interact with static/historical data, and it can be used to correlate real-time data with historical data.
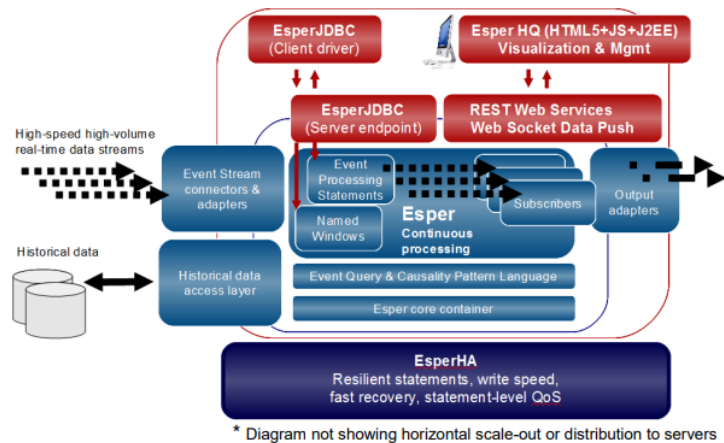
- It has a configurable push and pull communication.

3.2   EPL in a nutshell

EPL is a SQL-like language that allows you to define queries and patterns over event streams. It is used to define the logic that Esper uses to process events.

As in the DSMS approach, it offers stream-to-relation operators, e.g., windowing, relation-to-relation operators, like joins, proections, selections, and aggregation; and relation-to-stream operators to produce output. As in the CEP approach, it offers a comprehensive set of operators to define and detect patterns in event streams.

EPL is similar to SQL, as seen in some of its syntax (e.g., `SELECT`, `FROM`, `WHERE`, `GROUP BY`, etc.). However, it is based on event streams and views, rather than tables. Here, views define the data that is available for queries, they can represent windows over streams, and they can also sort events, derive statistics from event attributes, group events, etc.

3.3   EPL syntax: fire alarms example

To learn more about its syntax, let us consider the following example. Suppose we have a set of fire alarms that send events when they detect smoke. We want to detect when a fire is happening by correlating the events from the fire alarms. Specifically, to detect a fire, we need to receive a smoke event followed by a temperature > 50 event, within 2 minutes, at the same sensor.

In this section, we will learn how to:

- Define event types as schemas

- Create simple queries

- Define different types of windows

- Personalize the output stream

- Recognize complex patterns of events

3.3.1   Creating a schema

In EPL, we can define schemas to define the structure of the events that we are going to process. They are useful to register and recognize specific event types when creating queries, especially the most complex ones.

To define a schema, we mainly use the `CREATE SCHEMA` statement, although we can also use the runtime configuration API `addEventType` method. We use the following syntax:

```
1  create schema --keyword to create a schema
2  schema_name [as...] --name of schema (you can add an alias)
3  (
4      field_name data_type [,...] --field name and data type
5      [inherits inherited_schema_name] --it can inherit from another schema
6  );
```

In our example, we need to describe 3 types of events: smoke events, temperature events, and fire events. For that, we can define the following schemas:

```
1  create schema SmokeSensorEvent (
2      sensor string,
3      smoke boolean,
4  );
5
6  create schema TemperatureSensorEvent (
7      sensor string,
8      temperature double,
9  );
10
11 create schema FireEvent (
12     sensor string,
13     smoke boolean,
14     temperature double,
15 );
```

3.3.2 Creating a simple query

In EPL, we can define queries in the same way we do in SQL. We can use the SELECT statement to select fields, the FROM statement to select from a stream, the WHERE statement to filter the data, and so on. Let's see the following example:

```
1  @Name('Query_0')
2  select *
3  from TemperatureSensorEvent
4  where temperature > 50;
```

In this query, we are selecting all fields from the TemperatureSensorEvent stream where the temperature is greater than 50. However, in this case, the query is getting executed every time a new event arrives, so it is not efficient. We can consider using an event-based system style, like this:

```
1  @Name('Query_0_BIS')
2  select *
3  from TemperatureSensorEvent(temperature > 50);
```

In this way, the events are filtered before being processed by the query, which is a little more efficient.

