Politecnico di Milano
School of Industrial and Information Engineering
Academic Year 2024-2025

Streaming Data Analytics

Professor: Emanuelle Della Valle

Last updated: January 19, 2025

Made by Roberto Benatuil Valera

Contents

Chapter 1

Introduction: Streaming data engineering

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:

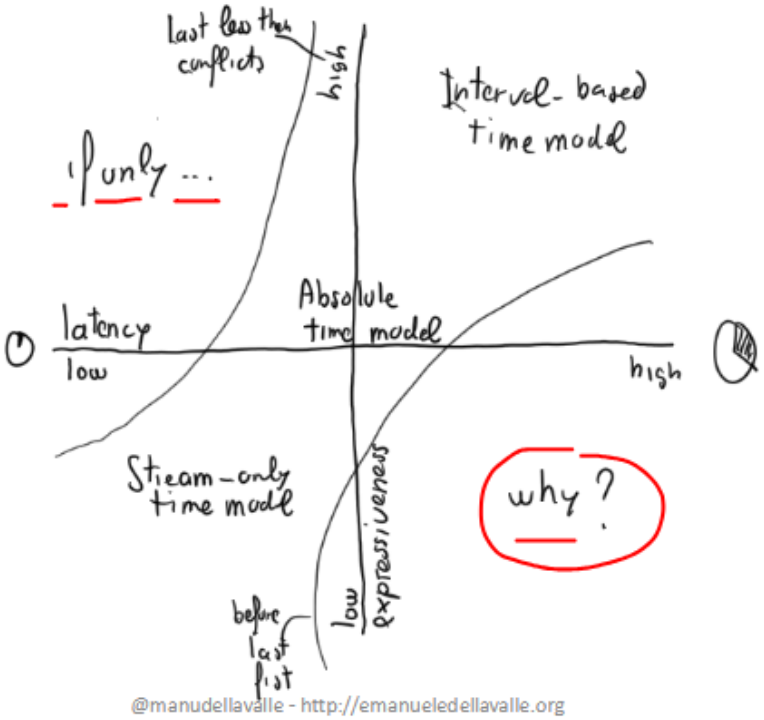


Figure 2.1: Trade-off between latency and expressiveness

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:

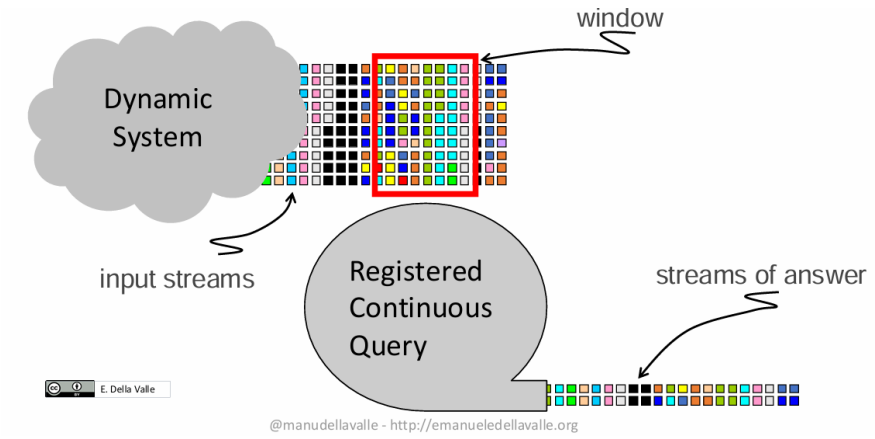


Figure 2.2: DSMS semantics

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

→ Filtering (more generally, mapping)

→ Grouping

→ Count, sum, min, max, means

→ 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

$n \leftarrow 0$
$\mu \leftarrow 0$
$M2 \leftarrow 0$

procedure UPDATE(x)
    $n \leftarrow n + 1$
    $\delta \leftarrow x - \mu$
    $\mu \leftarrow \mu + \frac{\delta}{n}$
    $\delta_2 \leftarrow x - \mu$
    $M2 \leftarrow M2 + \delta \cdot \delta_2$
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:

```
SELECT STREAM
    HOP_END(rowtime, INTERVAL '1' HOUR, INTERVAL '3' HOUR),
    COUNT(*),
    SUM(units)
FROM Orders
GROUP BY
    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:

```
SELECT STREAM o.rowtime o.productId, o.orderId,
            s.rowtime AS shipTime
FROM Orders AS o JOIN Shipments AS s
    ON o.orderId = s.orderId AND
        s.rowtime BETWEEN
            o.rowtime AND
            o.rowtime + INTERVAL '1' HOUR;
```

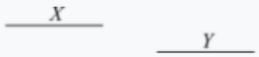
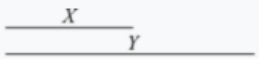
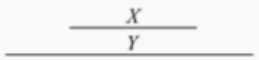
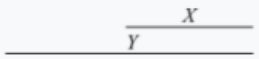
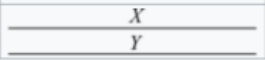
2.3.3 Complex Event Processing (CEP)

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` $\rightarrow$ `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 takes place before Y
X **m** Y Y **mi** X		X meets Y (*i* stands for *inverse*)
X **o** Y Y **oi** X		X overlaps with Y
X **s** Y Y **si** X		X starts Y
X **d** Y Y **di** X		X during Y
X **f** Y Y **fi** X		X finishes 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:

```
insert into alertIBM
select *
from pattern [
    every (
        StockTick(name="IBM", price > 100)
        ->
        (StockTick(name="IBM", price < 100)
            where timer:within(60 seconds))
    )
];
```

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.

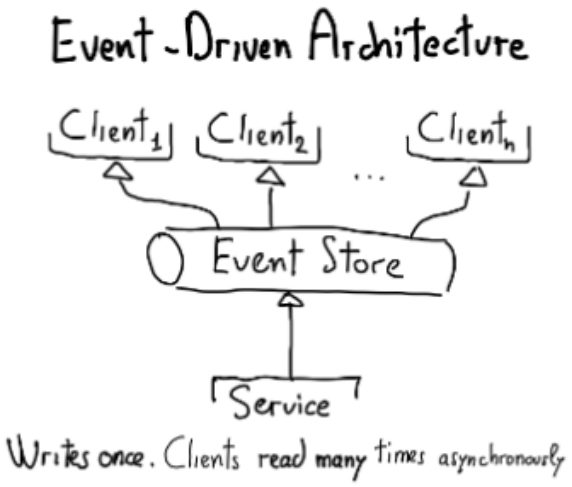


Figure 2.4: Event-driven architecture

EDAs have two main functionalities:

- Notifications, which is just a stream of observations
- Data replication, which is a stream of changes

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.

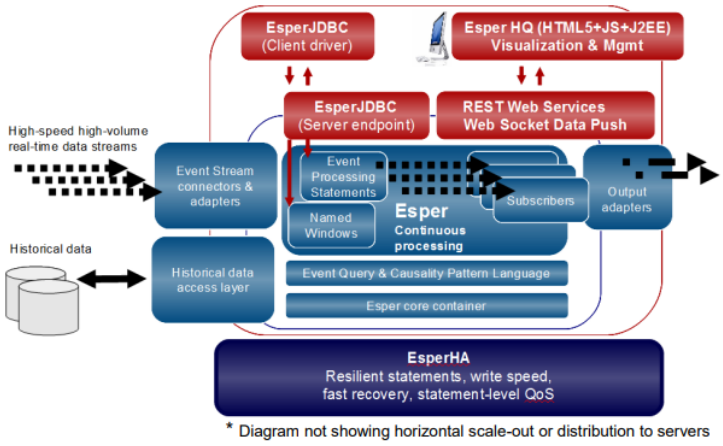


Figure 3.1: Esper Architecture

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).

- 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:

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

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:

```
create schema SmokeSensorEvent (
    sensor string,
    smoke boolean,
);

create schema TemperatureSensorEvent (
    sensor string,
    temperature double,
);

create schema FireEvent (
    sensor string,
    smoke boolean,
    temperature double,
);
```

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:

```
@Name('Query_0')
select *
from TemperatureSensorEvent
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:

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

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

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:

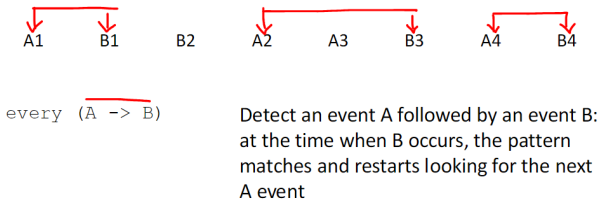
```
@Name('Query_8')
select *
from pattern [
    a=SmokeSensorEvent(smoke=true) ->
    b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
];
```

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.



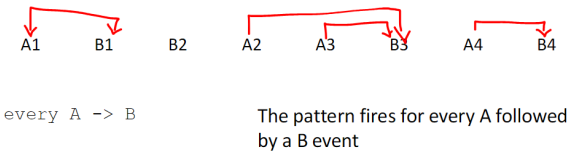
B1	{A1, B1}
B3	{A2, B3}
B4	{A4, B4}

Figure 3.2: Pattern matching with `EVERY (A -> B)`

Let's see an example of this:

```
@Name('Query_9')
select *
from pattern [
    every (
        a=SmokeSensorEvent(smoke=true)
        ->
        b=TemperatureSensorEvent(temperature > 50, sensor = a.sensor)
    )
];
```

- `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.



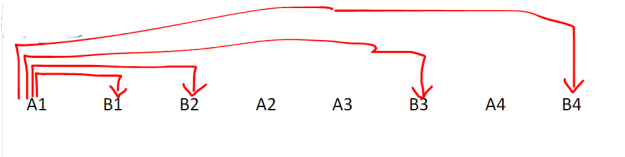
B1	{A1, B1}
B3	{A2, B3}, {A3, B3}
B4	{A4, B4}

Figure 3.3: Pattern matching with `EVERY A -> B`

Let's see an example of this:

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

- `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.



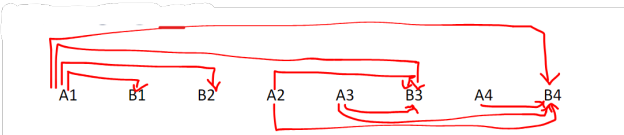
B1	{A1, B1}
B2	{A1, B2}
B3	{A1, B3}
B4	{A1, B4}

Figure 3.4: Pattern matching with `A -> EVERY B`

Let's see an example of this:

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

- `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.



every A -> every B

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

B1	{A1, B1}
B2	{A1, B2}
B3	{A1, B3}, {A2, B3}, {A3, B3}
B4	{A1, B4}, {A2, B4}, {A3, B4}, {A4, B4}

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

Let's see an example of this:

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

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:

```
@Name('Query_13')
select *
from pattern [
    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. 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:

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

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

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.

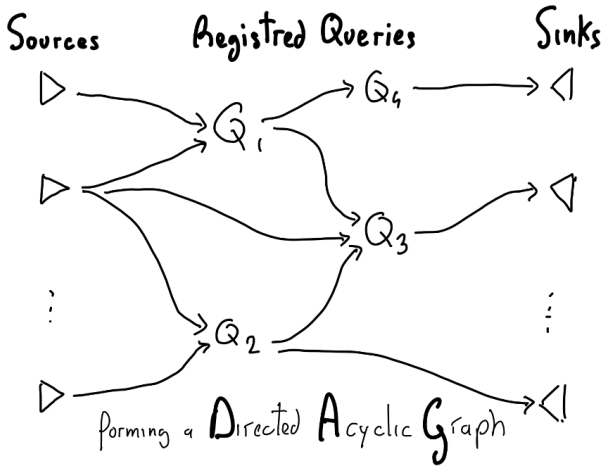


Figure 3.7: Query network

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:

```
1 select *
2 from View#time(9 sec) as v
3     inner join
4     Click#time(9 sec) as c
5     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:

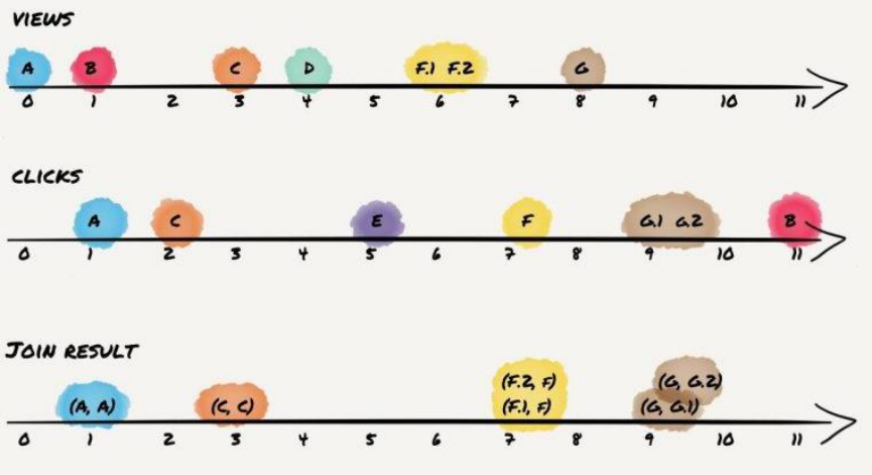


Figure 3.8: Stream to stream join

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;
```

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:

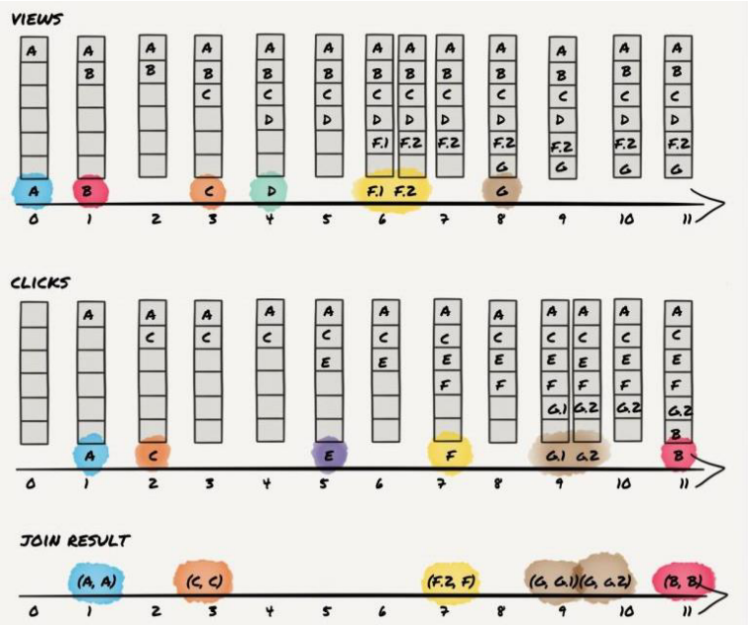


Figure 3.9: Table to table join

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:

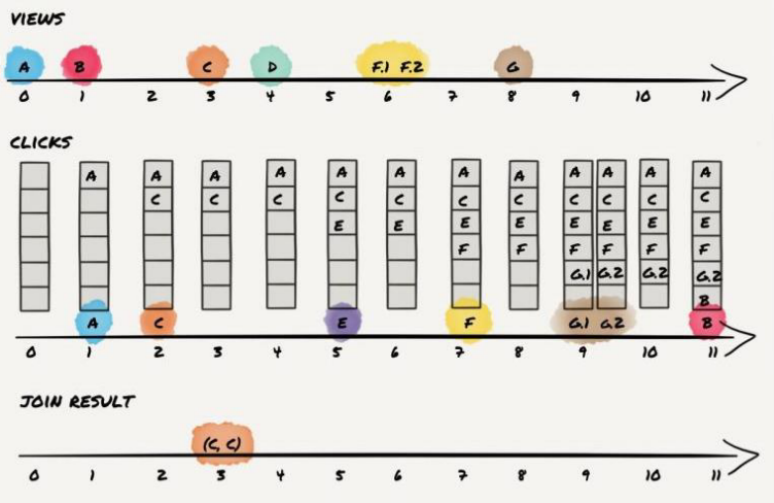


Figure 3.10: Stream to table join

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:

```
create context TimeBatchContext
partition by userId from UserActionEvent;
initiated by UserActionEvent(action = 'start_order');
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:

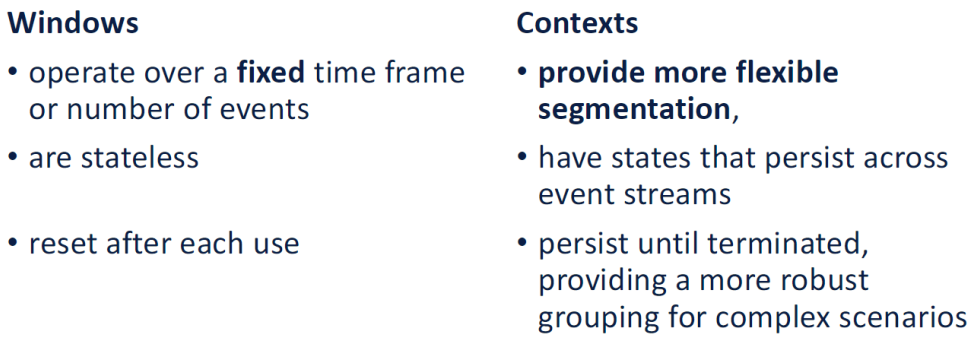


Figure 3.11: Windows vs. Contexts

Chapter 4

Kafka and Spark

4.1 Introduction: scaling stream ingestion and processing

To process a high volume of data, we need systems that are horizontally scalable. This means that we can add more machines to the system to increase its capacity. In this chapter, we will discuss two such systems: Apache Kafka and Apache Spark, and how they can be used together to process large volumes of data.

