Apache Hadoop

Apache Hadoop is an open-source framework designed to store and process large datasets efficiently, ranging from gigabytes to petabytes. Instead of relying on a single high-capacity computer for processing and storage, Hadoop enables you to cluster multiple computers together to analyze massive datasets more rapidly.

Hadoop consists of four main modules:

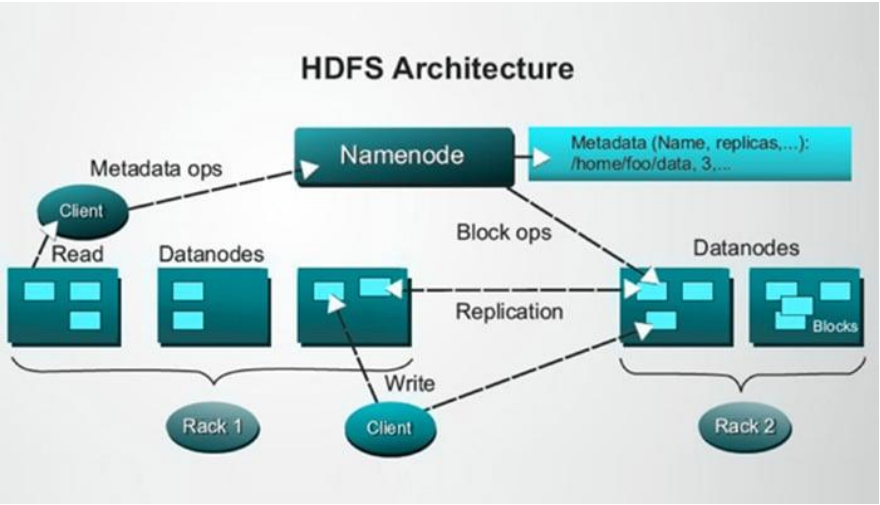
1. Hadoop Distributed File System (HDFS): A distributed file system that operates on standard or low-end hardware. HDFS offers superior data transfer rates compared to traditional file systems, high fault tolerance, and native support for large datasets.

   Namenode:

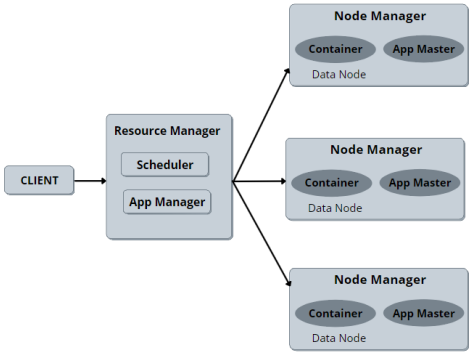
   - It acts as the master server that manages the metadata for the filesystem.
   - Stores information about the file system structure, such as file names, directories, and the mapping of blocks to files.
   - Does not store the actual data but rather keeps track of where data blocks are stored in the Datanodes.

   Datanode:

   - These are the worker nodes that actually store the data blocks.
   - Handle the read and write requests for data from the Namenode.
   - Regularly send heartbeat signals and block reports to the Namenode to ensure data integrity and availability.

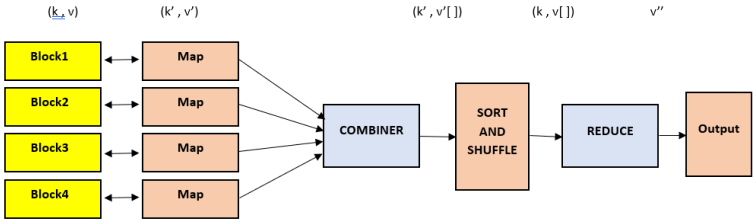


2. Yet Another Resource Negotiator (YARN): Manages and monitors the use of nodes and resources in a cluster. It schedules and oversees tasks and jobs.