19

We could also want to obtain the results of each sensor separately. For that, we can use the `GROUP BY` statement, as follows:

```
@Name('Query_1')
select sensor, avg(temperature)
from TemperatureSensorEvent
group by sensor;
```

In this query, we are selecting the sensor and the average temperature from the corresponding stream, grouping by sensor. This way, we can obtain the average temperature of each sensor, observed in the whole stream.

3.3.3 Defining windows

In EPL, we can define windows to group events in time or in the number of events. There are 2 main types of windows, regarding the chosen time model: logical windows, that are based on absolute time, and physical windows, that are based on the ocurrence of events.

From each type, we can define 3 subtypes: tumbling windows, sliding windows, and hopping windows. We will see how to define each one of them.

Tumbling windows

Tumbling windows are the simplest type of windows. They divide the stream into fixed-size, non-overlapping windows. This size may be measured in absolute time (logical tumbling windows) or in the number of events (physical tumbling windows).

To use a logical tumbling window, we use the following syntax:

```
@Name('Query_2')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:time_batch(4 seconds)
group by sensor;
```

In this query, we are selecting the sensor and the average temperature from the corresponding stream, grouping by sensor, and using a logical tumbling window of 4 seconds. This way, we can obtain the average temperature of each sensor, observed in the last 4 seconds, every 4 seconds.

We can also use physical tumbling windows, as follows:

```
@Name('Query_3')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:length_batch(4)
group by sensor;
```

In this way, we can obtain the average temperature of each sensor, observed in the last 4 events, every 4 events.

Sliding windows

Sliding windows are a more complex type of windows. They are defined by every event that arrives in the stream, and they can overlap. The slide size is determined either by the time (logical) or by a number of events (physical).

To use a sliding window, we use the following syntax:

```
@Name('Query_4')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:time(4 seconds)
group by sensor;
```

In this query, we obtain the average temperature of each sensor, observed in the last 4 seconds, every time an event arrives.

If we want to use a physical sliding window, we can use the following syntax:

```
@Name('Query_5')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:length(4)
group by sensor;
```

In this query, we obtain the average temperature of each sensor, observed in the last 4 events, every time an event arrives.

Hopping windows

Hopping windows are a combination of tumbling and sliding windows. They are defined by a fixed-size window that slides by a fixed-size increment. This type of windows tend to overlap. The window size is determined either by the time (logical) or by a number of events (physical).

To use a hopping window, we use the following syntax:

```
@Name('Query_6')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:time(4 seconds)
group by sensor;
output snapshot every 2 seconds;
```

Notice that to use a hopping window, we need to specify the hopping length by modifying the output stream. In this query, we obtain the average temperature of each sensor, observed in the last 4 seconds, every 2 seconds.

Now, let's deepen our knowlegde about how to personalize the output stream.

3.3.4 Output stream

In EPL, we can declare output policies to define how the output stream is going to be generated. We use the `OUTPUT` statement, followed by the desired policy. Also, with the `EVERY` keyword, we can specify the frequency of the output.

There are 4 main types of output policies: snapshot, all, first, and last. Let's see how to use each one of them:

- Snapshot: it generates a snapshot of the current state of the window.

- All: it generates all the events in the window that matches the query.

- First: it generates the first event in the window that matches the query.

- Last: it generates the last event in the window that matches the query.

Let's see an example of this:

```
@Name('Query_7_SNAP')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:time(4 seconds)
group by sensor;
output snapshot every 2 seconds;

@Name('Query_7_ALL')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:time(4 seconds)
group by sensor;
output all every 2 seconds;

@Name('Query_7_FIRST')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:time(4 seconds)
group by sensor;
output first every 2 seconds;

@Name('Query_7_LAST')
select sensor, avg(temperature)
from TemperatureSensorEvent.win:time(4 seconds)
group by sensor;
output last every 2 seconds;
```

In this example, we are obtaining the average temperature of each sensor, observed in the last 4 seconds, every 2 seconds, using the snapshot, all, first, and last output policies.

The main difference between `SNAPSHOT` and `ALL` is that the former generates a snapshot of the current state of the window, while the latter generates all the events in the window that matches the query, no matter if they are a null event.

3.3.5 Complex event patterns

In EPL, we can define complex event patterns to detect specific sequences of events in the stream. We use the `PATTERN` statement, followed by the desired pattern. Let's see an example of this:

```
1  @Name('Query_8')
2  select *
3  from pattern [
4      a=SmokeSensorEvent(smoke=true) ->
5      b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
6  ];
```

This query is searching for a smoke event followed by a temperature > 50 event. Note that this query only matches the first occurrence of the pattern. If we want to match all the occurrences, we should use the `EVERY` keyword.