In general, the roles of Kafka and Spark are as follows:

- Kafka is used to collect and store data in real-time. It manages the ingestion of data from multiple sources and stores it in a distributed manner, making it available for processing by other systems.
- Spark is used to process the data stored in Kafka, and generate insights. It is a distributed computing system that can process large volumes of data in parallel.

Before getting into the details of Kafka and Spark, let's discuss some important concepts that are used in these systems:

- Latency vs throughput
- Data/message

4.1.1 Latency vs throughput

We refer to latency as the time taken for a message to travel from the producer to the consumer. More generally, latency is the amount of time needed to complete a task. Throughput, on the other hand, is the number of messages that can be processed in a given time period.

We can see this with a visual example: consider a highway with cars moving at different speeds. The latency is the time taken for a car to travel from one end of the highway to the other, while the throughput is the number of cars that can pass through the highway in a certain time period, say an hour.

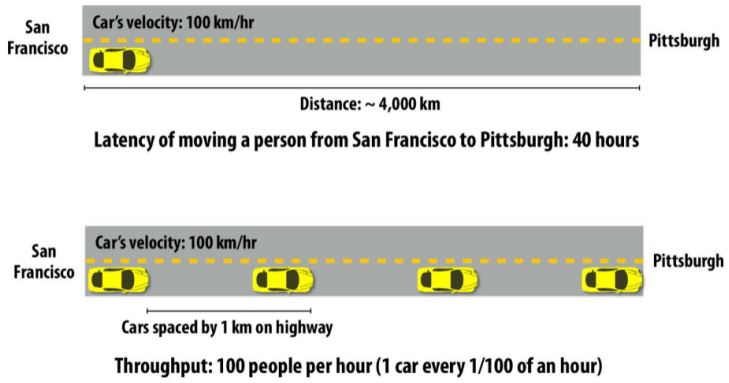


Figure 4.1: Latency vs throughput

We could improve the throughput by adding more lanes to the highway, but this would not necessarily reduce the latency. To improve the latency, we would need to reduce the distance between the two ends of the highway, or increase the speed of the cars. Notice that the latter would also improve the throughput, as more cars would be able to pass through the highway in a given time period.

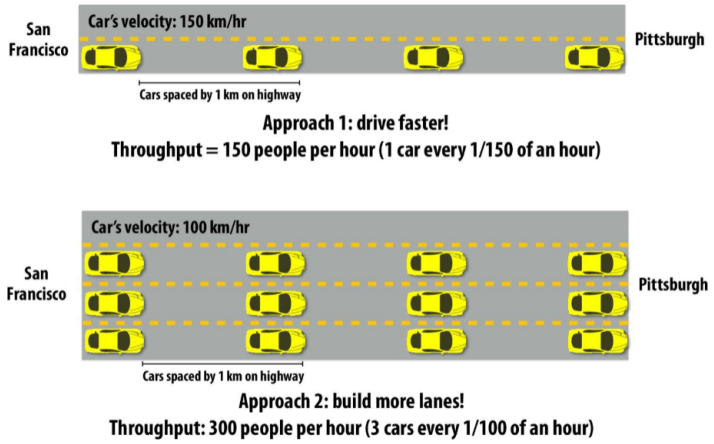


Figure 4.2: Improving latency and throughput

In the context of data processing, we usually want to minimize the latency and maximize the throughput. This is because we want to process data as quickly as possible, and we want to be able to process a large volume of data in a given time period.

4.1.2 Data/message

In the context of data processing, the data that comes in is usually in the form of messages. A message is a unit of data that is sent from a producer to a consumer. For example, in a messaging system like Kafka, a message could be a log entry, a sensor reading, or a user event. Messages can be of different types and sizes, and can contain different types of data. For example, a message

could be a JSON object, a binary blob, or a string.

Note that the way messages are represented can have a big impact on the performance of the system. Following the previous example, imagine that the messages are cars on a highway, and the people in the cars are the data. If the cars are small and can carry only one person, then we would need a lot of cars to transport a large number of people. On the other hand, if the cars are large and can carry many people, then we would need fewer cars to transport the same number of people. The same is true for messages: if the messages are small, then we would need more of them to transport the same amount of data, which would increase the overhead of the system. On the other hand, if the messages are large, then we would need fewer of them to transport the same amount of data, which would reduce the overhead of the system.

In general, the larger the data poonts per message, the higher the throughput, but the higher the latency. This is because larger messages take longer to process, but fewer of them are needed to transport the same amount of data. So, it is needed to find a balance between the size of the messages and the performance of the system.

Compression and binary encodig are the typical ways to increase the data per message, given a fixed message size.

4.2 Apache Kafka

Apache Kafka is a distributed streaming platform that is used for building real-time data pipelines and streaming applications. It is designed to be scalable, fault-tolerant, and highly available. Kafka is used for collecting, storing, and processing large volumes of data in real-time. It is used by many companies, including LinkedIn, Netflix, Uber, and Airbnb, to process data at scale.

4.2.1 Kafka: high-level vs system view

Let us take a conceptual view of Kafka. In general, Kafka is composed of the following components:

- **Producer**: A producer is a system that sends data to Kafka. It sends the messages based on a certain topic.
- **Consumer**: A consumer is a system that reads data from Kafka. It consumes the messages from certain topics that subscribes to.
- **Topics**: A topic is a category of messages in Kafka. It is a way to organize the messages in Kafka. Producers send messages to topics, and consumers read messages from topics.
- **Messages**: A message is a unit of data in Kafka. It is a key-value pair that is sent from a producer to a consumer. Messages are stored in topics and can be read by consumers.
- **Cluster**: A Kafka cluster manages a set of topics.

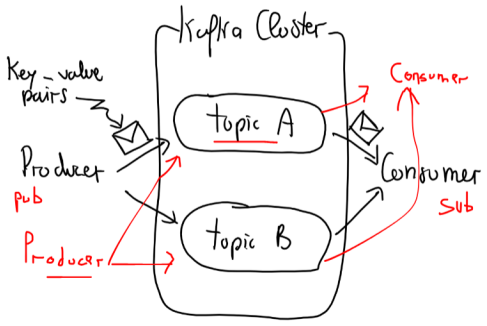


Figure 4.3: Kafka high-level view

This is a high-level view of Kafka. In practice, Kafka uses a distributed architecture to achieve scalability, fault-tolerance, and high availability. So, for that, we need to introduce the following concepts:

- **Brokers**: A broker is a Kafka server that stores the messages. A Kafka cluster is composed of multiple brokers. Each broker stores a subset of the messages in the cluster.
- **Partitions**: A partition is a unit of parallelism in Kafka. A topic is divided into multiple partitions, and each partition is stored on a different broker. This allows Kafka to scale out by distributing the load across multiple brokers.

In general, a broker receives messages from producers, stores them in partitions, and serves them to consumers. A production Kafka cluster typically consists of multiple brokers, each storing multiple partitions of multiple topics. Each broker can handle a certain amount of traffic, and the cluster as a whole can handle a large volume of data.

To ensure fault-tolerance, Kafka uses replication. Each partition is replicated across multiple brokers, so that if one broker fails, the data is still available on other brokers. This allows Kafka to provide high availability and durability.

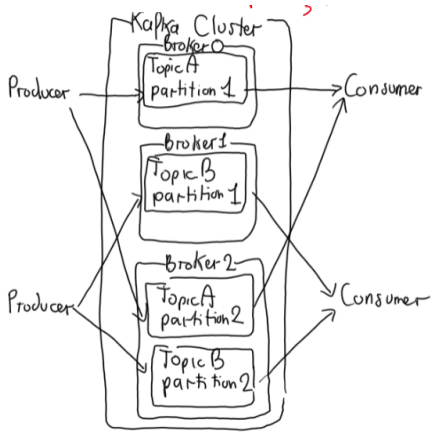


Figure 4.4: Kafka architecture

4.2.2 Kafka: how it works

Let us describe in details how Kafka works. In general, Kafka works as follows:

1. A producer sends a message to a topic. The producer specifies the topic and the message to be sent.
2. The message is sent to a broker. The broker receives the message and stores it in a partition. The partition is chosen based on the key of the message. If no key is specified, then round-robin is used to choose the partition.
3. The message is replicated to other brokers, to ensure fault-tolerance. The number of replicas is configurable, and the replication factor determines how many replicas are created.
4. Each partition is stored on the disk of the broker. The messages are stored in a log-structured format, which allows for fast reads and writes. Each message is assigned an offset, which is a unique identifier for the message. This log is called the **commit log**.

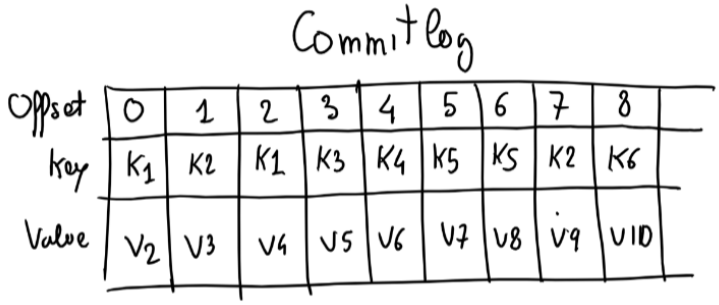


Figure 4.5: Commit log

5. A consumer reads messages from a topic. The consumer specifies the topic and the partition to read from. The consumer reads messages in order of their offset, starting from the last offset read.

Note that each broker can configure its log retention policy, which determines how long messages are kept in the commit log. This allows Kafka to manage the storage of messages and prevent the log from growing indefinitely. In general, the default retention policy is to keep messages for a certain amount of time, after which they are deleted, but we can also configure the retention policy to be based on the size of the log, or the number of messages in the log.

Also, when deleting a log, there is a concept of **compaction**, which is a process that removes duplicate messages from the log. This is useful when we want to keep only the latest version of a message, and discard older versions. Compaction is useful for maintaining the state of a system, and for ensuring that the log does not grow too large.

When making replicas, Kafka uses a leader-follower model. Each partition has one leader and multiple followers (replicas). The leader is responsible for handling reads and writes, while the followers are responsible for replicating the data. If the leader fails, one of the followers is elected as the new leader, and the system continues to operate.

4.2.3 Distributed consumption

Kafka allows for distributed consumption, which means that multiple consumers can read messages from a topic in parallel. This allows Kafka to scale out by distributing the load across multiple consumers.

Consumers can be part of a consumer group, which is a group of consumers that work together to read messages from a topic. Each consumer in the group reads messages from a different partition, so that the load is distributed evenly across the consumers. If a consumer fails, then the partitions that it was reading from are reassigned to other consumers in the group, so that the system continues to operate.

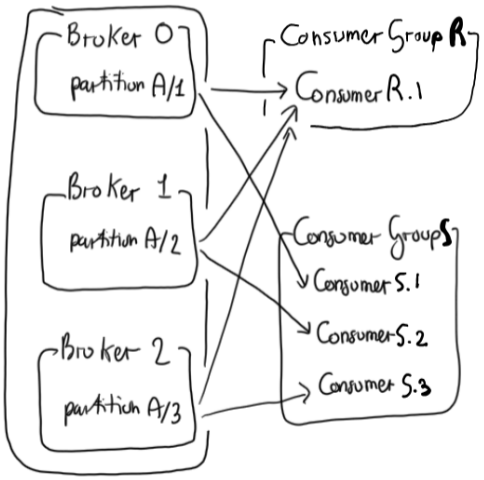


Figure 4.6: Consumer group

In general, Kafka provides at-least-once delivery guarantees, which means that messages are delivered to consumers at least once. This is because Kafka stores messages in the commit log, and consumers can re-read messages if needed. However, Kafka does not provide exactly-once delivery guarantees, which means that messages can be delivered more than once.

4.3 Apache Spark

Apache Spark is an unified analytics engine for big data processing, with built-in modules for streaming, SQL, machine learning, and graph processing. It is designed to be fast, easy to use, and general-purpose. Spark is used by many companies, including Netflix, Uber, and Airbnb, to process data at scale.

Spark is built around the concept of Resilient Distributed Datasets (RDDs), which are fault-tolerant collections of objects that can be operated on in parallel. RDDs are the fundamental data structure in Spark, and are used to represent data that is distributed across a cluster of machines.

4.3.1 Key concepts in Spark

Let us discuss some key concepts in Spark:

- **Use RAM instead of disk**: Spark is designed to use RAM instead of disk for storing intermediate results. This allows Spark to be much faster than traditional MapReduce systems, which use disk for storing intermediate results.

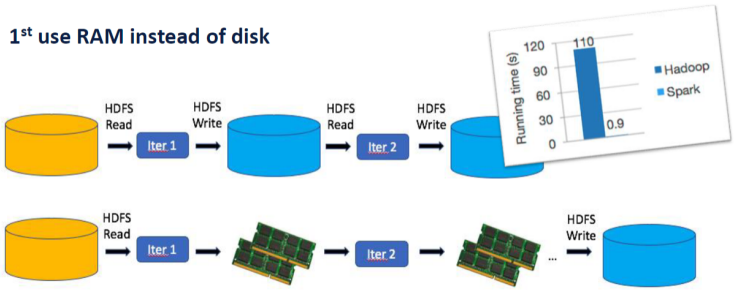


Figure 4.7: Spark in-memory processing

- **Ease of use**: Spark provides a high-level API that makes it easy to write distributed programs. The API is available in multiple languages, including Scala, Java, Python, and R.
- **Generality**: Spark is a general-purpose computing engine that can be used for a wide range of applications, including batch processing, real-time processing, machine learning, and graph processing. This makes Spark a versatile tool that can be used for many different use cases.

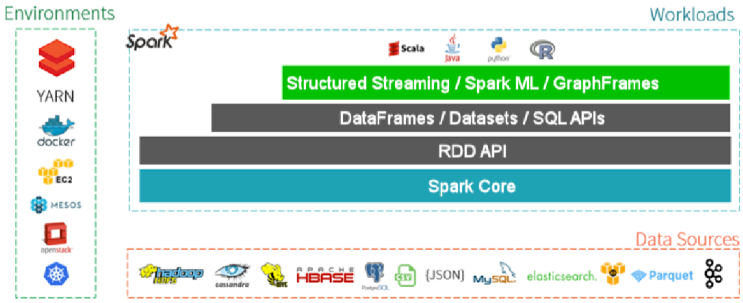


Figure 4.8: Spark generality

4.3.2 Resilient Distributed Datasets (RDDs)

RDDs are the fundamental data structure in Spark. They are fault-tolerant collections of objects that can be operated on in parallel. RDDs are immutable, which means that they cannot be changed once they are created. Instead, transformations are applied to RDDs to create new RDDs.

RDDs can be created from a variety of data sources, including HDFS, S3, Cassandra, and Kafka. Once created, RDDs can be transformed using a variety of operations, including map, filter, reduce, and join. These operations are applied in parallel across the cluster, so that the processing is distributed across multiple machines.

Transformations and actions

In Spark, there are two types of operations that can be applied to RDDs: transformations and actions. Transformations are operations that create a new RDD from an existing RDD, while actions are operations that return a value to the driver program.

Some common transformations in Spark include:

- **map**: Applies a function to each element in the RDD.
- **filter**: Filters the elements in the RDD based on a predicate.
- **reduce**: Aggregates the elements in the RDD using a commutative and associative function.
- **join**: Joins two RDDs based on a key.

Some common actions in Spark include:

- **collect**: Returns all the elements in the RDD to the driver program.
- **count**: Returns the number of elements in the RDD.
- **take**: Returns the first n elements in the RDD.
- **saveAsTextFile**: Saves the RDD to a text file.

4.3.3 Spark at work

Let us see how Spark can be used to process data. A Spark cluster consists of a driver program and multiple worker nodes. The driver program is responsible for coordinating the execution of the job, while the worker nodes are responsible for executing the tasks.

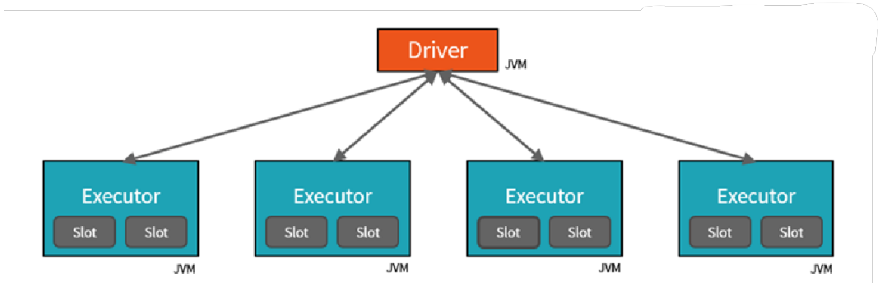


Figure 4.9: Spark components

When a job is submitted to the Spark cluster, the driver program breaks it down into stages, which are then executed in parallel across the worker nodes. Each stage consists of tasks, which are executed in parallel across the worker nodes. The tasks read data from RDDs, apply transformations to the data, and write the results to new RDDs.