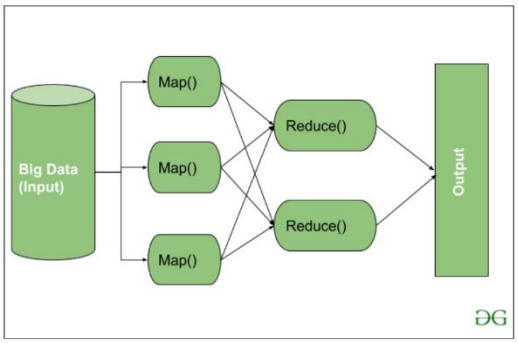


3. MapReduce: A framework that allows for parallel data processing. The Map function processes input data and transforms it into key-value pairs, which are then processed by the Reduce function to aggregate and generate the desired output.

    - The MapReduce can be implemented by using the MRJob library in Python programming language.



4. Hadoop Common: Provides a set of Java libraries and utilities that are used across all the Hadoop modules.



CAP Theorem

The CAP Theorem, also known as Brewer's Theorem, states that a distributed data store can only simultaneously achieve two out of the following three properties:

● Consistency refers to all nodes in a cluster having the same copy of data.

● Availability refers to data in each node that must be available every time it receives a request to access.

● Partition Tolerance refers to the system can continue working although network failure occurs.

Benefits of Apache

1. **Handles Large Data**: Manages data sizes beyond traditional databases.
2. **Distributed Processing**: Breaks tasks into smaller parts for simultaneous processing.
3. **Fault Tolerance**: Stores data redundantly across multiple nodes to prevent loss.
4. **Scalability**: Easily adds new nodes to scale up as needed.
5. **Supports Diverse Data**: Handles both structured and unstructured data.

Hadoop makes managing large data sets efficient and cost-effective.

How it work Apache Hadoop

Apache Hadoop works by breaking down large data sets into smaller chunks and storing them across multiple machines. Here's a simple overview of how it works:

1. **Data Storage**: Hadoop stores data in a distributed file system called Hadoop Distributed File System (HDFS). Data is split into blocks and spread across different machines (nodes) in the cluster.

2. **Data Processing**: When processing data, Hadoop uses a framework called MapReduce, which consists of two main steps:
    - **Map**: Data is processed in parallel across different machines, where each piece of data is handled by a "map" function.
    - **Reduce**: The intermediate results from the map phase are collected and processed by a "reduce" function to generate the final output.

This approach allows Hadoop to efficiently manage and process large volumes of data by leveraging the combined power of many machines.

MapReduce filters are used to filter data within the MapReduce process, selecting only data that meets specific criteria.

Example: Initial data: [10, 20, 30, 40, 50] Filter for values greater than 25.

- **Mapper**: Emit only items that meet the criteria, e.g., [(10, False), (20, False), (30, True), (40, True), (50, True)]
- **Reducer**: Collect items that passed the filter (True).

Result: [30, 40, 50]

MapReduce Distinct refers to a process used to eliminate duplicate data from a large dataset using the MapReduce programming model.

Example: Initial data: [1, 2, 2, 3, 3, 3, 4]

- **Mapper**: Emit each item with a count, e.g., [(1, 1), (2, 1), (2, 1), (3, 1), (3, 1), (3, 1), (4, 1)]
- **Reducer**: Collect unique items by removing duplicates.

Result: [1, 2, 3, 4]

MapReduce Binning is a technique used to organize and manage large datasets by grouping them into bins or buckets for easier analysis

Example: Initial data: [15, 25, 35, 45, 55] Group into bins (10-20, 20-30, 30-40, 40-50, 50-60).

- **Mapper**: Assign each item to a bin, e.g., [(15, '10-20'), (25, '20-30'), (35, '30-40'), (45, '40-50'), (55, '50-60')]
- **Reducer**: Collect items into their respective bins.

Result: { '10-20': [15], '20-30': [25], '30-40': [35], '40-50': [45], '50-60': [55] }

MapReduce Inverted Index is a process for creating an inverted index using the MapReduce framework to handle large-scale data processing for search and retrieval.