EVERY clause

The `EVERY` clause is used to re-start the pattern matching process after a successful or failed match. It has different approaches:

- `EVERY (A -> B)`: it matches the pattern every time the event A is followed by the event B. The moment the event B is matched, the pattern matching process is re-started, and the event A is expected again.
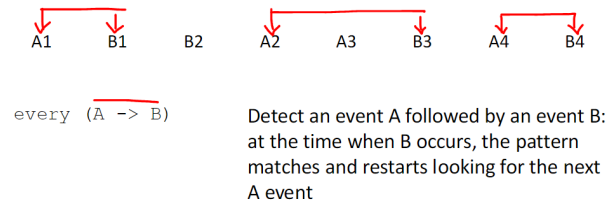
Figure 3.2: Pattern matching with `EVERY (A -> B)`

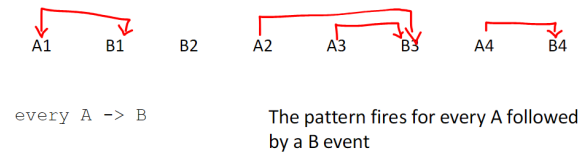Let's see an example of this:

```
1  @Name('Query_9')
2  select *
3  from pattern [
4      every (
5          a=SmokeSensorEvent(smoke=true)
6          ->
7          b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
8      )
9  ];
```

- EVERY A -> B: it matches the pattern every time the event A is followed by the event B. In this case, the pattern matching process fires every time an event A is matched, and succesfully returns when the event B is matched, so it will return every A followed by B.

every A -> B

The pattern fires for every A followed by a B event

B1	{A1, B1}
B3	{A2, B3}, {A3, B3}
B4	{A4, B4}

Figure 3.3: Pattern matching with EVERY A -> B

Let's see an example of this:

```
1  @Name('Query_10')
2  select *
3  from pattern [
4      every a=SmokeSensorEvent(smoke=true)
5          ->
6          b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
7  ];
```

- A -> EVERY B: it matches the pattern every time the event A is followed by the event B. In this case, once the event A is matched, this pattern will be matched every time the event B is encountered, so it will return every B following that A.
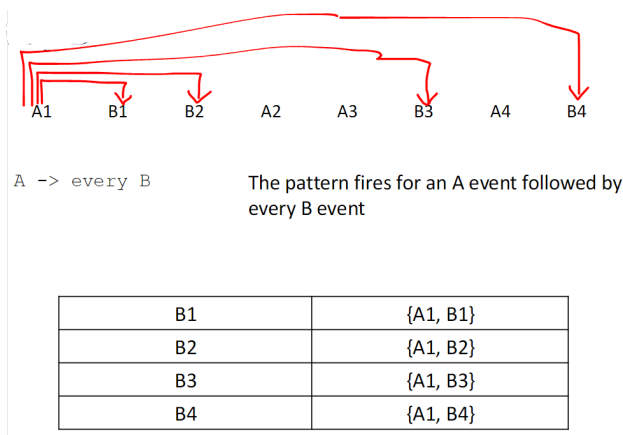

A -> every B

The pattern fires for an A event followed by every B event

B1	{A1, B1}
B2	{A1, B2}
B3	{A1, B3}
B4	{A1, B4}

Figure 3.4: Pattern matching with A -> EVERY B

Let's see an example of this:

```
1  @Name('Query_11')
2  select *
3  from pattern [
4      a=SmokeSensorEvent(smoke=true)
5          ->
6          every b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
7  ];
```

- EVERY A -> EVERY B: it matches the pattern every time the event A is followed by the event B. In this case, the pattern matching process fires every time an event A is matched, and for each A, it will return every B that follows that A.
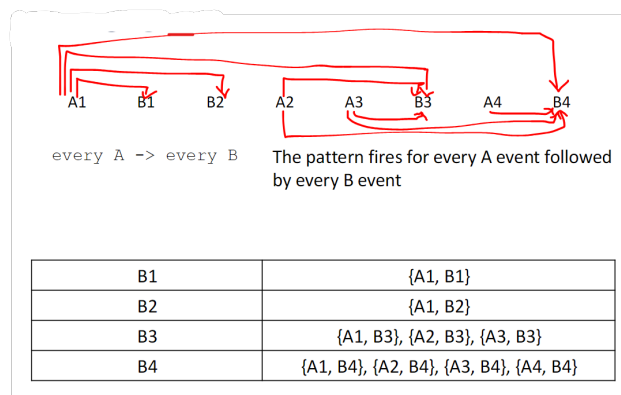