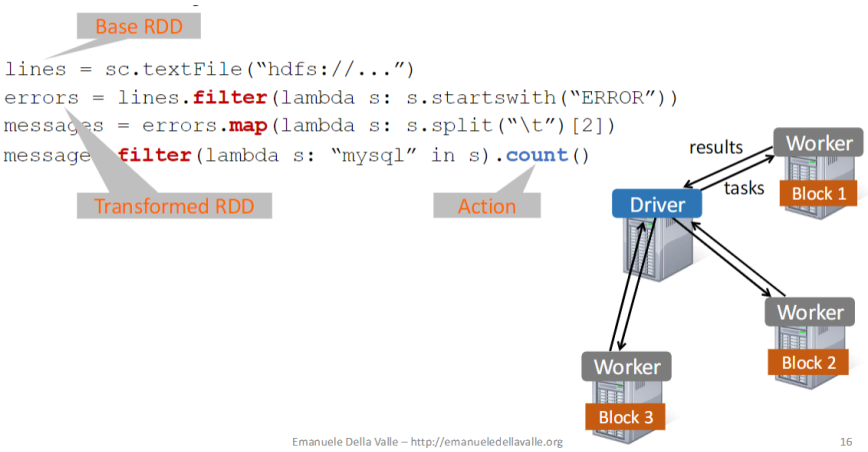


Figure 4.10: Spark execution model

In general, Spark uses a lazy evaluation model, which means that transformations are not executed immediately. Instead, they are queued up and executed when an action is called. This allows Spark to optimize the execution of the job, by combining multiple transformations into a single stage, and reducing the number of shuffles that are needed.

Spark also provides fault-tolerance through lineage, which is a record of the transformations that were applied to an RDD. If a partition of an RDD is lost, Spark can recompute it by replaying the transformations that were applied to the RDD. This allows Spark to recover from failures and continue processing the data.

Spark also provides caching, which allows RDDs to be stored in memory for faster access. This is useful when an RDD is used multiple times in a job, as it avoids the need to recompute the RDD each time it is used. Caching can be done at different levels, including memory only, disk only, and memory and disk.

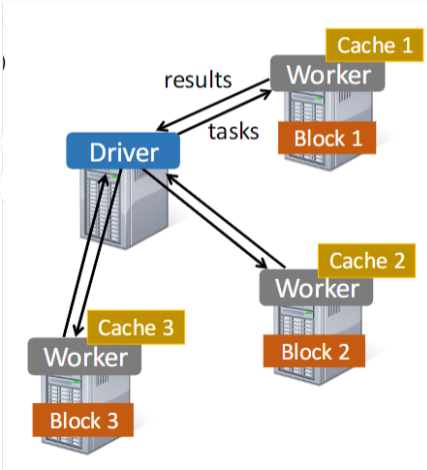


Figure 4.11: Spark caching

4.3.4 DataFrames in Spark

DataFrames are a high-level API for working with structured data in Spark. They provide a more user-friendly interface than RDDs, and are optimized for performance. DataFrames are built on top of RDDs, and provide a more structured way to work with data.

DataFrames can be created from a variety of data sources, including JSON, CSV, and Parquet files. Once created, DataFrames can be queried using SQL, and transformed using a variety of operations, including select, filter, groupBy, and join. These operations are optimized for performance, and are executed in parallel across the cluster.

Let us see an example of how to create a DataFrame in Spark:

```
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("example").getOrCreate()

# Create a DataFrame from a JSON file
df = spark.read.json("data.json")

# Show the first 5 rows of the DataFrame
df.show(5)

# Filter the DataFrame
filtered_df = df.filter(col("age") > 30)

# Create a temporary view to query the DataFrame using SQL
filtered_df.createOrReplaceTempView("filtered_df")

# Query the DataFrame using SQL
query = "SELECT * FROM filtered_df"
result = spark.sql(query)
```

Catalyst and Tungsten

DataFrames in Spark are built on top of two key components: Catalyst and Tungsten. Catalyst is a query optimizer that optimizes the execution of DataFrame operations. It uses a rule-based optimizer to transform the query plan and generate an optimized plan. Tungsten is a memory-centric execution engine that optimizes the execution of DataFrame operations. It uses code generation and memory management techniques to improve the performance of the operations.

Together, Catalyst and Tungsten provide a high-performance engine for working with structured data in Spark. They optimize the execution of DataFrame operations, and provide a more user-friendly interface for working with data.

4.4 Spark Structured Streaming

Spark Structured Streaming provides a fast, scalable, fault-tolerant, end-to-end exactly-once stream processing engine, without the user having to reason about streaming. It is built on the Spark SQL

engine, and provides a high-level API for working with streaming data. Structured Streaming allows users to write streaming queries in the same way as batch queries, and provides the same fault-tolerance and exactly-once guarantees as batch processing.

4.4.1 Programming model

Spark Structured Streaming treats a live data stream as an unbounded table, which is continuously updated as new data arrives. As a result of this idea, users can logically express streaming computation as a Directed Acyclic Graph (DAG) of standard batch-like queries on static tables. Spark will physically run this DAG incrementally and continuously on the streaming data.

It uses the model of a DSMS (Data Stream Management System), as we saw in previous chapters. In every micro-batch, Spark will execute the DAG incrementally to update the final result. For each query, it will mantain a state, which is updated in each micro-batch. We can see this in the following figure:

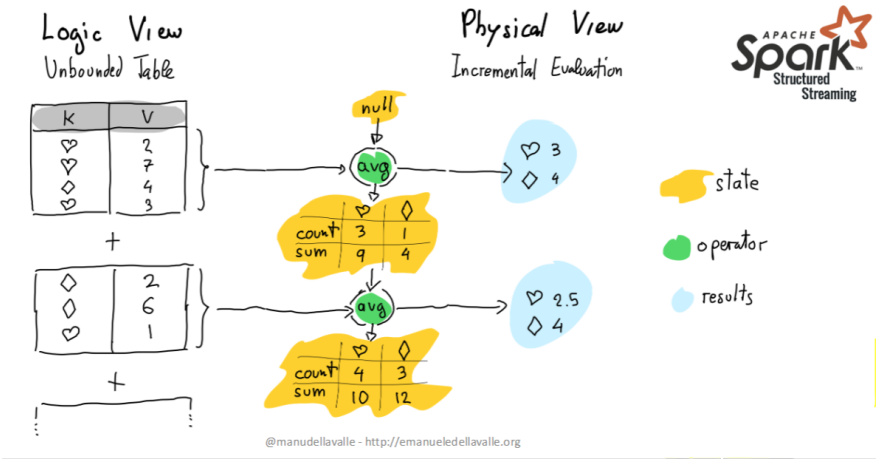


Figure 4.12: Spark incremental evaluation

It is important to note that Spark Structured Streaming does not materialize the entire table, but only the incremental changes to the table. It will read the latest available data from the stream, process it incrementally to update the result, and then discard the data. It will only keep around the minimal intermediate state data as required to update the result, for example, the intermediate counts and average for the above example.

This model is very powerful, as it allows users to express complex streaming computations in a simple and declarative way, and provides the same fault-tolerance and exactly-once guarantees as batch processing. It also allows users to easily integrate streaming and batch processing, as the same code can be used for both.

4.4.2 Creating streaming DataFrames

To create a streaming DataFrame, we can use the `readStream` method of the SparkSession object. It can receive as input the following sources:

- File sources, i.e., a folder in which new files get dropped.
- Kafka sources, i.e., a Kafka topic.
- Socket sources, i.e., a TCP socket. This is only for testing purposes, as it does not provide fault-tolerance.

Let us see an example of how to create a streaming DataFrame in Spark, reading from a Kafka topic:

```
from pyspark.sql import SparkSession

# Create a Spark session
spark = SparkSession.builder.appName("example").getOrCreate()

# Create a streaming DataFrame from a Kafka topic
raw_df = spark.readStream.format("kafka") \
    .option("kafka.bootstrap.servers", "localhost:9092") \
    .option("startingOffsets", "earliest") \
    .option("subscribe", "topic") \
    .load()

# Show the first 5 rows of the streaming DataFrame
raw_df.show(5)
```

In this example, we create a streaming DataFrame from a Kafka topic. We specify the Kafka bootstrap servers, the starting offsets, and the topic to subscribe to. We then show the first 5 rows of the streaming DataFrame.

Note that when extracting data from Kafka, our `raw_df` DataFrame will have the following schema:

- `key`: The key of the message, as a binary blob.
- `value`: The value of the message, as a binary blob.
- `topic`: The topic of the message, as a string.
- `partition`: The partition of the message, as an integer.
- `offset`: The offset of the message, as an integer.
- `timestamp`: The timestamp of the message, as a timestamp.
- `timestampType`: The timestamp type of the message, as an integer.

Here, the column that contains the actual data is the `value` column. We can extract the data from this column using the `select` method, as we will see in the following example:

```
# Extract the data from the value column
casting = ("CAST(key AS STRING)", "CAST(value AS STRING)")
data_df = raw_df.selectExpr(*casting)
```

In this example, we extract the data from the `key` and `value` columns, and cast them to strings. This allows us to work with the data as strings, which is more convenient for processing. But the data of value will be a stringified JSON object, so we need to parse it to a DataFrame. We can do this as follows:

```
from pyspark.sql.functions import from_json

# Parse the data from the value column
schema = "id STRING, name STRING, age INT"
parsed_df = data_df.select(from_json("value", schema).alias("data"))

# Obtain the final DataFrame
final_df = parsed_df.select("data.*")
```

In this example, we parse the data from the `value` column using the `from_json` function, and specify the schema of the data as a string. We then select the fields of the data, and obtain the final DataFrame. This allows us to work with the data as a structured DataFrame, which is more convenient for processing.

4.4.3 Writing streaming DataFrames

We have to remember that Spark is a lazy evaluation engine, so we need to call an action to start the execution of the streaming DataFrame. The most common action to start the execution of a streaming DataFrame is the `writeStream` method. This method allows us to write the streaming DataFrame to a sink, which can be a file, a Kafka topic, or a console.

To do so, we need to specify the following options:

- **Output sink**: The sink to write the data to.
- **Output mode**: The mode to use when writing the data, i.e., append, complete, or update.
- **Query name**: (Optional) Name of the query, which can be used to identify the query in the Spark UI.
- **Trigger interval**: (Optional) The interval at which to trigger the query, i.e., processing time or event time.
- **Checkpoint location**: (Optional) The location to store the checkpoint data, which is used to recover from failures.

We have 3 different output modes:

- **Append**: Only the new rows added to the result table since the last trigger will be written to the sink. This is only supported for queries where rows added to the result table are never updated. For example, queries with only `select`, `where`, `map`, `filter`, `flatmap`, `join`, etc.
- **Complete**: The entire result table will be written to the sink after every trigger. This is dangerous, as it can cause the sink to grow indefinitely, and can lead to out-of-memory errors. It is only supported for aggregation queries.
- **Update**: Only the rows in the result table that were updated since the last trigger will be written to the sink. Supported for aggregation queries and few others, not for joins.

4.4.4 Window operations on Event Time

Spark Structured Streaming supports only logical windows on event time. It allows for declaring tumbling, hopping and session windows, and it treats windows as a particular grouping criteria. For example, suppose we use the example of the fire alarms. We have:

```
temperature_sdf.groupBy(window("TS", "1 minutes", "30 seconds"), "SENSOR")
```

This will group the data in 1-minute windows, with a 30-second slide, and by sensor. This will create a new column with the window start and end times, and group the data by this column.

It is important to note that Spark Structured Streaming does not support physical windows on event time, as it is a continuous processing engine. This means that it does not have the concept of windows in the traditional sense, and does not have the notion of window boundaries. Instead, it treats windows as a logical grouping criteria, and processes the data continuously as it arrives.

4.4.5 Joins

Spark Structured Streaming supports joins between two streaming DataFrames. The following table describes the types of joins that are supported:

Inner	**Supported**, *optionally* specify *watermark* on both sides + *time constraints* for state cleanup
Left Outer	**Conditionally supported**, **must** specify **watermark** on right + **time constraints** for correct results, optionally specify watermark on left for all state cleanup
Right Outer	Conditionally supported, must specify watermark on left + time constraints for correct results, optionally specify watermark on right for all state cleanup
Full Outer	**Conditionally supported**, **must** specify **watermark** on one side + **time constraints** for correct results, optionally specify watermark on the other side for all state cleanup
Left Semi	Conditionally supported, must specify watermark on right + time constraints for correct results, optionally specify watermark on left for all state cleanup

Figure 4.13: Spark joins

Note that, comparing Spark Structured Streaming with EPL, Spark does not support the "followed by" operator (->). This is because Spark is a continuous processing engine, and does not have the concept of events that are followed by other events. To achieve the same result, we need to use a stream-to-stream join with temporal constraints on the events timestamps. For example:

```
join_sdf = smoke_events.join(
    high_temp_events, expr("""
        (sensorTemp == sensorSmoke) AND
        (tsTemp > tsSmoke) AND  ## this is the EPL's -> operator
        (tsTemp < tsSmoke + interval 2 minute)  ## this is the timer:within clause
    """)
)
```

Note that this will not tame the torrent effect, as it will report every event that matches the condition, but this is expected. These two joins are equivalent:

```
# Spark
join_sdf = A.join(B, expr("""
(idA == idB) AND
(tsB > tsA ) AND
(tsB < tsA + interval 2 minute )
"""))
```

```
# EPL
every x=A ->
    every B(id=x.id)
    where timer:within(2 minutes)
```

Spark Structured Streaming also supports stream-to-table joins, which allow us to join a streaming DataFrame with a static DataFrame. This is useful when we have a static lookup table that we want to join with a streaming DataFrame. For example, suppose we have a static DataFrame with the sensor information, and we want to join it with the streaming DataFrame with the temperature data. We can do this as follows:

```
# Create a static DataFrame with the sensor information
sensor_df = spark.read.csv("sensor.csv", header=True)

# Join the streaming DataFrame with the static DataFrame
join_sdf = temperature_sdf.join(sensor_df, "SENSOR")
```

4.4.6 Late arrivals

Late arrivals refer to events that arrive after the expected time window in which they were supposed to be processed. This happens because in real-time streaming, data may experience delays due to temporarily disconnection, network high latency, processing issues, etc.

To handle late arrivals, Spark Structured Streaming provides a watermarking mechanism. Watermarking allows us to specify a threshold on event time, after which we consider the data to be late. This allows us to handle late arrivals by specifying a window on event time, and discarding data that arrives after the window has closed.

To use watermarking, we need to specify the following options:

- **Event time column**: The column that contains the event time.
- **Watermark delay**: The threshold on event time, after which we consider the data to be late.

The system will track the processing time P as the maximum seen event time (last event time), and if if the difference between P and the event time of the event is smaller than the watermark delay, the event is considered on time. Otherwise, it is considered late, and it is discarded.

To use watermarking, we need to specify the event time column and the watermark delay in the query. For example, suppose we have a streaming DataFrame with the temperature data, and we want to handle late arrivals. We can do this as follows:

```
# Create a watermark column
watermarked_df = temperature_sdf.withWatermark("TS", "2 minutes")

# Group the data by window and sensor
windowed_df = watermarked_df.groupBy(window("TS", "1 minutes", "30 seconds"), "
    SENSOR")
```

In this example, we create a watermark column on the `TS` column with a delay of 2 minutes. This allows us to handle late arrivals by specifying a window on event time, and discarding data that arrives after the window has closed. We then group the data by window and sensor, and process the data accordingly.

The following graph shows how watermarking works, in a windowed groped aggregation with update mode:

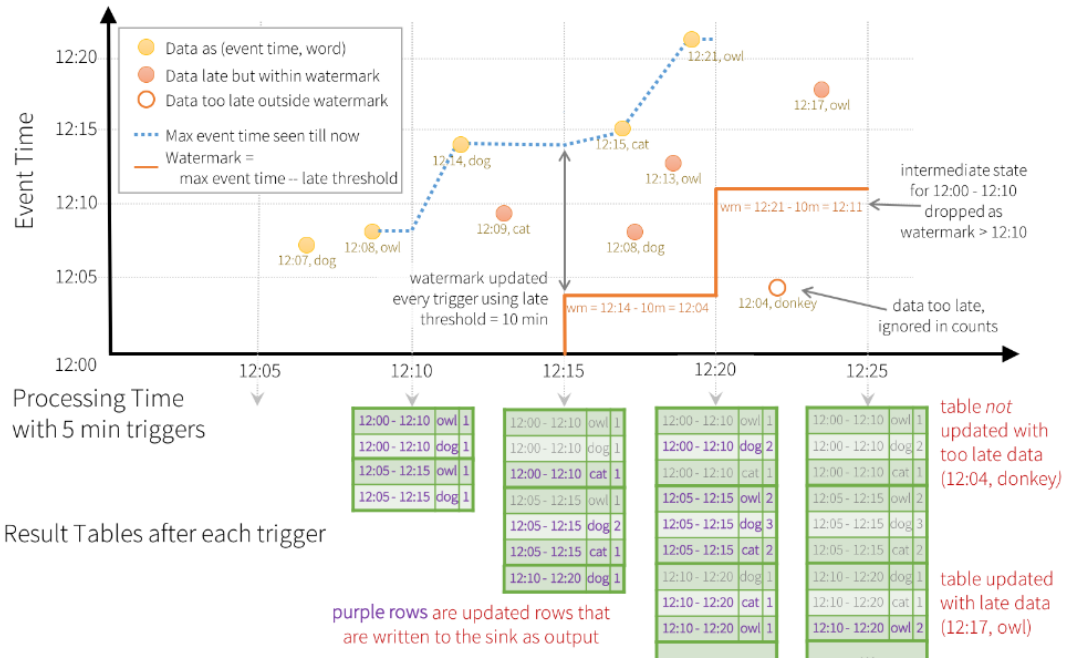


Figure 4.14: Spark watermarking in update mode

And the following graph shows how watermarking works, in a windowed groped aggregation with append mode:

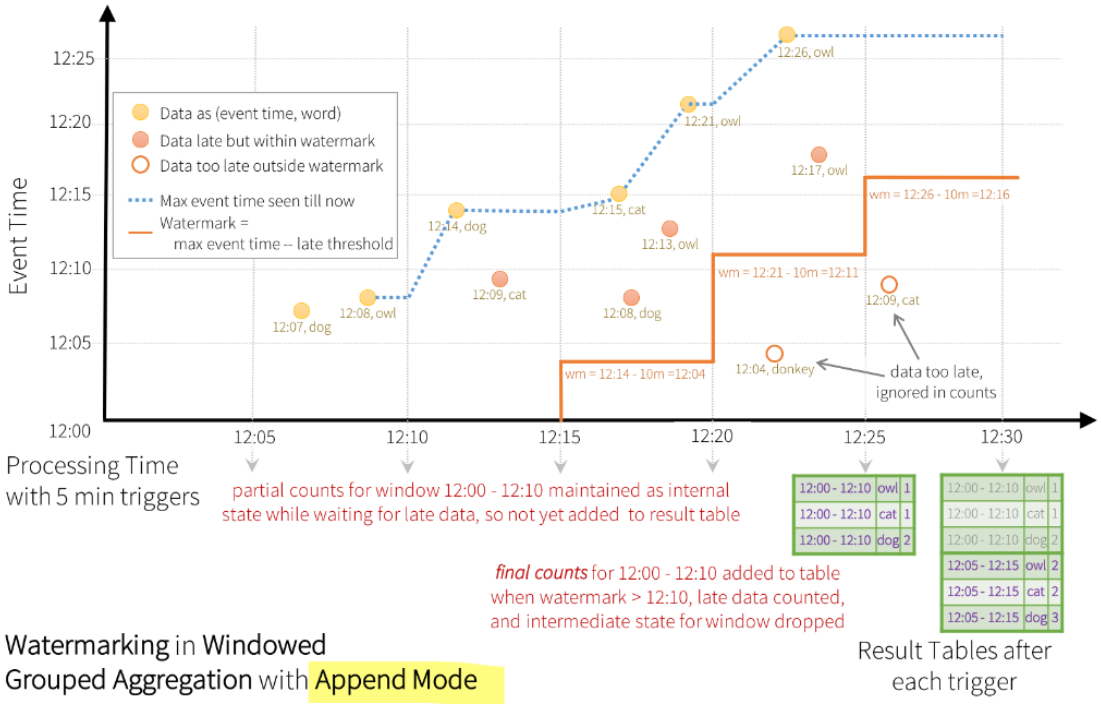


Figure 4.15: Spark watermarking in append mode

Note that a watermark delay of "X hours" guarantees that the engine will never drop any data that is less than X hours late. However, data later that that threshold is not guaranteed to be dropped. This data may or may not get aggregated, and obviously the more delayed the data is, the less likely it is to be aggregated.