Example: Documents: Doc1: "apple banana apple", Doc2: "banana orange apple"

- **Mapper**: Emit each term with its document, e.g., [(apple, Doc1), (banana, Doc1), (apple, Doc1), (banana, Doc2), (orange, Doc2), (apple, Doc2)]
- **Reducer**: Collect terms and list the documents where each term appears.

Result: { 'apple': ['Doc1', 'Doc2'], 'banana': ['Doc1', 'Doc2'], 'orange': ['Doc2'] }

MapReduce Sorting is a process used in the MapReduce framework, which is a programming model for processing large data sets with a distributed algorithm on a cluster of computers.

Example: Initial data: [4, 1, 7, 3, 9]

- **Mapper**: Emit each item with a placeholder value, e.g., [(4, 1), (1, 1), (7, 1), (3, 1), (9, 1)]
- **Result** : [1, 3, 4, 7, 9]

MapReduce Join Join uses for combining records in two or more files based on a field.

- **MapReduce Inner Join** Inner join returns the intersection or common values.
- **MapReduce Left Outer Join** Left outer join returns all rows in the left table (file) and returns nothing for unmatched columns in the right table (file).
- **MapReduce Right Outer Join** Right outer join returns all rows in the right table (file) and returns nothing for unmatched columns in the left table (file).
- **MapReduce Full Outer Join** Full outer join returns the union or common and not-common values where it returns nothing for a table with no record matches.

Apache Spark

Apache Spark is a fast and efficient big data processing platform. It uses in-memory computing and supports multiple languages like Java, Scala, Python, and R. Spark handles large-scale data processing with distributed computing and offers features for stream processing, machine learning, and more.

Spark Core

- Spark Core is the heart of Apache Spark which supports in-memory computing, fault-tolerance, and parallel computing.
- At low-level, it works on RDDs and cluster manager.
- For high-level, it supports libraries such as Spark SQL,Streaming, MLlib, GraphX, etc.

Spark API

- Spark supports several application programming interfaces (APIs).
- The supported programming languages such as SQL, Scala, Java, Python, and R.
- Therefore, it is easy to work with different developments.

Streaming , MLlib , GraphX

- Streaming - conducts stream computation on stream data.
- MLlib - machine learning libraries for analysing data.
- GraphX - supports graph computations and analysis.

Benefits of Apache Spark

Apache Spark offers numerous advantages that make it one of the most dynamic projects in the Hadoop ecosystem, including:

- **Speed**
  Spark leverages in-memory caching and optimized query execution to perform analytical queries quickly, regardless of data size.
- **Developer-Friendly**
  Apache Spark supports Java, Scala, R, and Python, providing multiple language options for building applications. These APIs simplify development by abstracting the complexity of distributed processing behind high-level operations, significantly reducing the amount of code required.
- **Versatile Workloads**
  Apache Spark can handle a variety of workloads, including interactive queries, real-time

analytics, machine learning, and graph processing. Applications can seamlessly integrate multiple workloads within a single framework.

How it work Apache Spark

Apache Spark is a framework for processing large-scale data that is designed to be faster and more efficient compared to other systems like Hadoop MapReduce. Here's a summary of how Spark works:

1. **Resilient Distributed Datasets (RDDs)**: Spark uses RDDs, which are distributed memory abstractions that allow data to be processed quickly. RDDs keep data in RAM across multiple nodes in a cluster, enabling faster access and computation.
2. **In-Memory Computing**: By processing data in memory rather than frequently reading and writing from disk, Spark significantly speeds up operations. This approach reduces I/O overhead and improves performance.
3. **Distributed Computing**: Spark distributes processing tasks across multiple nodes in a cluster. This distribution allows Spark to handle large-scale data processing efficiently by parallelizing tasks and leveraging the collective computing power of the cluster.
4. **Job Management and Queuing**: Spark manages and schedules tasks through jobs and stages, breaking down complex workflows into smaller, manageable steps. It optimizes the execution plan to ensure efficient processing.
5. **Language Support**: Spark supports multiple programming languages, including Java, Scala, Python, and R. This flexibility allows developers to use the language they are most comfortable with or the one best suited for their project.
6. **Libraries and Modules**: Spark includes various modules for different types of data processing:
    - **Spark SQL**: For working with structured data and SQL queries.
    - **Spark Streaming**: For real-time data stream processing.
    - **MLlib**: For machine learning and data mining.
    - **GraphX**: For graph processing and analytics.