Figure 3.5: Pattern matching with EVERY A -> EVERY B

Let's see an example of this:

```
1  @Name('Query_12')
2  select *
3  from pattern [
4      every a=SmokeSensorEvent(smoke=true)
5          ->
6          every b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
7  ];
```

Note that the EVERY clause restarts the pattern matching process multiple times. This can be very resource-intensive, so it should be used with caution. Also, we don't always want to match every pattern in the stream, but only in an specific window, or in an specific context. For that, we have pattern guards.

Pattern guards

Pattern guards are used to restrict the pattern matching process to a specific context. There are 2 main types of pattern guards: TIMER:WITHIN and AND NOT. Let's see how to use each one of them:

- `TIMER:WITHIN`: it restricts the pattern matching process to a specific time window. It is used to specify a time limit for the pattern matching process. Let's see an example of this:

```
1  @Name('Query_13')
2  select *
3  from pattern [
4      a=SmokeSensorEvent(smoke=true)
5          ->
6          b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
7      where timer:within(2 minutes)
8  ];
```

In this query, we are searching for a smoke event followed by a temperature > 50 event, within 2 minutes. This way, we are restricting the pattern matching process to a specific time window.

- `AND NOT`: it restricts the pattern matching process to exclude specific events. It is used to specify that an event should not be matched. Let's see an example of this:

```
1  @Name('Query_14')
2  select *
3  from pattern [
4      a=SmokeSensorEvent(smoke=true)
5          ->
6          b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
7      and not c=FireEvent(sensor = a.sensor)
8  ];
```

In this query, we are searching for a smoke event followed by a temperature > 50 event, and excluding the case when a fire event is detected at the same sensor. This way, we are restricting the pattern matching process to exclude specific events.

Operators precedence

For all these operations, we need to consider the operators precedence. The operators are evaluated in the following order:

Order	Operator	Description	Example
1	guard postfix	where timer:within and while (expression) (incl. withinmax and plug-in pattern guard)	MyEvent where timer:within(1 sec) a=MyEvent while (a.price between 1 and 10)
2	unary	every, not, every-distinct	every MyEvent timer:interval(5 min) and not MyEvent
3	repeat	[num], until	[5] MyEvent [1..3] MyEvent until MyOtherEvent
4	and	and	every (MyEvent and MyOtherEvent)
5	or	or	every (MyEvent or MyOtherEvent)
6	followed-by	->	every (MyEvent -> MyOtherEvent)

Figure 3.6: Operators precedence

26

3.3.6   Composing queries

Typically, queries in EPL are placed in a network. In this way, we can make queries interact with each other, and we can define the order in which they are executed. This allows us to define complex queries by composing simpler ones.
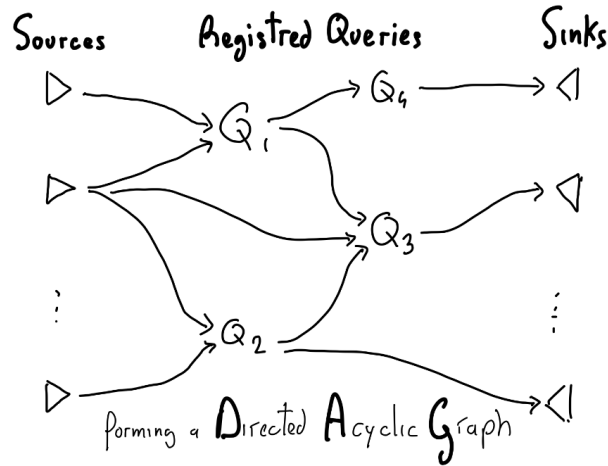

Figure 3.7: Query network

We use the `INSERT INTO` statement to insert the results of a query into a stream. In this way, we can use the results of a query as input for another query. Let's see an example of this:

```
@Name('Query_15')
insert into FireEvent
select a.sensor, a.smoke, b.temperature
from pattern [
    every (
        a=SmokeSensorEvent(smoke=true)
        ->
        b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
        where timer:within(2 minutes)
    )
];
```

In this query, we are searching for a smoke event followed by a temperature > 50 event, within 2 minutes. If the pattern is matched, the results are inserted into the `FireEvent` stream. Then, we can use this query:

```
@Name('Query_16')
select count(*)
from FireEvent.win:time(10 minutes);
```

In this query, we are counting the number of fire events detected in the last 10 minutes. This way, we can use the results of the first query as input for the second query.