4.4.7 Unsupported operations

Spark Structured Streaming does not support all operations that are supported in batch processing. Some of the operations that are not supported include:

- Aggregation without grouping (e.g., counting rows of a table)
- OrderBy operations unless after aggregation and in Complete Output mode
- Limit and take the first N rows
- Distinct operations
- Chaining multiple stateful operations in Update and Complete mode
- Few types of outer joins

Chapter 5

Introduction: Streaming data science

As we know, society as a whole is undergoing a data-centered revolution. The amount of data generated by humans is growing exponentially, and the need to analyze and extract value from this data is becoming more and more important. This is especially true in the context of streaming data, where data is generated continuously and in real-time.

The world is characterized by these 4 factors:

- **Volatility**: The speed at which things change.
- **Uncertainty**: The lack of predictability.
- **Complexity**: The number of factors involved.
- **Ambiguity**: The lack of clarity, or the existence of multiple interpretations.

In this context, the traditional data science approach is not enough. Changes in data can cause the traditional models to become obsolete, and the need for real-time analysis is becoming more and more important. This is where streaming data science comes in. In this and the following chapters, we will explore the world of streaming data science.

Until now, we have assume the data points we receive are independent and identically distributed (i.i.d.). But often in real situations, this assumption doesn't hold:

- **Non identically distributed data**: changes happen, and produce different data distributions for the same data source.
- **Non independent data**: data points may not be independent from each other. Furthermore, in many cases we have a strong temporal dependency between data points.

To deal with these situations, we have many approaches, depending on the conditions of the problem: **Time Series Analysis**, **Streaming Machine Learning** and also **Continual AI**.

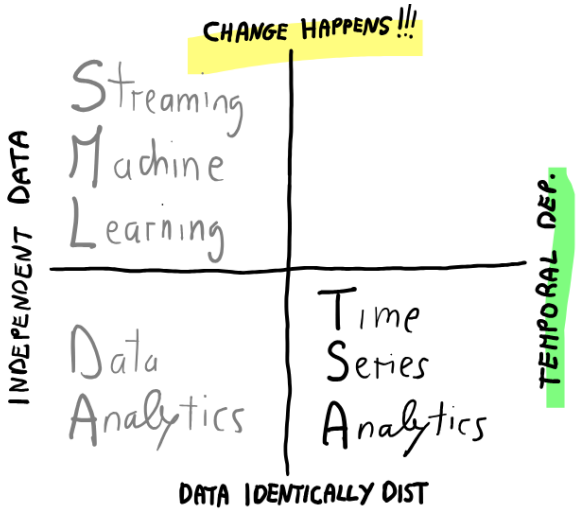


Figure 5.1: Approaches to deal with streaming data science.

5.1 Time Series Analysis

Time series analysis (TSA) is a statistical technique that deals with time-ordered data. It allows us to explain the past and forecast the future of a continuos flow of data **without assuming data independence**.

As the name says, this approach is focused on time series data. A time series is a sequence of observations on one (or more) quantitative variable **regularly collected over time**. In here, we need to make the difference between time series and events:

- In **events**, the phenomenon happens and it is observed **irregularly** (the observations are note necessarily regularly spaced in time).
- In **time series**, we monitor the phenomenon **regularly** (the observations are regularly spaced in time).

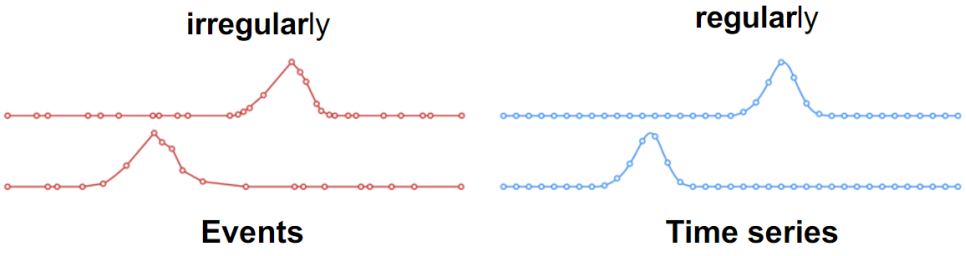


Figure 5.2: Time series vs events

Note that we can convert events into time series by **windowing** the data. For example, the average trade price of Apple stock every 10 minutes over the course of a day; or the average response

time for requests in an application over 1 minute intervals.

TSA has two main purposes:

- **Modeling (descriptive analysis)**: We want to understand the underlying structure of the data, and how it changes over time. Specially, we want to extract the **trend** and **seasonality** of the data.
- **Forecasting (predictive analysis)**: We want to predict the future values of the time series, to make decisions based on this predictions.

Note also that TSA is **not inherently adaptive** nor focused on real-time learning. Models are (almost always) retrained from scratch when new data is available. For this, we need to use **Streaming Machine Learning**.

5.2 Streaming Machine Learning

Streaming Machine Learning (SML) is a subfield of machine learning that deals with data that is generated continuously and in real-time. It allows us to learn one data at a time from a continuous flow of data **without assuming identically distributed data**.

In SML, the type of data we receive arrives in a continuous stream, potentially in real-time, and we assume that data points are **independent**, but the **data distribution can change over time**.

The purpose of SML is to **extract the insights** from the data as it arrives, and discard the data points once they have been used. Often, we have a **low-latency** requirement and a **minimal computation footprint**. Classification tasks are the main application domain of SML, but also forecasting and clustering are in scope.

Notice that data can arrive in a continuous stream as easy as a **balanced** one, or as complex as a **dynamically imbalanced** one.

5.2.1 Techniques in SML: Overview

Some techniques used in SML are the **Hoeffding trees**: these are decision trees **built incrementally**:

- A data point at a time.
- Memory stores only the model.

For these trees, we have theoretical guarantees that the model will converge to the tree that would have been built if all the data was available at once.

To adapt to the changing data distribution, we can use **data and concept drift detection** techniques. These techniques allow us to detect when the data distribution has changed, and adapt the model accordingly:

- We can monitor the input distribution: is there a statictically significant difference between the distribution of the recent data and the distribution of the old data? This detects **data drifts**.
- We can monitor the prediction error: is there a statistically significat growth between the recent and the old errors? This detects **concept drifts**.

By adding these techniques to the model, we can adapt to the changing data distribution and keep the model up-to-date. We obtain the **Hoeffding Adaptive Trees (HAT)**, that work as follows:

- When a concept drift is detected, start growing alternative branches.
- When the alternative branches are better than the current ones, replace them.

We also have techniques like **ensembles**, that combine multiple models to improve the performance of the model. We have:

- **Bagging**: We train multiple models on different subsets of the data, and combine the predictions of these models.
- **Boosting**: We train multiple models sequentially, and each model is trained to correct the errors of the previous model.

5.2.2 Learning characteristics and challenges

SML is **adaptive** and designed to handle **changing data distributions** and **concept drifts**. Ir is well-suited for applicationts requiring **immediate response** to incoming data changes.

However, SML has this immediate reaction to changes at a sometimes high cost: **it forgets**. This is, the model is not able to remember the past data, and it can only learn from the data that is currently available. This can lead to **catastrophic forgetting**, where the model forgets the past data and becomes obsolete.

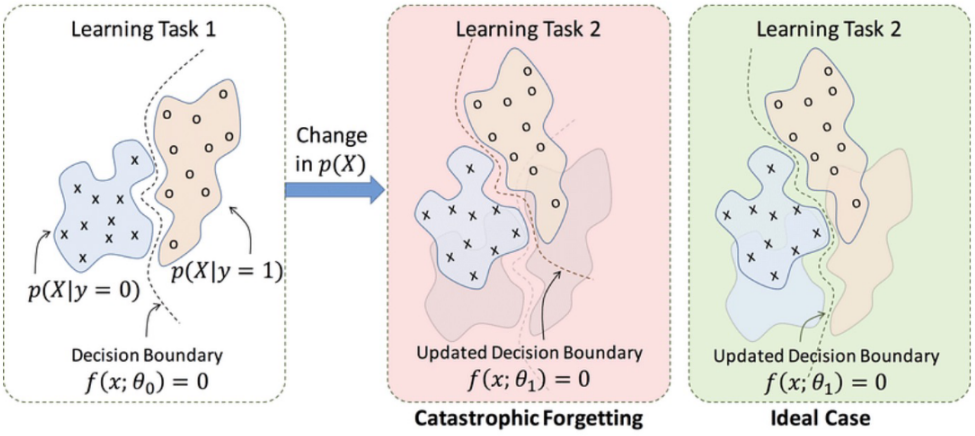


Figure 5.3: Catastrophic forgetting

This illustrates the **plasticity-stability dilemma** in AI:

- **Plasticity**: The ability of the model to adapt to new data.
- **Stability**: The ability of the model to retain the knowledge of the past data.

If a model is too plastic, it will forget the past data and become obsolete. If a model is too stable, it will not adapt to the new data and will not be able to learn from it.

To solve this dilemma, we need to use **Continual AI**.

5.3 Continual AI

Continual AI (CAI) is a subfield of AI that allows us to build a model that learns from a sequence of training episodes intermixed with situations that require applying (recently or prevously) learned skills.

In this case, data arrives in (often manually drafted) **experiences**, where training and test episodes **intermix**, assuming that data points are **independent and identically distributed only within each experience**. Data in two experiences are normally distributed differently.

As an example, we have the training of a robotic arm to perform various tasks. The following diagram shows the key features of CAI.

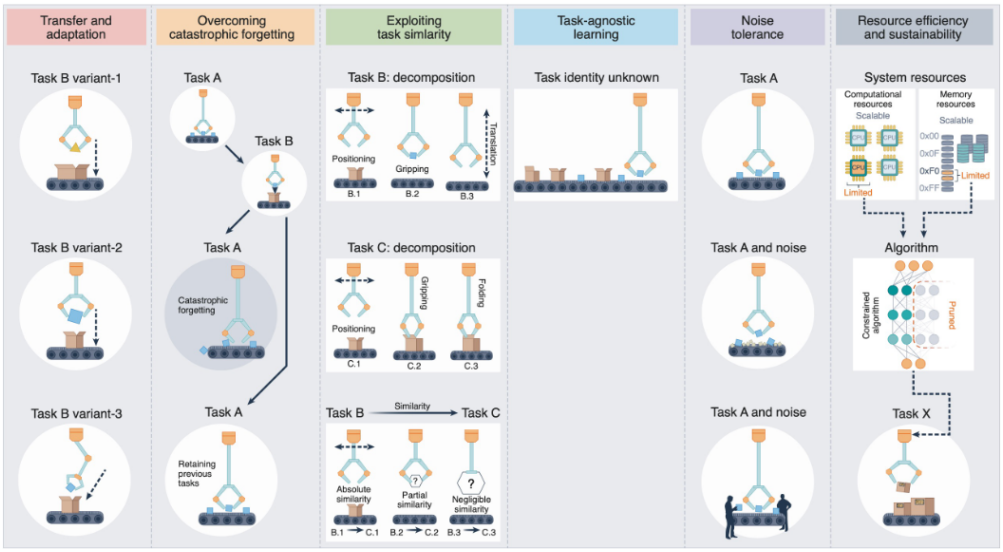


Figure 5.4: Continual AI

CAI is primarily used to **avoid catastrophic forgetting** while learning new experiences. It balances between **plasticity** and **stability**, potentially via task similarity. Eventually, it also learns when the task identity is unknown, tolerates noise and it is more prone to resource constraint environments.

5.3.1 Techniques in CAI: Overview

In CAI, we will take a look at two main techniques:

- **Rehearsal strategies (episodic replay)**: alleviate catastrophic forgetting by:
  - storing past data points in an episodic memory system.
  - regularly replaying past data points with real ones from the new task.
  - learning from the obtained mix of replayed and real data points.
- **Architectural strategies (neurogenesis)**: alleviate catastrophic forgetting by:
  - choosing specific deep learning architectures, layers, activation functions, etc.
  - weight-freezing and dynamic architectures (e.g., adding new layers).

5.3.2 Learning characteristics

Continual Learning ensures that models can **adapt to new information** while **retaining previously learned knowledge**. It focuses on long-term model stability.

5.4 Summary

In summary, streaming machine learning is designed for real-time data stream processing and model adaptation, time series analytics is primarily retrospective and focuses on historical data analysis, while continual learning is concerned with the long-term adaptation of models to new data while avoiding forgetting essential knowledge.

The choice of approach depends on the specific requirements of the application and the nature of data being analysed.

Chapter 6

Time Series Analysis

6.1 First foundational concept: Stationarity

"If a time series is stationary, it is predictable".

A **stationary** time series is one whose properties **do not depend on the time** at which the series is observed. More specifically, we say tha a time series is stationary if it satisfies the following three conditions:

- The **mean** of the series is **constant** over time.
- The **variance** of the series is **constant** over time.
- No repetitive patterns in the series, i.e., the series is **not seasonal**.

For example, the white noise, i.e., a sequence of random numbers with a mean of zero and a constant variance, is a stationary time series. If you predict 0, you will minimize the error, which is proportional to the variance.

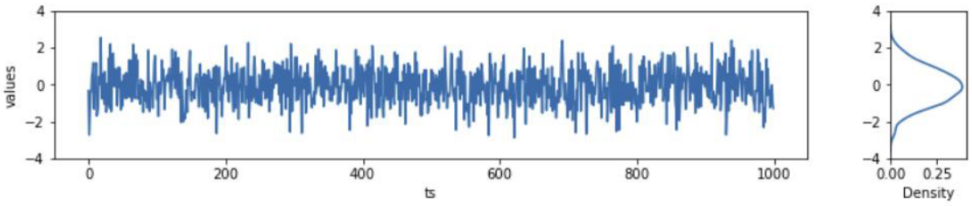


Figure 6.1: White noise

Formally: Let $\{X_t\}$ be a stochastic process and let $F_X(t_{1+\tau}, ..., t_{k+\tau})$ represent the cumulative distribution function of the unconditional (i.e., with no reference to any particular starting value) joint distribution of $\{X_t\}$ at times $t_1 + \tau, ..., t_k + \tau$. Then, $\{X_t\}$ is said to be **strictly stationary**, **strongly stationary** or **strict-sense stationary** if, for all k and all τ:

$$F_X(t_1, ..., t_k) = F_X(t_{1+\tau}, ..., t_{k+\tau}) \tag{6.1}$$

6.2 Decomposition and detrending

Stationarity is generally a **good property** for a time series to have, as it makes it easier to model and predict. However, very few time series in the real world are actually stationary. Normally, a time series includes:

- a long-term trend (including aperiodic cycles).
- one or more seasonal patterns.

The question is: can we try to remove what causes non-stationarity? Yes, by **time series decomposition**.

Time series decomposition is a mathematical procedure which transforms a time series into multiple different time series, each representing a different component of the original one. Generally, we want to decompose a time series into three components:

- **Trend**: the long-term progression of the series.
- **Seasonality**: the repeating short-term cycle in the series.
- **Residuals**: the random noise left after the trend and seasonality have been removed. Hopefully, this is a stationary time series.