Apache Spark vs Hadoop

The main advantage of Apache Spark over Hadoop is its speed. Spark can perform operations up to 100 times faster in memory and up to 10 times faster on disk compared to Hadoop, which operates only on disk.

Additionally, Hadoop lacks the capability to analyze data in real-time, whereas Spark excels in real-time data processing.

Database

Relational Database Management System (**RDMS**)

An RDBMS is a system designed to manage and store data in a structured format using tables. These tables consist of rows and columns, and the system is capable of handling relationships between different tables.

RDBMS has ACID properties which are :

- Atomicity - the incomplete transaction cannot update the database.
- Consistency - data is consistent before and after a transaction.
- Isolation - each transaction does not interfere with others.

○ Durability - a database must be updated to ensure data will not be lost if the system fails.

NoSQL

- Non-relational database
- NoSQL has BASE properties which are :
  - ○ Basically Available - the system is always available even when a network failure occurs.
  - ○ Soft state - means it has flexibility with consistent requirements.
  - ○ Eventually consistent - the system eventually becomes consistent.
- NoSQL is horizontal scaling which allows more data to be inserted into the database.
- Working with NoSQL must concern the CAP theorem.

RDBMS VS NoSQL (Edward and Sabharwal, 2015)

	RDBMS	NoSQL
Schema flexibility	Inflexible, often ends up creating new tables	Column-oriented which allows adding more columns. It also supports semi-structured data.
Complex query	Often uses complex JOIN queries which are difficult to implement and maintain	It does not support relationships and foreign keys, thus, no complex query
Data update	If the system does not allow for updating multiple nodes at the same time, there is a risk of node failure	Synchronisation across nodes is challenging. However, NoSQL solutions offer synchronisation options.
Scalability	Low speed for large amounts of data	Provide great scalability

Key-Value

- It stores data in the form of key-value pairs.
- Key is usually a string or an integer.
- Key is usually a string or an integer.

- Advantages:
  - ○ Scalability - it is horizontal scaling through partitioning and replication. It also has low overhead.
  - ○ Mobility - it is easy to move from one to another system without changing in code/architecture required.
- Disadvantages:
  - ○ All joins must be done in code.
  - ○ No complex query filters.

Document Databases

- It is similar to key-value databases in that each document has a unique key (ID).
- Each document can store any type of data.
- Its query is JSON-like documents.

- Advantages:
  - It collects data from RAM which is fast to access.
  - It is horizontal scaling.
- Disadvantages:
  - Selecting data from multiple collections requires multiple queries.
  - Data duplication can occur which makes it difficult to handle.

Column-Oriented Databases

are a type of database management system designed to store and manage data by columns rather than by rows. This approach is different from traditional row-oriented databases, which store data row by row.

- Advantages:
  - It is scalable and flexible.
  - Load and aggregation times are very fast.
- Disadvantages:
  - It is slow when deleting rows.
  - It can be slow when querying data using a join query.

Graph Database are a type of database designed to store and manage data that is interconnected in a complex way. They organize data in the form of graphs, consisting of **nodes** (or vertices) and **edges** (or relationships) that connect them.

- **Nodes**: Represent data or objects of interest, such as users, products, or events.
- **Edges**: Represent relationships between nodes, such as "friends," "purchased," or "works at."
- **Properties**: Additional information about nodes or edges, such as a user's name or purchase date.
- **Graph Traversal**: Searching for information by navigating through edges to connected nodes.

- Advantages:
  - It is easy to understand data and has descriptive queries.
  - It is flexible.
- Disadvantages:
  - It is difficult to scale.
  - It does not have a standard language.