3.4 Joins in EPL

In SQL, joins are performed on tables, whic are static data structures. The semantics of SQL joins are based on relational data model and queries that work with complete data sets. These joins do not consider the time dimension of the data.

In EPL, joins are performed on event streams, which are dynamic data structures. The semantics of EPL joins are based on event streams and queries that work with continuous data. These joins intrinsically consider the time dimension of the data.

Generally, joins are classified in 3 main types:

- Inner join: it returns only the rows that have matching values in both tables.

- Left/ight join: it returns all the rows from the left/right table, and the matched rows from the other table.

- Full join: it returns all the rows when there is a match in one of the tables.

Note that this is true for SQL and EPL. Also based on the structure on which the join is performed, in EPL we can distinguish between 3 types of joins:

- Stream to stream joins

- Table to table joins

- Stream to table joins (and viceversa)

Note that complex event pattern matching (CEP), seen in the previous section, is a form of join, where the join condition is based on the temporal order of the events.

3.4.1 Stream to stream joins

Stream to stream joins are used to join 2 event streams. In each stream, we specify a window to define the time frame in which the join is performed. For example, if we use a window of 5 seconds, we will only consider the events that arrive in the last 5 seconds. Its syntax is as follows:

```
select *
from View#time(9 sec) as v
    inner join
    Click#time(9 sec) as c
    on v.id = c.id;
```

In this query, we are joining the `View` and `Click` streams on the `id` field. We are using a window of 9 seconds for each stream. This way, we are only considering the events that arrive in the last 9 seconds.

Consider that the queries are only executed when an event arrives in the stream, and only returns the new results that match the join condition. The following figure describes the process of a stream to stream (inner) join:
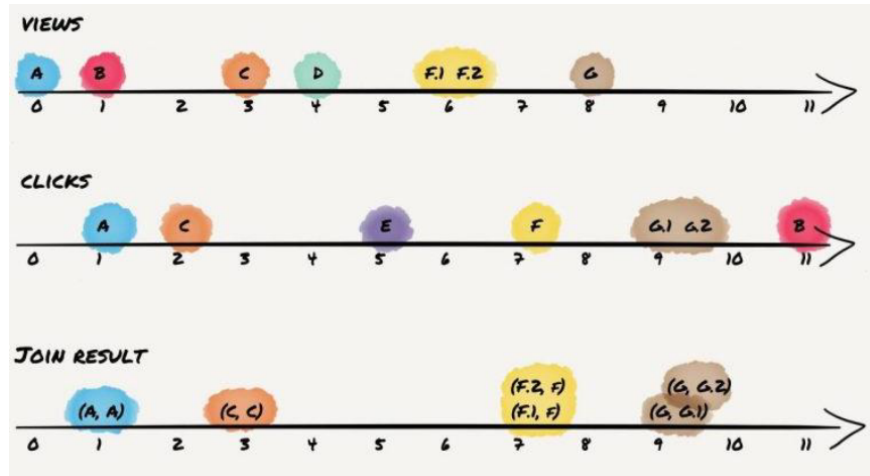
Figure 3.8: Stream to stream join

Note that we can also perform left/right outer joins and full outer joins. The syntax is similar to the inner join, but we use the `left outer join`, `right outer join`, and `full outer join` statements, respectively.

Joins among streams have profoundly different semantics, because they are designed to run on real-time data streams, essentially unbounded record sequences. Events are joined only if they arrive within a specific time window. This temporal aspect is central and instrinsic to the join semantics of streams. The concept of time is crucial, as Joins depend on the time synchronization of the events in the streams, which instroduces a significant difference with respect to traditional SQL joins.

3.4.2   Table to table joins

EPL offers 3 ways to build a table from an event stream:

- `KEEP ALL`: this window is one that retains all arriving events. However, take should be taken to remove events from the window, in a timely manner.