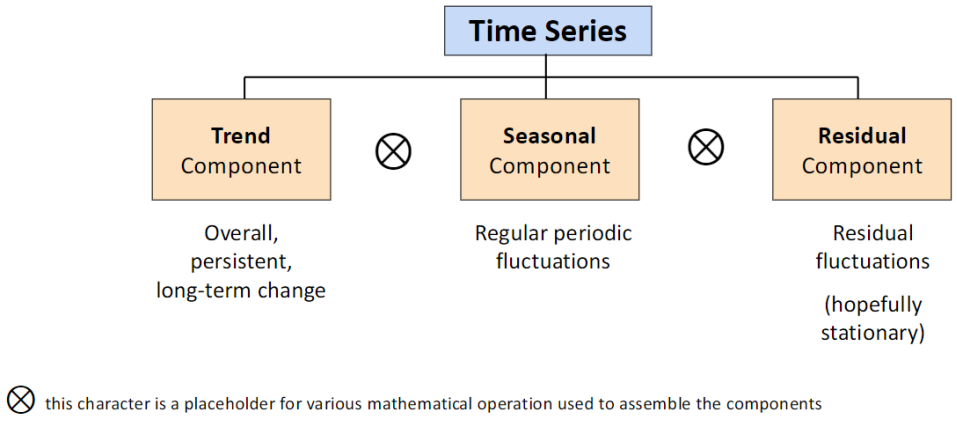


Figure 6.2: Time series decomposition

6.2.1 Additive and multiplicative decomposition

We have two main methods to decompose a time series:

- **Additive decomposition**:

$$X_t = m_t + s_t + Y_t$$

- **Multiplicative decomposition**:

$$X_t = m_t \cdot s_t \cdot Y_t$$

where X_t is the original time series, m_t is the trend component, s_t is the seasonal component, and Y_t is the residual component.

Which one is better? Let us see a decomposition example:

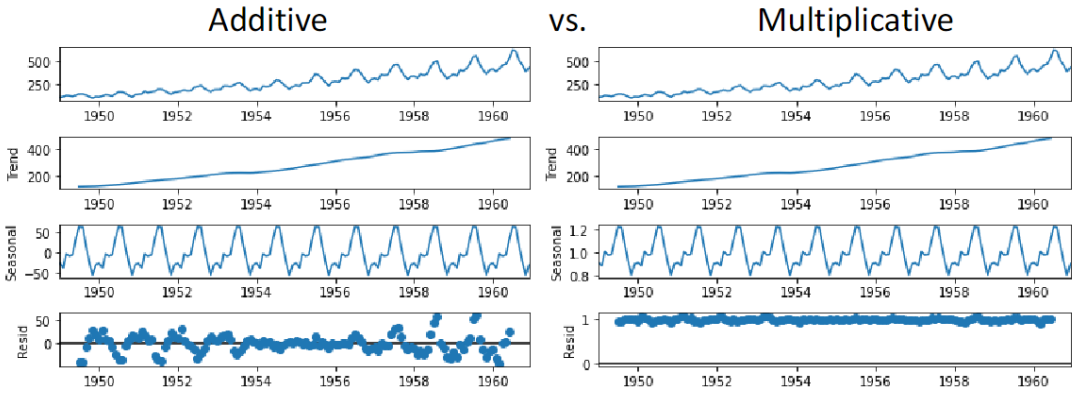


Figure 6.3: Additive vs multiplicative decomposition

In this case, the **multiplicative decomposition** is better because, in the end, we obtain a residual that is more stationary than the one obtained with the additive decomposition, which still shows some seasonality.

In general, we choose between additive and multiplicative decomposition based on the following criteria:

- **Additive** decomposition: it is useful when the seasonal pattern is relatively independent of the trend. E.g., when the seasonal variation is constant over time.
- **Multiplicative** decomposition: it is useful when the seasonal pattern is correlated with the trend. E.g., when it increases overtime as the trend increases.

Now, let us see how can we determine these decompositions. We have two main simplified time series models, based on how many components we want to include in the decomposition:

- **Non-seasonal** decomposition model with **trend**.
- Decomposition model with **trend** and **seasonality**.

Notice that these methods are additive decompositions. We can obtain the multiplicative decomposition by taking the logarithm of the time series and then applying the additive decomposition:

$$\log(X_t) = \log(m_t) + \log(s_t) + \log(Y_t)$$

$$\Rightarrow X_t = m_t \cdot s_t \cdot Y_t$$

6.2.2 Non-seasonal decomposition model with trend

This model only considers the trend component. There are two basic methods for estimating and removing the trend from a time series:

1. **Trend elmimination by differencing**

2. **Trend estimation by model fitting and removal**

Trend elimination by differencing

Differencing of a time series $\{X_t\}$ in discrete time t is the transformation of the series to a new time series $\{D_t\}$, where the values are the differences between consecutive observations:

$$d_t = x_t - x_{t-1}$$

If a trend is linear, differencing once is sufficient to remove it. If instead of a linear trend, we have a quadratic trend, we need to difference the series twice. More generally, if a trend can be model with a polynomial of degree n, then we need to difference the series n times.

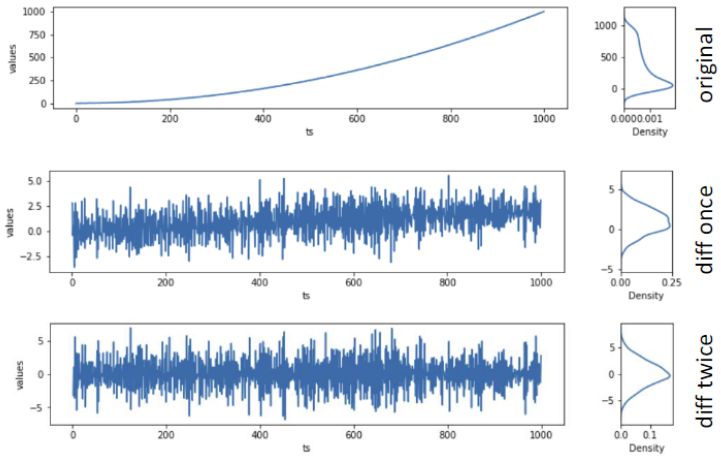


Figure 6.4: Removing a quadratic trend by differentiation

Trend estimation by model fitting and removal

In this method, we fit a model to the time series and then remove the model from the series. The most common models are:

- **Linear regression**: we fit a linear model to the time series and then subtract the model from the series.

- **Polynomial regression**: we fit a polynomial model to the time series and then subtract the model from the series.

Combine the methods

We can combine the previous tow methods, to better remove any trends from the time series. We can do as follows:

1. Differentiate the time series.
2. Observe if the series still show a trend.
3. If there is still a trend, detrend by fitting a regresion model (linear or polynomic, depending on the observed trend).

6.2.3 Decomposition model with trend and seasonality

This method also includes a seasonal component in the decomposition. There are 3 basic methods for estimating and removing the trend and seasonality:

1. **Differencing:** we differentiate one or more times to remove the trend. Then we perform a "seasonal differencing".
2. **Filtering:** we estimate and remove the trend by using a "centered" moving average. Then we estimate and remove the seasonal component using periodic averages.
3. **Joint-fit method:** fitting a combined polynomial and dynamic harmonic regression.

Seasonal differencing

Seasonal differencing of a time series $\{X_t\}$ in discrete time t given the seasonality's period d is the transformation of the series into a new time series $\{S_t\}$ where the values are the differences between the value of $\{X_t\}$ at time t and the value of $\{X_t\}$ a period d before:

$$s_t = x_t - x_{t-d}$$

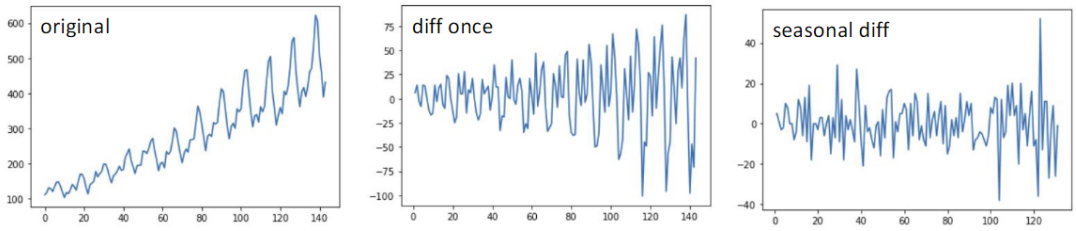


Figure 6.5: Seasonal differentiation of a time series

Filtering: centered moving and periodic averages

Given the seasonality period d:

- If d is even, the "centered" moving average is defined as:

$$\hat{m}_t = (0.5x_{t-d} + x_{t-d+1} + ... + x_{t+d-1} + 0.5x_{t+d})/d$$

- If d is odd, then it is defined as:

$$\hat{m}_t = (x_{t-d} + x_{t-d+1} + ... + x_{t+d-1} + x_{t+d})/d$$

Note that there are no values for either the first d or the last d points, because we do not have enough observations on either side to define the moving average for those values of t. Also, note that the "centered" moving average is different from the "normal" moving average.

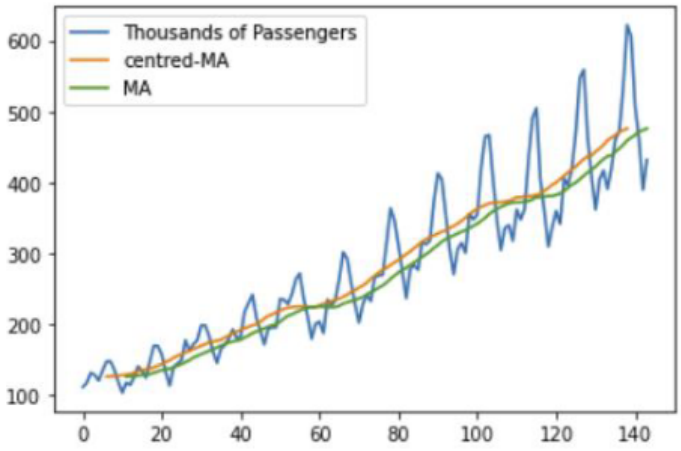


Figure 6.6: Centered moving average example

Now, we need to estimate the seasonal component. For that, we use **periodic averages**:

1. Divide the detrended value in seasons of length d.

2. Compute the seasonal component values w_k by averaging each of the d points of the season $k = 1, ..., d$.

3. Compute the adjusted seasonal component values s_k to ensure that they add to zero:

$$\hat{s}_k = w_k - d^{-1} \sum_{j=1}^{d} w_j, \quad k = 1, ..., d$$

4. String together the adjusted seasonal component values in a sequence.

5. Replace the sequence for each season.

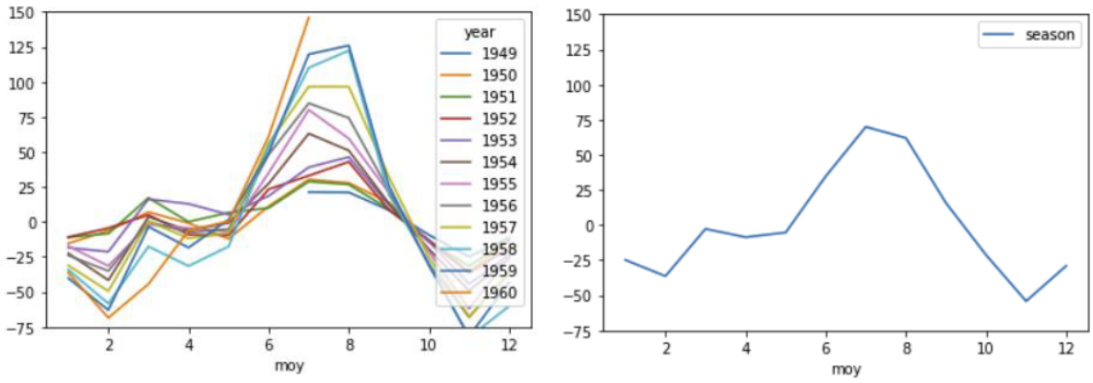


Figure 6.7: Periodic averages example

Joint-fit method

This method is based on fitting a combined polynomial and dynamic harmonic regression to estimate and then remove the trend and seasonal components simultaneously. The method is as follows:

$$X_t = m_t + s_t + Y_t =$$

$$= (\beta_0 + \beta_1 t + \beta_2 t^2) + \left[\sum_{j=1}^{k} (\alpha_j \cos(\lambda_j t) + \gamma_j \sin(\lambda_j t))\right] + Y_t$$

where $\beta_0, \beta_1, \beta_2$ are the coefficients of the polynomial regression model, and $\alpha_j, \gamma_j, \lambda_j$ are the coefficients of the harmonic regression model.

The following diagram represents the anatomy of the harmonic function:

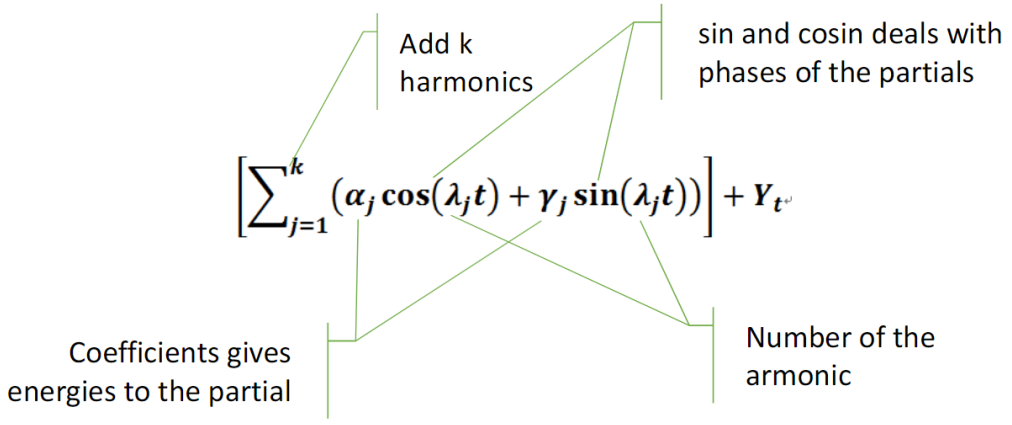


Figure 6.8: Anatomy of the harmonic function

6.3 Forecasting baselines

Once we have decomposed the time series into its components, we can use the residuals to forecast the future values of the series, assuming that the residuals are stationary.

How far can we predict?

- **Short-term forecasting**: a one-step-ahear is a forecast of the next observation only.
- **Medium and long-term forecasting**: a multi-step-ahead forecast is a forecast of the next n observations.

What can we predict?

- **Trend:** we can predict the long-term progression of the series trend.
- **Seasonality:** we can predict in medium-term, we don't have the certainty that the seasonality will repeat identically in the future.
- **Residuals:** we can predict the random noise left after the trend and seasonality have been removed. This is the most difficult component to predict, and we can predict it only in the short-term.

The overall forecasting process is based on decomposing the time series $\{X_t\}$ into its components, then we predict the residuals, add the seasonality component and finally add the trend component based on the fitting models.

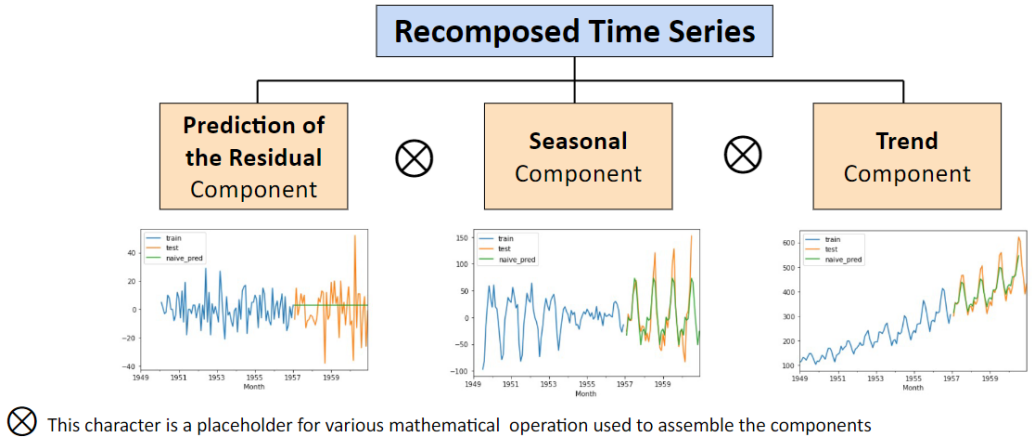


Figure 6.9: Recomposition of the time series

6.3.1 Error metrics

To evaluate the quality of the forecast, we need to compare the forecasted values with the actual values on the test set. The most common error metrics are:

- **Mean Absolute Error (MAE)**:

$$MAE = \frac{1}{n}\sum_{i=1}^{n}|y_i - \hat{y}_i|$$

- **Mean Absolute Percentage Error (MAPE)**:

$$MAPE = \frac{100}{n}\sum_{i=1}^{n}\left|\frac{y_i - \hat{y}_i}{y_i}\right|$$

- **Mean Squared Error (MSE)**:

$$MSE = \frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2$$

- **Root Mean Squared Error (RMSE)**:

$$RMSE = \sqrt{MSE} = \sqrt{\frac{1}{n}\sum_{i=1}^{n}(y_i - \hat{y}_i)^2}$$

How do we choose which error metric to use? It depends on the context of the problem. The following table shows the benefits and limitations of each error metric:

Error Metric	Benefit	Limitation
Mean Absolute Error (MAE)	Easy to calculate, understand and interpret.	Sensitive to outliers; may not penalize large errors sufficiently.
Mean Absolute Percentage Error (MAPE)	Provides a percentage error representation; can be used to make comparisons across different datasets.	Susceptible to division by zero, i.e., when actual values are close to zero. Biased towards actual values. Asymmetric.
Mean Square Error (MSE)	Penalizes larger errors more significantly than small errors	Units are squared, which might be less intuitive in some contexts.
Root Mean Square Error (RMSE)	Provides an interpretable scale for MSE like the original data.	Sensitive to outliers like MSE; not robust against extreme values.