MonogoDB

Select All :

SQL: select * from <table>

MQL: { }

Select ..... Where :

SQL: select * from <table> where <column> = <value>

MQL: {<column>:<value>}

Ex. { "name": "Alice", "age": { $gte: 20 } }, { "name": 1, "age": 1 }

Select ..... Where ..... And

SQL: select * from <table> where <column1> =
<value1> and <column2> = <value2>

MQL: {<column1>:<value1>,
<column2>:<value2>}

Ex. { $and: [{ age: 30 }, { position: "Manager" }] }

Select ..... Where ..... Or

SQL: select * from <table> where <column1> =
<value1> or <column2> = <value2>

MQL: {$or: [{<column1>:<value1>},
{<column2>:<value2>}] }

Ex. { $or: [{ age: 20 }, { name: "Alice" }] }

Select ..... Where ..... And ..... Or

SQL: select * from <table> where (<column1> = <value1>) and
(<column2> = <value2> or <column3> = <value3>)

MQL: {{<column1>:<value1>}, $or: [
{<column2>:<value2>},{<column3>:<value3>}] }

Ex. : { $or: [{ num_reactions: {$gt: 3000 } } , { num_commnets: {$lt : 500} }] }

Assignment (1 point)

- Please select all records that have a number of likes equal to 500 and also have a number of reactions greater than 3,000 or a number of comments greater than 10

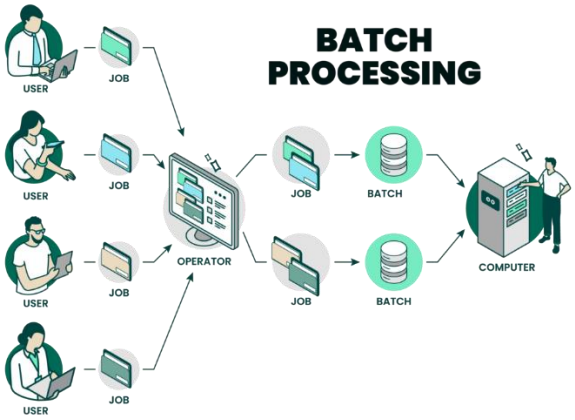

```
_id: ObjectId('668e4c01f5b615d3c9216f5b')
status_id : "246675545449582_515902548526879"
status_type : "video"
status_published : "3/6/2014 5:29"
num_reactions : 500
num_comments : 16
num_shares : 0
num_likes : 500
num_loves : 0
num_wows : 0
num_hahas : 0
num_sads : 0
num_angrys : 0
```


Ans . { num_likes: 500**, $or: [{ num_reactions: { $gt:** 3000 **} }, { num_comments: { $gt:** 10 **} }] }**

Low level Spark

Batch Processing

Batch Processing is a method of handling a group of data or tasks all at once, without user interaction during the process. This method is often used for tasks that require repetitive operations or when there is a large amount of data that needs to be processed simultaneously**.**



Spark Application

- Spark Application is a program developed by users over the Spark using Spark APIs.
- SparkSession communicates Spark with users via Spark APIs.
- Task is the smallest execution unit that is executed in an executor.
- Stage is a collection of tasks that execute the same code on different chunks of the dataset.
- Job consists of several stages and permits to execute the application in Spark.
- SparkSession is used in Spark 2.0 where SparkContext is used in the previous version of Spark.
- SparkSession is created using the builder function:
  SparkSession.builder()

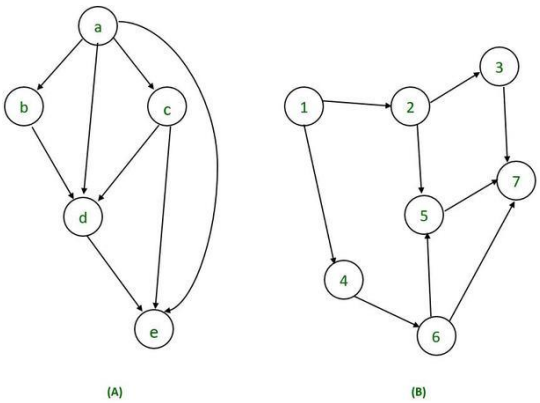
- To retrieve an existing session, use the function getOrCreate().