- `UNIQUE`: this window is one that retains only the most recent event for each unique key.

- `CREATE TABLE` with `INSERT INTO`: this window is one that retains all arriving events, but it is not a window in the sense of a time window.

We will mainly use the `UNIQUE` window to build tables from event streams. This window is one that retains only the most recent event for each unique key. Its syntax is as follows:

```
select *
from View#unique(id) as v
    inner join
    Click#unique(id) as c
    on v.id = c.id;
```

29

In this query, we are joining the `View` and `Click` streams on the `id` field. We are using the `unique` window to build tables from the event streams. This way, we are only considering the most recent event for each unique key, in this case, the `id` field.

Note that we are not actually joining tables, but event streams. The tables are built from the event streams, and the join is performed on these tables. So we call them "tables" as they seem to inherit the unique key constraint from the SQL tables. The following figure describes the process of a table to table (inner) join:
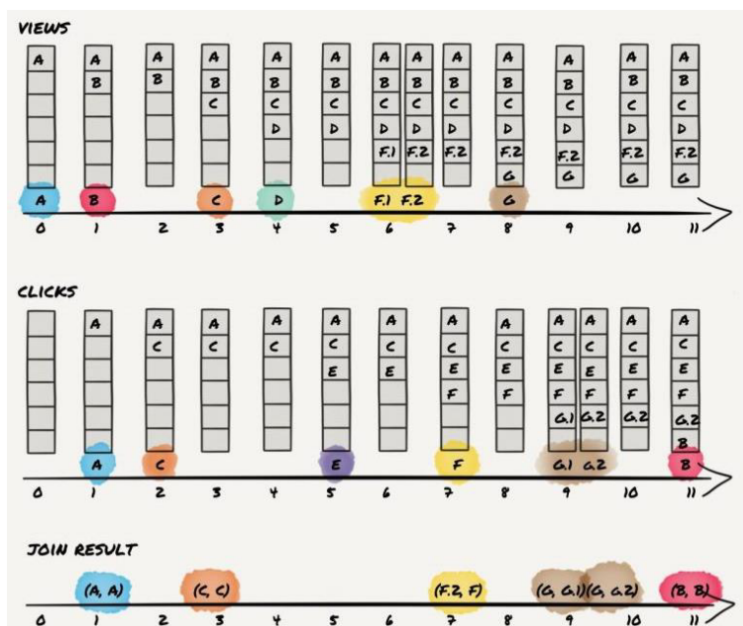

Figure 3.9: Table to table join

Note that the events persist over time, only being replaced by newer events with the same key. This is a key difference with respect to stream to stream joins, where the events are only considered within a specific time window.

We can also perform left/right outer joins and full outer joins. The syntax is similar to the inner join, but we use the `left outer join`, `right outer join`, and `full outer join` statements, respectively.

3.4.3 Stream to table joins

These type of joins introduce a new keyword: `UNIDIRECTIONAL`. This keyword is used in the `JOIN` clause to identify streams that provide the events to execute the join. If the keyword is present for a stream, all other streams in the `FROM` clause become passive streams. That means that when a new event arrives or leaves a data window of a passive stream, the join does not generate new results.

Therefore, the `UNIDIRECTIONAL` keyword makes the stream-to-table join asymmetric: only the input from the stream with the `UNIDIRECTIONAL` keyword will trigger the join. And because the

join is not-windowed, this stream is stateless; thus join lookups from the table to the stream are not possible.

The syntax for a stream to table join is as follows:

```
select *
from View as v
    unidirectional inner join
    Click#unique(id) as c
    on v.id = c.id;
```

In this query, we are joining the `View` stream with the `Click` table on the `id` field. We are using the `unique` window to build the table from the `Click` stream. This way, we are only considering the most recent event for each unique key, in this case, the `id` field. In this case, a new click event won't trigger the join, but a new view event will, and because the stream is stateless, join lookups from the Click table to the View stream are not possible.