Figure 6.10: Error metrics

6.3.2 Basic forecasting methods

The residuals are the random noise left after the trend and seasonality have been removed. We can predict the residuals using the following baseline methods:

- **Naive approach:** The only important information is the last value of the residuals. We can predict the residuals as the last value of the residuals in the training set:

$$\hat{y}_{t+1} = y_t$$

- **Mean approach:** We can predict the residuals as the mean of the residuals in the training set:

$$\hat{y}_{t+1} = \frac{1}{n}\sum_{i=1}^{n} y_i$$

We also have a more sophisticated method to predict the residuals, which is the **last-k average** approach. This method appears to be in the middle between the naive and mean approaches. The method is as follows:

$$\hat{y}_{t+1} = \frac{1}{k}\sum_{i=1}^{k} y_{t-i}$$

The problem with this method is that we need to choose the value of k, which is a hyperparameter. It is hard to set a value that clearly separates the data to "forget" and the data to use for the prediction.

We ask the question: is there a way to consider all the data in the training set while still introducing a "forgetting" mechanism? Yes, we can use the **exponential smoothing** method.

6.3.3 Exponential smoothing

Exponential smoothing is a **stationary time series** forecasting method for univariate data. It is a simple yet effective method for forecasting. The method is based on the idea of weighting the most recent observations more heavily than the older observations.

The method is as follows:

$$\hat{y}_{t+1} = \alpha y_t + (1-\alpha)\hat{y}_t$$

where α is the smoothing factor, which is a value between 0 and 1. The value of α determines the weight of the most recent observation. The closer α is to 1, the more weight is given to the most recent observation. The closer α is to 0, the more weight is given to the previous forecast.

Often, it is easier to set α as the number of points (span) with a considerable impact on the forecast:

$$\alpha = \frac{2}{span + 1}$$

Notice that this is a **streaming algorithm**, which means that we can update the forecast only as fast as new observations become available. This is because the forecast at time $t+1$ depends on the forecast at time t and the true observation at time t.

Also, if we rearrange the formula, we get:

$$\hat{y}_{t+1} = \alpha y_t + (1-\alpha)\hat{y}_t = \hat{y}_t + \alpha(y_t - \hat{y}_t)$$

This shows that the next prediction is the sum of the current prediction and a correction term. The correction term is propotional (with factor α) to the current error, which is the difference between the current observation and the current prediction.

Also notice that we need a starting value for the forecast, then we have:

$$\hat{y}_1 = \ell_0$$

$$\hat{y}_t = \alpha y_{t-1} + (1-\alpha)\hat{y}_{t-1}$$

where ℓ_0 is the starting value of the forecast. We can set ℓ_0 as the first value of the residuals in the training set.

What about directly forecasting a time series with trend and seasonality? We can extend the exponential smoothing method to include only the trend component or both the trend and seasonal components:

- Residual + trend: **Holt's linear trend method**, also known as **double exponential smoothing**.
- Residual + trend + seasonality: **Holt-Winters' method**, also known as **triple exponential smoothing**.

6.3.4 Double exponential smoothing: Holt's linear method

Holt's linear method is an extension of the exponential smoothing method tha **adds support for trends** using an additional smoothing factor β to control the decay of the influence of the change in trend.

This method supports trends that change in different ways:

- **Additive**, when the trend is linear:

$$\begin{aligned} \text{Forecast equation: } & \hat{y}_{t+h} = \ell_t + h \cdot b_t \\ \text{Level equation: } & \ell_t = \alpha y_t + (1-\alpha)(\ell_{t-1} + b_{t-1}) \\ \text{Trend equation: } & b_t = \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1} \end{aligned}$$

- **Multiplicative**, when the trend is exponential. This only changes the forecast equation:

$$\hat{y}_{t+h} = \ell_t \cdot (b_t)^h$$

Let us take a look at the anatomy of the Holt's linear method:

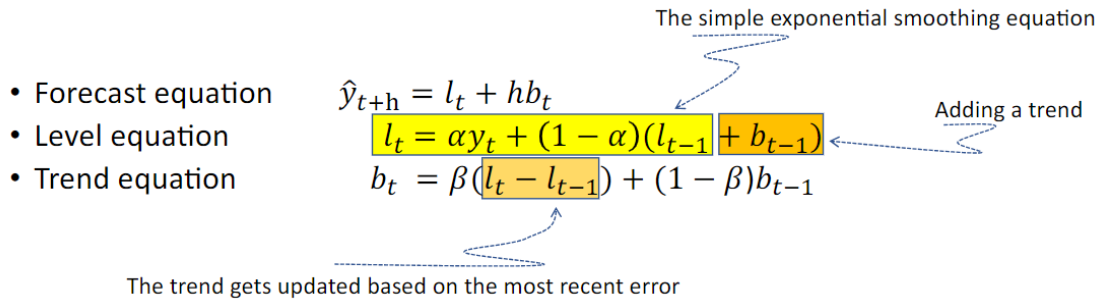


Figure 6.11: Anatomy of Holt's linear method

Note that the trend can vary adaptively over time. The trend smoothing factor β is used to control the decay of the influence of the change in trend. The closer β is to 1, the more weight is given to the most recent trend. The closer β is to 0, the more weight is given to the previous trend.

6.3.5 Triple exponential smoothing: Holt-Winters' method

Holt-Winters' method is an extension of the Holt's linear method that **adds support for seasonality** using an additional smoothing factor γ to control the decay of the influence of the change in seasonality.

As with the trend, the seasonality may be modelled as an **additive or multiplicative** process for a linear or exponential change in the seasonality.

The equations for the additive method, for a time series with seasonality of period d are:

$$\begin{aligned}
\text{Forecast equation: } & \hat{y}_{t+h} = \ell_t + h \cdot b_t + s_{t-d+h} \\
\text{Level equation: } & \ell_t = \alpha(y_t - s_{t-d}) + (1-\alpha)(\ell_{t-1} + b_{t-1}) \\
\text{Trend equation: } & b_t = \beta(\ell_t - \ell_{t-1}) + (1-\beta)b_{t-1} \\
\text{Seasonal equation: } & s_t = \gamma(y_t - \ell_{t-1} - b_{t-1}) + (1-\gamma)s_{t-d}
\end{aligned}$$

Let us take a look at the anatomy of the Holt-Winters' method:

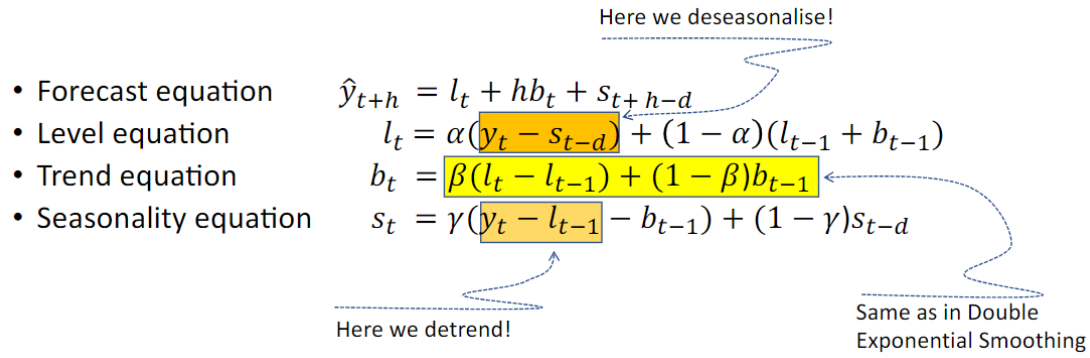


Figure 6.12: Anatomy of Holt-Winters' method

Note that both the trend and seasonality can vary adaptively over time. The trend smoothing level β and the seasonality smoothing level γ are used to control the speed of adjusting the trend and seasonality, respectively. The only fixed parameter is the seasonality period d.

6.4 Second foundational concept: Temporal dependence

The second foundational concept in time series analysis is **temporal dependence**. Temporal dependence is the idea that the value of a time series at time t is dependent on the values of the time series at previous times.

To understand temporal dependence, we need to introduce the following concepts:

- **Correlation**
- **Autocorrelation**
- **Partial autocorrelation**

6.4.1 Correlation

Correlation is a measure of the strength and direction of a linear relationship between two variables. In other words, measures the degree to which two variables X_1 and X_2 move in coordination with each other.

The correlation coefficient is a value between -1 and 1. The closer the value is to 1, the stronger the positive correlation, meaning that the two variables move in the same direction. The closer the value is to -1, the stronger the negative correlation, meaning that the two variables move in opposite directions. A value of 0 indicates no correlation.

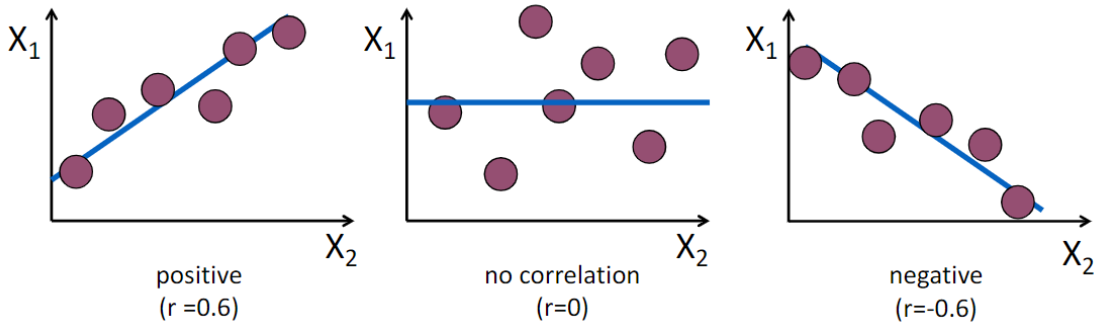


Figure 6.13: Correlation examples

The correlation coefficient is defined as:

$$\rho_{X_1,X_2} = Corr(X_1, X_2) = \frac{Cov(X_1, X_2)}{\sigma_{X_1}\sigma_{X_2}}$$

where $Cov(X_1, X_2)$ is the covariance between X_1 and X_2, and σ_{X_1} and σ_{X_2} are the standard deviations of X_1 and X_2, respectively.

6.4.2 Autocorrelation

Autocorrelation is a measure of the strength and direction of a linear relationship between a time series and a lagged version of itself. In other words, it measures the degree to which the value of a time series at time t is dependent on the value of the time series at time $t-k$, where k is the lag.

The **autocorrelation function (ACF)** is a plot of the autocorrelation of a time series against the lag. The ACF is used to identify the presence of temporal dependence in a time series.

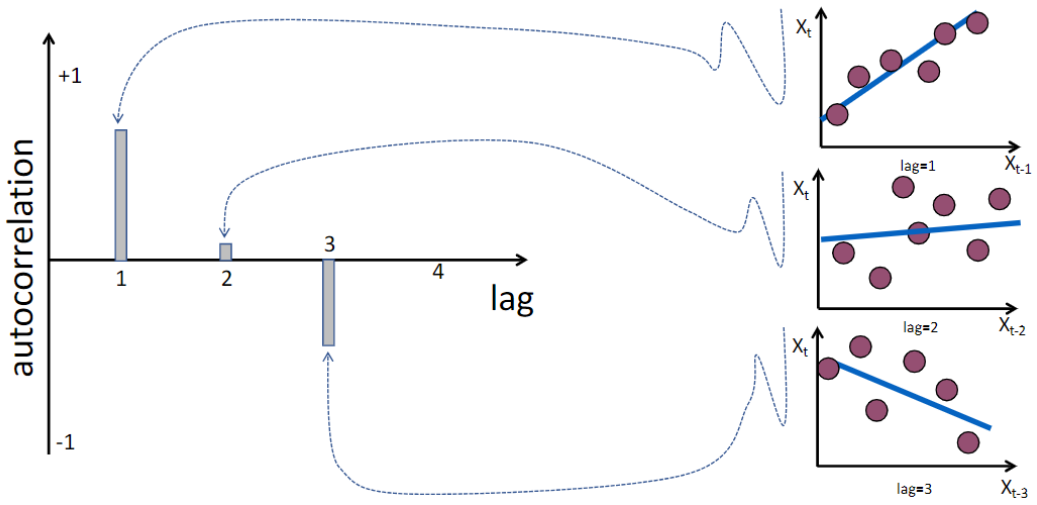


Figure 6.14: Autocorrelation function

The autocorrelation coefficient at lag k is defined as the correlation between the time series and a lagged version of itself:

$$\rho_k = Corr(X_t, X_{t-k})$$

Let us see an example of the autocorrelation function, for a time series representing the number of passengers on an airline flight:

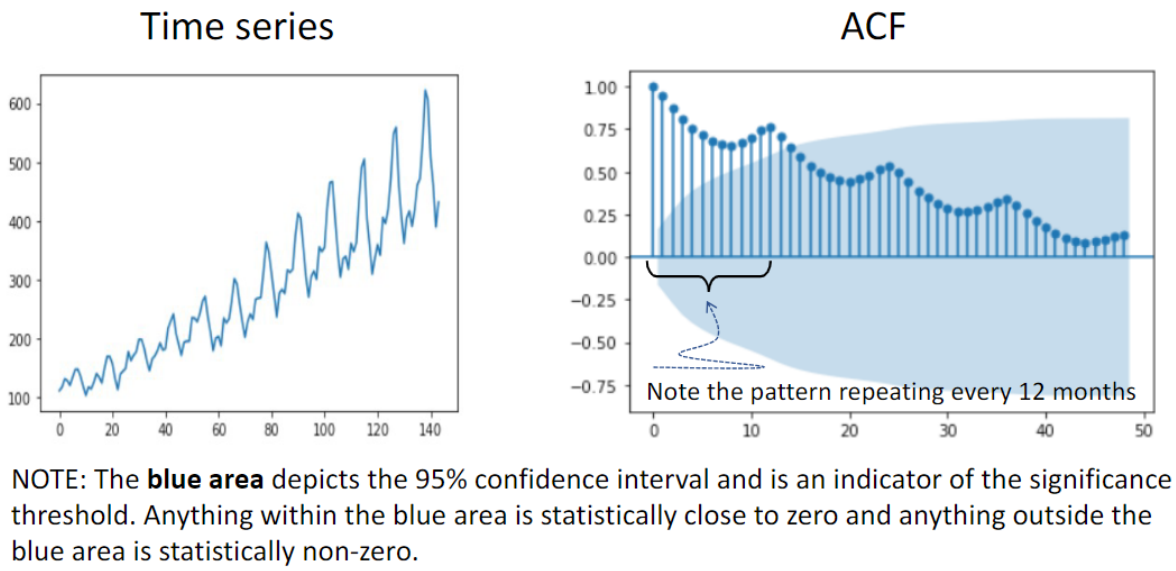


Figure 6.15: Autocorrelation function example

This example depicts the presence of a seasonality with a period of 12 months. The autocorrelation function shows a peak at lag 12, which indicates that the number of passengers in a given month is correlated with the number of passengers 12 months earlier.

6.4.3 Partial autocorrelation

Partial autocorrelation is a measure of the strength and direction of a linear relationship between a time series and a lagged version of itself, after accounting for the effect of other lags. In other words, it measures the degree to which the value of a time series at time t is dependent on the value of the time series at time $t-k$, after removing the effect of the intermediate lags $t-1, t-2, ..., t-k+1$.

To calculate the partial autocorrelation coefficient at lag k, we need to fit a linear regression model to the time series using the intermediate lags as predictors. The partial autocorrelation coefficient is the coefficient of the lag k in the regression model. In other words, if we compute the OLS regression of X_t on $X_{t-1}, X_{t-2}, ..., X_{t--k}$:

$$X_t = \beta_0 + \beta_1 X_{t-1} + \beta_2 X_{t-2} + ... + \beta_k X_{t-k} + \epsilon_t$$

then the partial autocorrelation coefficient at lag k is $\rho_k = \beta_k$.

The **partial autocorrelation function (PACF)** is a plot of the partial autocorrelation of a time series against the lag. The PACF is used to identify the presence of temporal dependence in a time series.