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
print(spark)
```

```
<pyspark.sql.session.SparkSession object at 0x7fd9d0ccc3d0>
```

- Spark operations are classified into 2 operations:
  - **Transformations**
    - Transformations take a Resilient Distributed Dataset (RDD) or DataFrame as an input and return RDD or DataFrame as an output.
    - Immutable - preserves the original copy of data
    - Not executed immediately - memorised and creates a transformation lineage which is a sequence of operations recorded in a Directed Acyclic Graph (DAG) that will be executed when an action is called. This is known as Lazy Evaluation.

- **Directed Acyclic Graph (DAG)**



A DAG is a type of graph where nodes (represented by circles or squares) are connected by edges (represented by arrows). The key characteristic of a DAG is that there are **no directed cycles**. This means that if you start at any node and follow the arrows, you'll never end up back at the same node.

Transformation จะถูกแบ่งออกเป็น **2** ประเภท

1.1 Narrow transformations

operations without data shuffling, in other words, one data partition results in one output map(), mapPartition(), filter(), union()

1.2 Wide transformations

involve data shuffling, in other words, one data partition results in multiple output partitions. join(), distinct(), aggregate(), repartition(), intersect()

- **Actions**
  - Spark operations return a single value.
  - It triggers the transformations.
  - Example functions: collect(), count(), min(), max(), top(),

Resilient Distributed Datasets (RDDs)

- Datasets are divided into logical partitions
- partitions are processed in parallel across different nodes
- RDDs can be created by existing collections or from external datasets such as text, CSV, and JSON files

High Level Spark

Dataframes

is a data structure in Python, widely used in the Pandas library, to manage tabular data, which is similar to an Excel spreadsheet. Each row in a DataFrame represents a record, and each column represents a feature or attribute of the data.

A DataFrame can be created from various data sources such as:

- A dictionary of lists
- CSV, Excel, or other files
- Numpy arrays

```python
import pandas as pd

# Create a DataFrame from a dictionary
data = {
    'Name': ['John', 'Jane', 'Doe'],
    'Age': [25, 30, 22],
    'City': ['Bangkok', 'Chiang Mai', 'Khon Kaen']
}

df = pd.DataFrame(data)

print(df)
```

```
   Name  Age       City
0  John   25    Bangkok
1  Jane   30  Chiang Mai
2   Doe   22  Khon Kaen
```

Save Modes (Antolínez García, 2023)

- Data can be saved into a file.
- Save Modes are:
  - errorifexists or error - exception is sent if data already exists
  - append - data is appended to the destination
  - overwrite - data is overwritten if data already exists
  - ignore - data will be ignored if data already exists

```
sqlDF.write.mode("overwrite").csv("<folder>") # Save data
```

Random Split (Chambers and Zaharia, 2018)

- randomSplit is used for splitting data in a Dataframe.
- It is useful when creating training and test sets are required such as when working with machine learning.

```
split = sqlDF.randomSplit([number1, number2]) # Split data
where numbers 1 and 2 are sum to 1.0.
split[0].show() # Show data in the first set (number1)
split[1].show() # Show data in the second set (number2)
```

Aggregations

Aggregations collect data and produce a single result for each group, using functions like count, first, last, min, max, and sum.

Functions

- countDistinct is used for counting number of unique groups.
- count is used for counting rows.
- first and last are used to retrieve the first and last rows from a Dataframe.
- min and max are used for finding the minimum and maximum values from a Dataframe.
- sum and sumDistinct are used for summing a total where sumDistinct sums a distinct set of values. (**Distinct =** ค่าที่แตกต่างกัน)
- avg (average) is used for calculating an average value of a column in a Dataframe.

Join

● Join combines two sets of data based on the key of the left and right datasets (tables).

● Spark discards the unmatched rows and then returns matched rows.

● Join evaluates the result using a join expression.