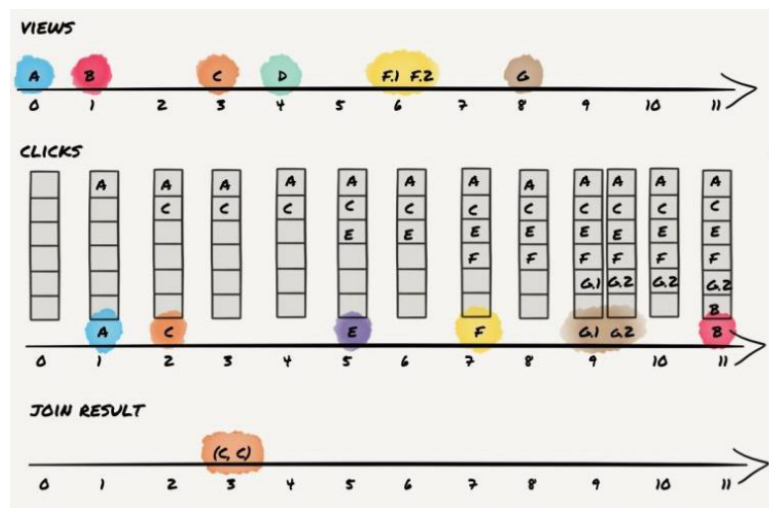The following figure describes the process of a stream to table (inner) join:

Figure 3.10: Stream to table join

We can also perform left/right outer joins and full outer joins. The syntax is similar to the inner join, but we use the `left outer join`, `right outer join`, and `full outer join` statements, respectively.

3.5 Contexts in EPL

In EPL, we can also define contexts. A context enables more flexible and stateful processing in scenarios requiring grouping or conditions that go beyond simple event windows. In fact, contexts define a new type of window (different than tumbling and hopping windows), which we will name a session window.

Why use contexts? Here is a list of advantages:

→ Better control: context gives fine-grained control over how events are processed.

→ Stateful processing: handles cases where the state is important, e.g., sessions or transactions.

→ Efficient partitioning: it helps reduce complexity in event, partitioning by user, session, or custom logic.

We have three main types of contexts:

- Context by key: partitions events based on a key, e.g., `userId`.

- Context by start/end conditions: defined by specific events or triggers.

- Context by time: defines temporal windows (generalizes logical windows, e.g., it allows to declare session windows).

3.5.1 Context by key

Context by key is used to partition events based on a key. It is useful when we want to group events by a specific attribute, e.g., `userId`. Its syntax is as follows:

```
create context UserSessionContext
partition by userId from UserActionEvent;
```

In this context, we are partitioning events by the `userId` attribute from the `UserActionEvent` stream. This way, we are grouping events by the `userId` attribute. Note that this process is highly parallelizable, as each partition can be processed independently.

3.5.2 Context by start/end conditions

Context by start/end conditions is defined by specific events or triggers. It is useful when we want to group events based on specific conditions, e.g., when a session starts or ends. Its syntax is as follows:

```
create context OrderContext
initiated by OrderEvent(orderStatus = 'NEW')
terminated by OrderEvent(orderStatus = 'COMPLETED');
```

In this context, we are defining a session window that starts when an `OrderEvent` with `orderStatus = 'NEW'` arrives and ends when an `OrderEvent` with `orderStatus = 'COMPLETED'` arrives. This way, we are grouping events based on the start and end conditions.

Note that we could use this context along with the context by key to partition events based on a key and start/end conditions. Look at the following example:

```
create context OrderContext;
partition by userId from UserActionEvent;
initiated by UserActionEvent(action = 'start_order');
terminated by UserActionEvent(action = 'complete_order');
```

3.5.3 Context by time

Context by time allows us to track actions within fixed periods of time. They are triggered by some condition and last for a specific time. Its syntax is as follows:

```
1  create context TimeBatchContext
2  partition by userId from UserActionEvent;
3  initiated by UserActionEvent(action = 'start_order');
4  terminated after 10 minutes;
```

In this context, we are defining a session window that starts when an `UserActionEvent` with `action = 'start_order'` arrives and ends after 10 minutes. This way, we are grouping events based on the time window. Note that we are also partitioning events by the `userId` attribute.

3.5.4 Comparison: Windows vs. Contexts

Windows and contexts are both used to group events, but they have different semantics. Let us look at the following figure that shows the main differences between windows and contexts:

Windows

- operate over a **fixed** time frame or number of events
- are stateless
- reset after each use

Contexts

- **provide more flexible segmentation**,
- have states that persist across event streams
- persist until terminated, providing a more robust grouping for complex scenarios

Figure 3.11: Windows vs. Contexts