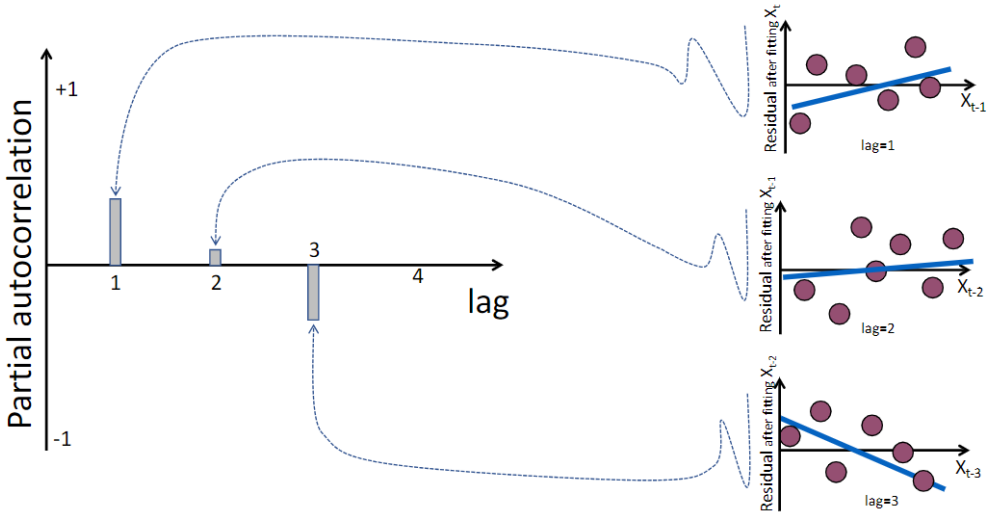


Figure 6.16: Partial autocorrelation function

Note that the PACF at lag $k = 1$ does not consider any intermediate observations. It is essentially the correlation between observations y_t and y_{t-1} while ignoring the effect of any other lags (there are no intermediate lags). So the PACF at lag $k = 1$ is the same as the ACF at lag $k = 1$:

$$PACF(1) = ACF(1) = Corr(y_t, y_{t-1})$$

6.5 ARMA models

ARMA stands for **Auto Regressive Moving Average** models. ARMA models are used to model and forecast **stationary time series**. An ARMA model of order (p, q) is the sum of:

- an **Auto Regressive (AR) model** of order p
- a **Moving Average (MA) model** of order q

$$\left.\begin{aligned} y_t = c + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \ldots + \alpha_p y_{t-p} \end{aligned}\right\} \text{AR}(p)$$
$$+$$
$$\left.\begin{aligned} \epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_p \epsilon_{t-q} \end{aligned}\right\} \text{MA}(q)$$

Figure 6.17: ARMA model equation

6.5.1 Auto Regressive (AR) model

An **Auto Regressive (AR) model** of order p, denoted as $AR(p)$, models a time series so that the current value of a time series is a linear combination of the **past p values of the time series** plus a white noise term ϵ_t:

$$y_t = c + \alpha_1 y_{t-1} + \alpha_2 y_{t-2} + \ldots + \alpha_p y_{t-p} + \epsilon_t = c + \sum_{i=1}^{p} \alpha_i y_{t-i} + \epsilon_t \tag{6.2}$$

where c is a constant, $\alpha_1, \ldots, \alpha_p$ are the coefficients of the model, and ϵ_t is the white noise term. Notice that, in some way it is similar to the exponential smoothing method, but the AR model only uses p values and the weights α_i do not decay exponentially.

This method can improve the forcast with an horizon smaller than p (short-term).

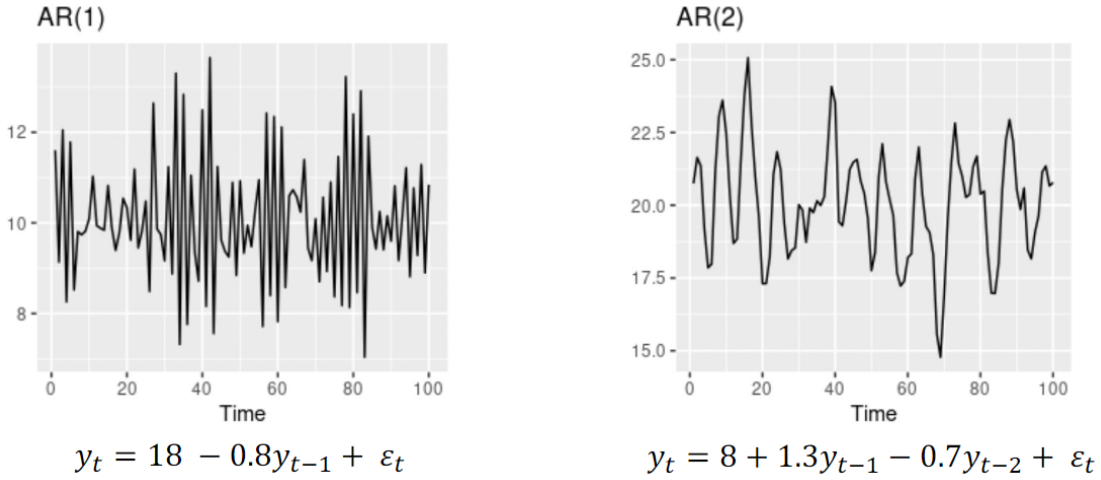


Figure 6.18: Auto Regressive models example

Let us see some special $AR(1)$ models:

- When $c = \alpha_1 = 0$, $AR(1)$ is equivalent to a white noise model.

- When $c = 0$ and $\alpha = 1$, $AR(1)$ is equivalent to a random walk model.
- When $c \neq 0$ and $\alpha = 1$, $AR(1)$ is equivalent to a random walk with drift model.
- When $c = 0$ and $\alpha < 1$, $AR(1)$ tends to oscillate between positive and negative values.

6.5.2 Moving Average (MA) model

A **Moving Average (MA) model** of order q, denoted as $MA(q)$, models the residual of a time series using **regression of past q estimation errors**:

$$y_t = c + \epsilon_t + \theta_1\epsilon_{t-1} + \theta_2\epsilon_{t-2} + ... + \theta_q\epsilon_{t-q} = c + \sum_{i=1}^{q} \theta_i\epsilon_{t-i} + \epsilon_t \tag{6.3}$$

where c is a constant, $\theta_1, ..., \theta_q$ are the coefficients of the model, and ϵ_t is the white noise term. Note that this model works on estimation errors, and it is different from the moving average of the time series.

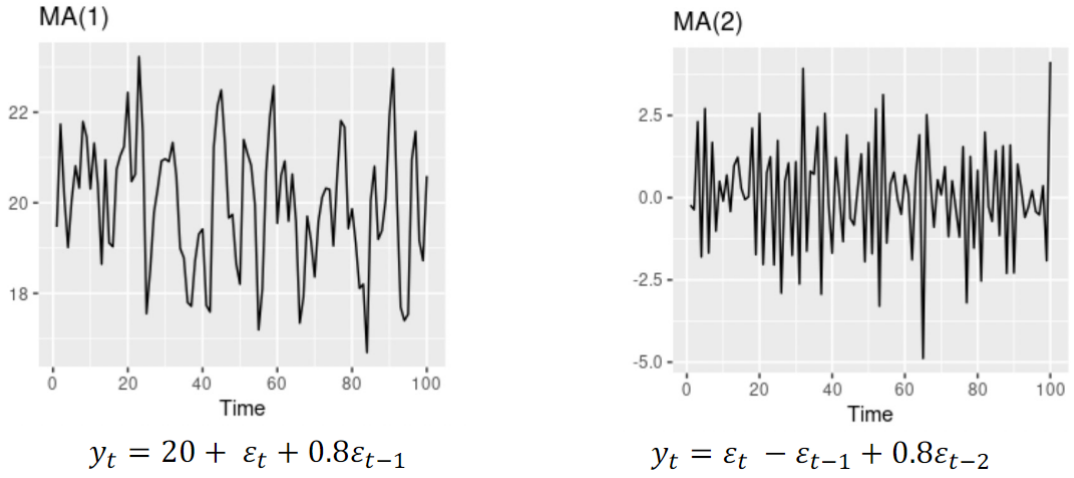


Figure 6.19: Moving Average models example

From $AR(p)$ to $MA(\infty)$

It is possible to write any **stationary** $AR(p)$ model as an $MA(\infty)$ model. For example, using repeated substitution, we can demonstrate this for an $AR(1)$ model:

$$\begin{aligned} y_t &= \alpha_1 y_{t-1} + \epsilon_t \\ &= \alpha_1(\alpha_1 y_{t-2} + \epsilon_{t-1}) + \epsilon_t \\ &= \alpha_1^2 y_{t-2} + \alpha_1\epsilon_{t-1} + \epsilon_t \\ &= \alpha_1^3 y_{t-3} + \alpha_1^2\epsilon_{t-2} + \alpha_1\epsilon_{t-1} + \epsilon_t \\ &... \end{aligned}$$

Provided $-1 < \alpha_1 < 1$, the value of α_1^k will get smaller as k gets larger. So eventually, we obtain the above equation equal to an $MA(\infty)$ model.

6.5.3 ARIMA model: ARMA + differencing

ARIMA stands for **Auto Regressive Integrated Moving Average** models. ARIMA models are a generalization of ARMA models that include **differencing** to make the time series stationary, by removing the trend component. Notice that this method **does not address the seasonality component**, so we need to remove it before applying the ARIMA model (e.g., by seasonal differencing).

An ARIMA model of order (p, d, q) is the sum of:

- an **Auto Regressive (AR) model** of p previous lagged values
- a **Moving Average (MA) model** of q previous lagged estimation errors

$$
\begin{aligned}
y'_t &= c + \alpha_1 y'_{t-1} + \alpha_2 y'_{t-2} + \ldots + \alpha_p y'_{t-\mathrm{p}} \quad \} \mathrm{AR}(p) \\
&\quad + \\
&\epsilon_t + \theta_1 \epsilon_{t-1} + \theta_2 \epsilon_{t-2} + \ldots + \theta_p \epsilon_{t-q} \quad \} \mathrm{MA}(q)
\end{aligned}
$$

$$
\text{Where } y'_t = y_t - y_{t-1}
$$

Figure 6.20: ARIMA model equation

Notice that the parameter d is the order of differencing. The goal is to remove the trend component of the time series. In other words, if we have a linear trend, we use $d = 1$ to remove it. If we have a quadratic trend, we use $d = 2$ to remove it, and so on.

The biggest question is: **how do we choose the values of p and q?** We can use the **ACF** and **PACF** plots to determine the values of p and q, using the **Box-Jenkins methodology**.

6.5.4 Box-Jenkins method

The **Box-Jenkins methodology** helps us to determine the values of p and q for an ARIMA model, along with evaluating the model's goodness of fit. The methodology is as follows:

1. **Identification:** determine the order of differencing d to make the time series stationary. Then, determine the values of p and q using the ACF and PACF plots.
2. **Estimation:** fit the ARIMA model to the time series using the values of p, d, and q determined in the identification step.
3. **Diagnostic checking:** evaluate the goodness of fit of the ARIMA model using the residuals. The residuals should be white noise, meaning that they are uncorrelated and have constant variance.
4. **Forecasting:** use the ARIMA model to forecast future values of the time series.

This process is iterative, meaning that we may need to go back and forth between the steps to refine the values of p and q and to improve the model's goodness of fit.

Identification step

The identification step is the most important step in the Box-Jenkins methodology. The goal is to determine the values of p and q using the ACF and PACF plots. In general, we use the following rules to determine the values of p and q:

- **AR model:** the ACF plot of an AR model will show a **slow decay** and the PACF plot will show a **sharp cutoff** after lag p. In this case, the model is ARMA$(p, 0)$ (order of AR model is p).

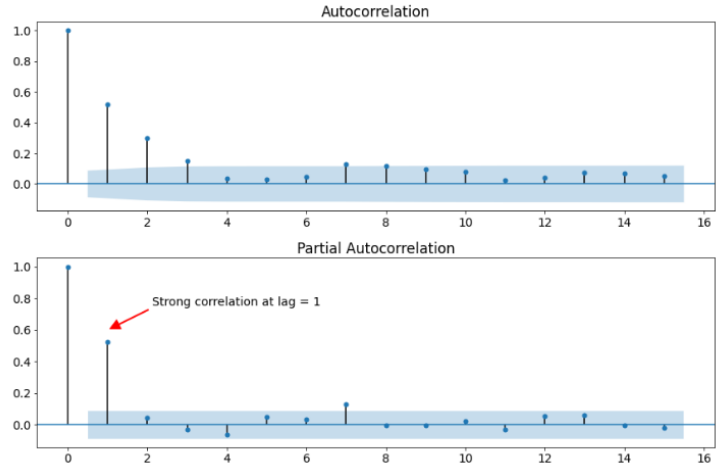


Figure 6.21: Example of an ARMA$(1, 0)$ model

- **MA model:** the PACF plot of an MA model will show a **slow decay** and the ACF plot will show a **sharp cutoff** after lag q. In this case, the model is ARMA$(0, q)$ (order of MA model is q).

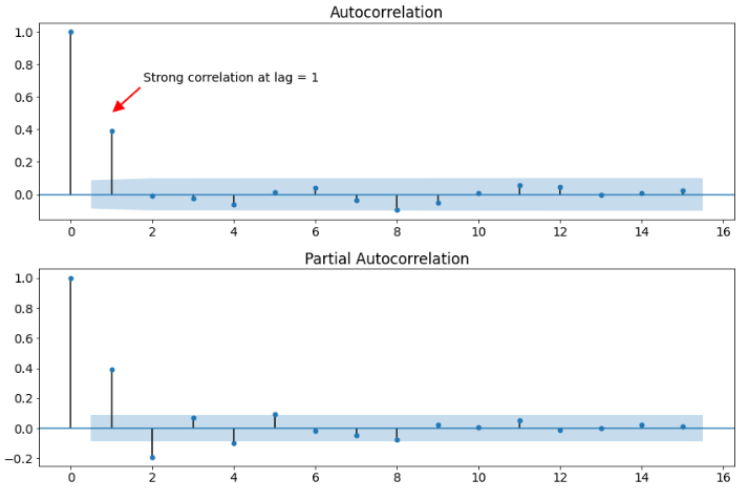


Figure 6.22: Example of an ARMA$(0, 1)$ model

We can also perform what is known as **grid search** to determine the values of p and q. We can fit multiple ARIMA models with different values of p and q, and then evaluate the models using the AIC or BIC criteria. The model with the lowest AIC or BIC value is the best model.

The **Akaike Information Criterion (AIC)** is a measure of the goodness of fit of a model. The AIC is defined as:

$$AIC = -2\log(L) + 2k$$

where L is the likelihood of the model and k is the number of parameters in the model.

The **Bayesian Information Criterion (BIC)** is also a measure of the goodness of fit of a model. The BIC is defined as:

$$BIC = -2\log(L) + k\log(n)$$

where L is the likelihood of the model, k is the number of parameters in the model, and n is the number of observations in the time series.

Estimation step

The estimation step is the process of fitting the ARIMA model to the time series using the values of p, d, and q determined in the identification step. The goal is to estimate the coefficients of the model that best describe the time series. The estimation step is done using the maximum likelihood estimation method.

In general, there are tools that can help us to fit the ARIMA model to the time series, such as the **statsmodels** library in Python. We will not cover in details the estimation step, as it is a technical process that requires knowledge of the maximum likelihood estimation method.

Diagnostic checking step

The diagnostic checking step is the process of evaluating the goodness of fit of the ARIMA model using the residuals. The residuals should be white noise, meaning that they are uncorrelated and have constant variance.

The diagnostic checking step is done using one or more of the following tools:

- **Residual plots:** plot the residuals of the ARIMA model to check for patterns or trends. The residuals should be randomly distributed around zero.
- **ACF and PACF plots of the residuals:** plot the ACF and PACF of the residuals to check for autocorrelation. The ACF and PACF of the residuals should not show any significant autocorrelation.
- **Ljung-Box test:** perform the Ljung-Box test on the residuals to check for autocorrelation. The Ljung-Box test is a statistical test that tests the null hypothesis that the residuals are uncorrelated. In general, if the p-value of the Ljung-Box test is less than 0.05, we reject the null hypothesis and conclude that the residuals are correlated.

- **Jarque-Bera test:** perform the Jarque-Bera test on the residuals to check for normality. The Jarque-Bera test is a statistical test that tests the null hypothesis that the residuals are normally distributed. In general, if the p-value of the Jarque-Bera test is less than 0.05, we reject the null hypothesis and conclude that the residuals are not normally distributed.

We also have other tools to evaluate the randomness of the residuals, depicted in the following figure:

Test	Good fit
Standardized residual	There are no obvious patterns in the residuals
Histogram plus kde estimate	The KDE curve should be very similar to the normal distribution
Normal Q-Q	Most of the data points should lie on the straight line
Correlogram	95% of correlations for lag greater than zero should not be significant

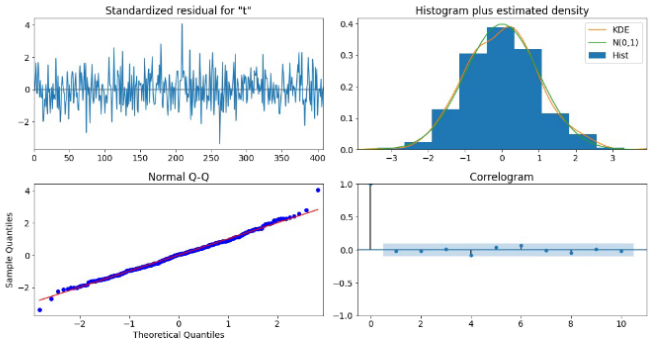


Figure 6.23: Residuals evaluation tools

Forecasting step

The forecasting step is the process of using the ARIMA model to forecast future values of the time series. The ARIMA model can be used to forecast the next n observations of the time series. The forecasted values are based on the estimated coefficients of the model and the observed values of the time series.

Note that, over time, errors accumulate, leading to straight line forecasts. In general, over the limits of order p and q, **predictions tend to converge into a constant value**, due to reliance on forecasted values. This is known as the **forecast horizon problem**.

In the end, **prolonged predictions may result in a loss of accuracy**, often manifesting as a linear trend or stable value over time.