```
join_column = sqlDF1[“<column name>”] == \
              sqlDF2[“<column name>”] # Identify column
                                       # for matching
```

● Inner join keeps rows with keys matched in the left and right datasets.

```
sqlDF1.join(sqlDF2, join_column).show() # Inner join and
                                        # show results

joinType = "inner"
sqlDF1.join(sqlDF2, join_column, joinType).show() # Inner
                                  # join and show results
                                  # with using joinType
```

- Outer join keeps rows with keys in either the left or right datasets.

```
joinType = "outer"
sqlDF1.join(sqlDF2, join_column, joinType).show()
```

- Left outer join keeps rows with keys in the left dataset.
- Right outer join keeps rows with keys in the right dataset.

```
joinTypeLeft = "left_outer" # Left outer join
sqlDF1.join(sqlDF2, join_column, joinTypeLeft).show()
joinTypeRight = "right_outer" # Right outer join
sqlDF1.join(sqlDF2, join_column, joinTypeRight).show()
```

Structured Streaming

Streaming Processing

Streaming Processing is a type of data processing where data is continuously processed in real-time or near real-time as it is generated or received. This approach is suitable for scenarios that require immediate responses or analysis, such as fraud detection in financial transactions, real-time sensor data analysis, or monitoring social media status.

Spark Structured Streaming

The primary function of Structured Streaming is to convert streaming data into a structured format using DataFrames or Datasets, which can then be processed using SQL, DataFrame API, or Dataset API. Examples of use cases include event detection in data, real-time data analysis, or real-time statistical computations

Dataframes Streaming API

- Streaming Dataframes created using
  SparkSession.readStream()
- Input Sources:
  - Socket source
  - File source - a file such as CSV, JSON, or Parquet is read as a stream of data.
  - Kafka source - data is read from the Kafka source.

Socket Source

- Data can be ingested by listening to a socket connection.
- It is often used for testing.
- To execute, use NetCat which is in a Nmap package.
- Download Nmap (.exe)

ncat -l 9999 # 9999 **Port**

File Source

- A file is read as a stream of data.
- Once the file is modified, the file will be processed in the structured streaming
- Supported file formats are such as Text, CSV, JSON, etc.
- File source used in Structured Streaming requires a specified schema.
- This is to ensure a consistent schema being used for the streaming query.

Source and Sink

Data Sources

- Streaming Dataframes created using SparkSession.readStream()
- Input Sources:
  - Socket source
  - File source
  - Kafka source

Data Sinks

- Spark Structured Streaming output sinks are used for saving processed data into an external source.
  - Console sink - used for testing and debugging
  - File sink - stores data in the file system directory.
    - It needs checkpointing streaming.

Event-Time

Event Stream Processing

- Data needs to be processed over a specific interval of time because new sets of data are continuously ingested.
- As a group by operation often needs to see all data before executing the aggregation, windows can be used to illustrate finite data at a specific interval of time.

- Three types of windows:
  - Tumbling
  - Sliding
  - Session

Tumbling Window

Tumbling Window is a technique used in stream processing to group data that arrives continuously into fixed-size time intervals. Each interval is non-overlapping and distinct from others. For example, if you set a Tumbling Window to 5 minutes, incoming data will be grouped into 5-minute intervals, with each interval being separate and non-overlapping

Sliding Window

Sliding Window is a technique used in stream processing to manage and process incoming data by creating a "window" of data that moves over time. Instead of holding all

incoming data, this technique only processes a subset of data within the defined window size.

Session Window

refers to a method of grouping data based on a defined period during which a user performs activities within a single session

- For example, all received elements within 5 seconds are inserted into the same window. If there is no new data coming for 5 seconds, the current window will be closed.

Watermark

- Spark Structured Streaming uses watermark as a cutoff for controlling of how long the Processing will wait for late events.
- A timestamp is required to declare a watermark.