6.5.5 Seasonal ARIMA models (SARIMA)

Seasonal ARIMA (SARIMA) models are an extension of ARIMA models that include seasonal components. Seasonal ARIMA models are used to model and forecast time series with seasonality. A seasonal ARIMA model of order $(p, d, q)(P, D, Q)_s$ is the sum of:

- an **Auto Regressive (AR) model** of order p and seasonal order P
- a **Moving Average (MA) model** of order q and seasonal order Q

- differencing of order d and seasonal order D

This model is very similar to the ARIMA model, except that there is an **additional set of autoregressive and moving average terms** to account for the seasonality (of period s). The SARIMA model is defined as:

$$y_t = c + \sum_{n=1}^{p} \alpha_n y_{t-n} + \sum_{n=1}^{q} \theta_n \epsilon_{t-n} + \sum_{n=1}^{P} \phi_n y_{t-sn} + \sum_{n=1}^{Q} \eta_n \epsilon_{t-sn} + \epsilon_t$$

Figure 6.24: Seasonal ARIMA model equation

6.5.6 SARIMAX

SARIMAX models are an extension of SARIMA models that include exogenous variables. Exogenous variables are external variables that are not part of the time series but may influence the time series. SARIMAX models are used to model and forecast time series with seasonality and exogenous variables.

SARIMAX includes all the components of SARIMA models, but with an extension to **incorporate r exogenous variables** $X_t = (x_{1,t}, x_{2,t}, ..., x_{r,t})$. The SARIMAX model is defined as:

$$y_t = c + \sum_{n=1}^{p} \alpha_n y_{t-n} + \sum_{n=1}^{q} \theta_n \epsilon_{t-n} + \sum_{n=1}^{r} \beta_n x_{n_t} + \sum_{n=1}^{P} \phi_n y_{t-sn} + \sum_{n=1}^{Q} \eta_n \epsilon_{t-sn} + \epsilon_t$$

Figure 6.25: SARIMAX model equation

6.6 Practical considerations

So far we have considered a fairly simple model, which decomposes a time series into a trend, seasonality, and residuals. Unfortunately, this model is not always sufficient to capture the complexity of real-world time series.

In practice, we need to consider the following practical considerations:

- **Multiple seasonality:** some time series may exhibit multiple seasonal patterns. For example, a time series may have a daily, weekly, and yearly seasonality. In this case, we need to use a model that can capture multiple seasonal patterns.
- **Changing trends:** some time series may exhibit changing trends over time. For example, a time series may have a linear trend in the beginning and a quadratic trend in the end. In this case, we need to use a model that can capture changing trends.

There are some advanced models that can capture these complexities, such as a **Multiple Seasonal-Trend decomposition**, that can capture multiple seasonal patterns and changing trends. For each component we do a forecast, then we add the components to get the final forecast. Also, there are some others like:

- VARIMA: Vector Auto Regressive Integrated Moving Average
- ARCH: Auto Regressive Conditional Heteroskedasticity
- GARCH: Generalized Auto Regressive Conditional Heteroskedasticity
- BATS: Box-Cox transformation, ARMA errors, Trend and Seasonal components
- TBATS: Trigonometric seasonality, Box-Cox transformation, ARMA errors, Trend and Seasonal components

Nonetheless, on real-world scenarios, we always have many characteristics to take into account, like daily seasonality, weekly seasonality, monthly seasonality, yearly seasonality, changing trends, holiday spikes, missing data, etc.

6.7 Prophet

Prophet is a forecasting tool developed by Meta that is designed for forecasting time series data that display patterns on different time scales. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

The creation of Prophet was motivated by the following:

- Completely automatic forecasting techniques can be brittle and they are often too inflexible to incorporate useful assumptions or heuristics.
- Analysts who can produce high-quality forecasts are quite rare because forecasting is a specialized skill that requires substantial experience.

6.7.1 Prophet under the hood

At its core, the Prophet procedure is an **additive regression model** with the following main components:

- A piecewise linear or logistic growth curve trend. Prophet automatically detects changes in trends by selecting changepoints from the data.
- A daily, weekly, and yearly seasonal component modeled using Fourier series.
- A user-provided list of important holidays that can impact the forecast.

The Prophet model is defined as:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

where:

- $g(t)$ is the growth (trend) function that models non-periodic changes in the value of the time series.

- $s(t)$ is the seasonal function that models periodic changes in the value of the time series.
- $h(t)$ is the holiday function that models the effect of holidays on the value of the time series.
- ϵ_t is the error term that models the random noise in the time series.

Growth (trend) function

The growth function $g(t)$ is a function that models non-periodic changes in the value of the time series. It captures changing trends through identifying changepoints in the data. These changepoints can also be manually specified by the user.

Note that the growth function can be piecewise linear or logistic:

- **Piecewise linear growth:**

$$g(t) = (k + a(t)^T\delta) \cdot t + (m + a(t)^T\gamma)$$

It is a simple modification of the linear model, yet very useful. For different intervals of time, we can have different linear models. The breakpoints are the value of x where the slope changes.

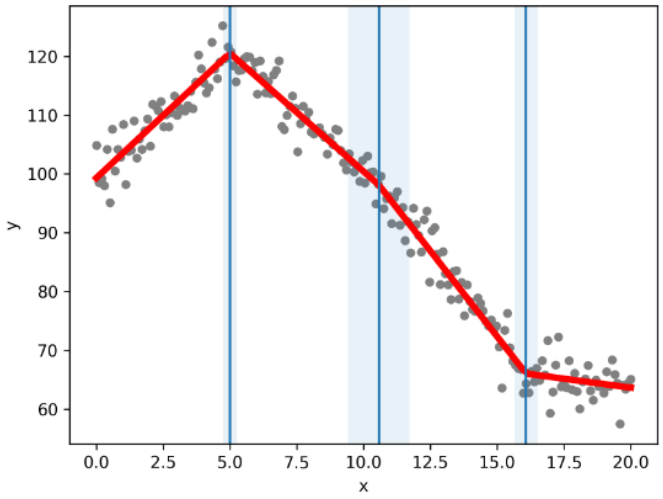


Figure 6.26: Piecewise linear growth curve

- **Logistic growth:**

$$g(t) = \frac{C(t)}{1 + \exp(-(k + a(t)^T\delta)(t - (m + a(t)^T\gamma)))}$$

It handles non-linear growth with saturation, i.e., the initial stage of growth is approximately exponential (geometric), then as saturation begins, the growth slows to linear (arithmetic) and at maturity, the growth stops.

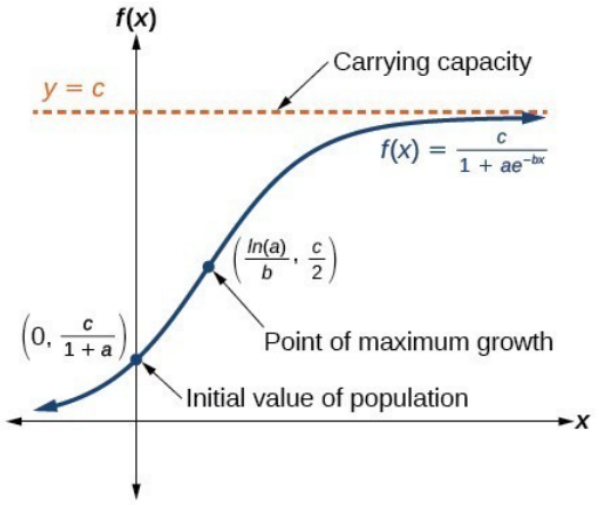


Figure 6.27: Logistic growth curve

Notice that the term $a(t)$ is a binary vector that indicates the presence of changepoints at time t. The term $C(t)$ is the carrying capacity of the growth curve, and it avoids the infinite or negative growth of the logistic curve.

Seasonal function

This function captures periodic changes in the value of the time series. It relies on Fourier series to provide a malleable model of periodic effects. The seasonal component $s(t)$ provides adaptability to the model b allowing periodic changes based on sub-daily, daily, weekly, and yearly patterns.

The seasonal function is defined as:

$$s(t) = \sum_{n=1}^{N} \left(a_n \cos\left(\frac{2\pi nt}{P}\right) + b_n \sin\left(\frac{2\pi nt}{P}\right) \right)$$

where N is the number of Fourier terms, P is the period of the seasonality, and a_n and b_n are the Fourier coefficients.

Multiple seasonalities can be modeled with Fourier series with a smoothing prior and a low-pass filter. Notice that truncating the series at N applies a low-pass filter to the seasonality. So, along with increasing the risk of overfitting, increasing N allows for fitting seasonal patterns that change more rapidly.

Note that we can write the seasonal function as a matrix multiplication:

$$s(t) = X(t) \cdot \beta$$

where $X(t)$ is the matrix of Fourier terms and β is the vector of Fourier coefficients. Also, $\beta \sim N(0, \sigma^2)$ as a prior. The choice of N is a hyperparameter that can be tuned using indicators like the AIC.

Holiday function

This function adds peaks and drops that can be explained by holidays. It can account for holidays with dates that changes year to year, like Easter, Thanksgiving, etc.

Holidays are modeled as independent dummy external regressors allowing the inclusion of a window of days around the holidays to model their effects. The holiday function $h(t)$ is defined as:

$$h(t) = \sum_{j=1}^{J} k_j \cdot I(t \in S_j)$$

where J is the number of holidays, k_j is the effect of holiday j, and $I(t \in S_j)$ is an indicator function that is 1 if t is in the window of holiday j and 0 otherwise. We can rewrite the holiday function as a matrix multiplication:

$$h(t) = Z(t) \cdot \kappa, \quad Z(t) = \begin{bmatrix} I(t \in S_1) \\ I(t \in S_2) \\ \vdots \\ I(t \in S_J) \end{bmatrix}$$

where $Z(t)$ is the matrix of holiday indicators and κ is the vector of holiday effects. As with seasonalities, Prophet uses a prior $\kappa \sim N(0, \sigma^2)$.

Additional regressors

It is also posible to add exogenous variables, i.e., additional regressors. Prophet provides a more general interface for defining extra linear regressors and does not require that the regressor be a binary indicator. Another time series could be used as a regressor, although its future values would have to be known to make forecasts.

Additional regressors are a generalization of the holidays and events, which are binary indicators.

6.7.2 Where Prophet shines

Prophet is optimized for the business forecast tasks at Meta, which typically have any of the following characteristics:

- Hourly, daily or weekly observations with at least a few months (preferably a year) of history.
- Strong multiple "human-scale" seasonalities: day of week and time of year.
- Important holidays that occur at irregular intervals that are known in advance (e.g., the Super Bowl).
- A reasonable number of missing observations or large outliers.
- Historical trend changes, for instance due to product launches or logging changes.
- Trends that are non-linear growth curves, where a trend hits a natural limit or saturates.

6.8 Deep learning for time series forecasting

For four decades, the "mantra" in applied forecasting has been "simple models outperform complex models", influenced largely by the results of the Makridakis series forecasting competition. This, until the most recent M-competitions, where every top solution **relied on Machine Learning**.

There were some serious challenges faced by standard existing models. Most methods were designed to forecasting individual series or small groups. From this, new problems emerged:

- Forecasting **many** individual or grouped time series.
- Trying to learn a global model facing the difficulty of **dealing with scale of different time-series** that would otherwise be related.
- Many classical models cannot **account for exogenous** or **multivariate inputs**.
- **Cold start problem** for new items to be included in the forecast.

This provided a set of goals for novel models. In general, they would need to have the ability to **learn** and **generalize from similar series**, that provides the ability to learn more complex models without overfitting; and the ability to **estimate the probability distribution** of a time series future, given its past (**probabilistic forecasting**).

6.8.1 DeepAR

One of the first solutions to these problems was the **DeepAR** model, developed by Amazon. It is a forecasting model based on **Autoregressive RNNs**, which learns a **global** model from historical data **of all time series** in all datasets.

DeepAR is the first succesful model to **combine Deep Learning** with traditional **Probabilistic Forecasting**. Its main advantages are:

- **Multiple time-series support:** the model is trained on multiple time series, learning global characteristics that further enhance forecasting accuracy.
- **Extra covariates:** DeepAR allows extra features (covariates). For instance, if your task is temperature forecasting, you can include humidity-level, air-pressure, etc.
- **Probabilistic output:** instead of making a single prediction, the model leverages Monte Carlo samples to output prediction intervals.
- **"Cold" forecast:** by learning from thousands of time series that potentially share a few similarities, DeepAR can provide forecasts for time-series that have little to no historical data.

6.8.2 DeepAR under the hood

DeepAR is a **sequence-to-sequence** model. It uses **LSTM** networks to create probabilistic outputs. Note that LSTMs are used in numerous time series forecasting model architectures:

- Multi-stacked LSTMs.
- LSTMs with CNNs
- LSTMs in encoder-decoder architectures
- Etc...

Contrary to the previous models, DeepAR uses LSTMs a but different: instead of using them to calculate predictions directly, DeepAR **leverages LSTMs to predict all parameters of a probability distribution** for the next time point. For instance, DeepAR estimates the $\theta = (\mu, \sigma)$ of a Gaussian distribution.

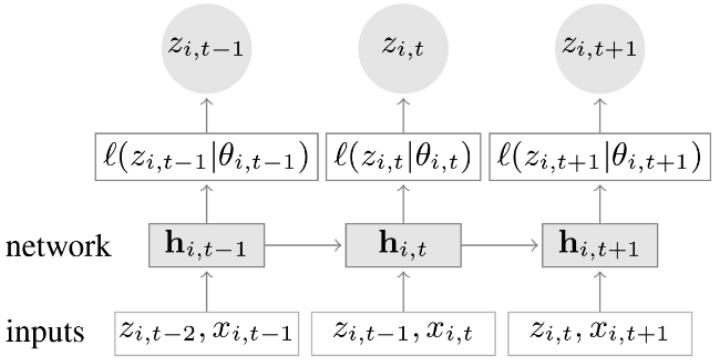


Figure 6.28: Simple diagram of LSTMs in DeepAR

Then, based on the result of the LSTM, DeepAR uses **two dense layers to derive the parameters of the distribution**. The first dense layer is used to calculate the mean μ of the distribution, and the second dense layer is used to calculate the standard deviation σ of the distribution.

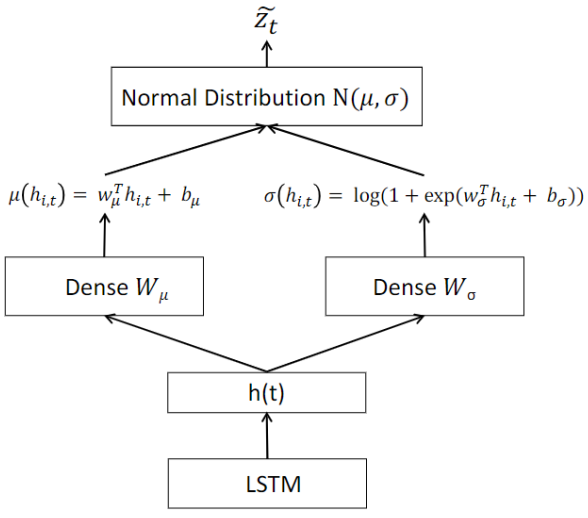


Figure 6.29: DeepAR architecture

DeepAR in various tests **outperformed traditional statistical methods** such as ARIMA. Also, the great advantage of DeepAR over those models is that **it does not require extra featuring preprocessing** (e.g., making the time series stationary first). This model os **prevalent in production**: it is part of Amazon's GluonTS toolkit for timeseries forecasting and it can be trained on Amazon SageMaker.

There are various other models like the following:

- D-Linear, N-Linear
- N-BEATS
- N-HiTS
- Temporal Convolutional Network
- Temporal Fusion Transformer
- TiDE - Time Series Dense Encoder
- TSMixer
- Space time former
- PatchTST
- TimesNet

Figure 6.30: Deep learning models for time series forecasting

6.8.3 Trade-offs

Deep learning models have some trade-offs that need to be considered. For instance, they are **computationally expensive** and **require large amounts of data**. Also, they are **black-box models**, which means that they are difficult to interpret and understand.

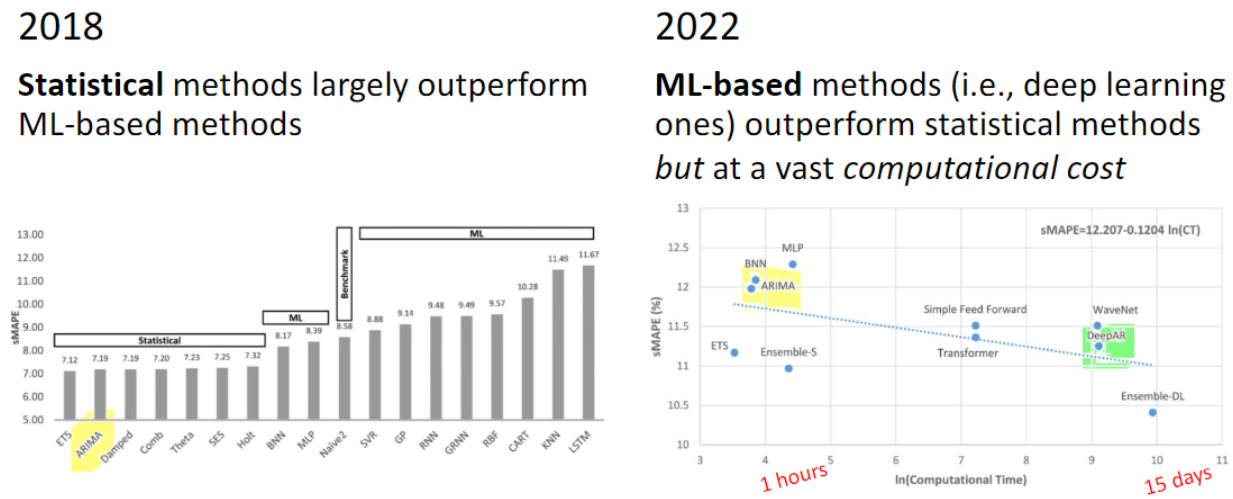


Figure 6.31: Trade-offs of deep learning models

Chapter 7

Streaming Machine Learning