Spark Checkpointing Streaming

- Spark uses checkpointing to recover from failures.
- Checkpointing restores transitional states in the event of failures.
- Trigger is used to define how often a streaming query will be triggered.
    - One Time - trigger once and stops
    - Processing Time - trigger with user-defined interval
- Checkpoint Location points to file system directory for storing fault-tolerant in folders such as data checkpointing and metadata checkpointing.

Datanode ?

- HDFS splits data files into blocks.
- Each block is replicated into 3 copies and distributed to nodes in a cluster.
- The node that stores these copies of blocks is called Datanode.
- The Datanode serves data in the block when the read or write operation is called.

Namenode ?

- The Namenode or master node does not contain blocks of data.
- It stores the file system metadata including the mapping of blocks and their corresponding Datanodes.
- The file system includes fsimage and edits files.
    - fsimage file - contains the file system meta-data.
    - edits file - contains the updates of the meta-data. Therefore, the edits file is increased in size over time.

1. Namenode

- **Role:** The Namenode acts as the central management unit of the HDFS, responsible for storing and managing the metadata of all files within the system.
- **Functionality:**
    - It stores information about the structure of the files, such as where each file is located across different Datanodes.
    - It manages access permissions for the files.
    - It does not store the actual data but only the metadata.
    - When a file is read from or written to HDFS, the Namenode directs the client to the Datanode that holds the relevant portion of the file.

2. Datanode

- **Role:** Datanodes are the servers responsible for storing the actual data blocks that are distributed across the HDFS.
- **Functionality:**
    - They receive and store data blocks as instructed by the Namenode.
    - They periodically report the status of the stored data to the Namenode.
    - When a client needs to access data, the Datanode sends the data blocks to the client as directed by the Namenode.

Similarities

- Both Namenode and Datanode are essential components of the HDFS and work together to ensure efficient management and access to data.
- They communicate continuously to ensure that the data is managed correctly and remains available at all times.

Differences

- **Responsibility:** The Namenode manages the metadata of the HDFS, while the Datanode stores the actual data.
- **Criticality:** The Namenode is highly critical because if it fails, the entire HDFS might become non-operational. On the other hand, there are usually multiple Datanodes, so if one fails, the data remains safe due to replication across multiple Datanodes.

What is the difference between Hadoop Distributed File System (HDFS) and Yet Another Resource Negotiator (YARN)?

HDFS handles data storage, while YARN manages resource allocation and application execution within the Hadoop cluster.

What is the difference between MapReduce and Hadoop Common ?

MapReduce is specifically for processing data, whereas Hadoop Common provides the necessary infrastructure and support for various Hadoop components to work together.

What is the difference between Low-level and High-level ?

Low-level (RDD): Provides more control but is more complex to use. Suitable for tasks that require deep customization.
High-level (DataFrame, Dataset): Easier to use and reduces complexity. Ideal for tasks focused on data processing and general data analysis.

Low-Level Spark API

Advantages:

1. **Fine-Grained Control:** You have precise control over the data processing and execution plan. This is useful for optimizing performance or handling complex data transformations.
2. **Flexibility:** You can implement custom logic that might not be possible with higher-level abstractions.
3. **Optimization Opportunities:** Greater ability to optimize data processing, as you can fine-tune tasks and transformations at a lower level.

Disadvantages:

1. **Complexity:** More complex to write and maintain. Requires a deeper understanding of Spark's internals and data structures.
2. **Error-Prone:** Increased likelihood of bugs and inefficiencies due to manual handling of transformations and actions.
3. **Development Time:** Typically more time-consuming to develop and debug compared to higher-level APIs.

High-Level Spark API

Advantages:

1. **Ease of Use:** Simplified syntax and abstractions make it easier to write and understand code. Ideal for common data processing tasks.
2. **Productivity:** Faster development and fewer lines of code. The high-level API often handles optimization internally.
3. **Less Boilerplate:** Abstracts away the complexity of data transformations and actions, allowing you to focus on business logic.

Disadvantages:

1. **Less Control:** Limited control over the execution plan and optimization strategies. May not be suitable for highly specialized performance tuning.
2. **Potential Overhead:** The abstractions can introduce overhead that might impact performance in some cases.
3. **Limited Flexibility:** Some complex data processing tasks might be difficult or impossible to express with high-level abstractions.

Summary

- **Low-Level API** is best for situations where performance optimization and custom data processing logic are crucial. It requires more expertise and effort.
- **High-Level API** is suitable for most common tasks, offering ease of use and rapid development at the cost of some flexibility and control.

Data Sources

- **Socket Source**: This refers to data being sourced through network connections using protocols like TCP or UDP. It allows for receiving data from external sources or services over the network in real-time.
- **File Source**: This involves pulling data from files stored in a file system, such as CSV, JSON, or log files. It's often used for reading large amounts of data or continuously updated files.
- **Kafka Source**: This pertains to retrieving data from Apache Kafka, a distributed event streaming platform. Kafka Source pulls data from specified Kafka topics, enabling real-time processing and handling of large volumes of data from multiple sources or services.

Similarities:

1. **Data Ingestion**: All three are methods of data ingestion, meaning they are used to bring data from an external source into a system for processing.
2. **Real-time Capabilities**: Socket Source and Kafka Source are particularly suited for real-time data streaming, while File Source can also handle real-time updates if configured to watch for changes in files.
3. **Versatility**: Each source can handle various types of data, although the specific types they are best suited for differ (e.g., Kafka for event streams, File Source for structured data like CSV).

Differences:

1. Socket Source

- **Advantages:**
    - **Real-time Data Transfer**: Excellent for real-time data acquisition from external systems.

    - **Flexibility**: Can connect to any system that supports network communication via protocols like TCP or UDP.
    - **Low Latency**: Direct connection allows for minimal delay in data transfer.
- **Disadvantages:**
    - **Complexity**: Requires managing connections and handling potential network issues like latency, packet loss, or disconnections.
    - **Scalability**: Can be challenging to scale for high volumes of data or multiple connections.
    - **Limited Durability**: Data might be lost if the connection drops, as there's typically no built-in mechanism for buffering or retrying.

2. File Source

- **Advantages:**
    - **Simplicity**: Easy to set up and use for ingesting structured data from files.
    - **Reliability**: Data is persisted in files, making it easy to reprocess or backtrack if needed.
    - **Batch Processing**: Ideal for processing large datasets that are not time-sensitive.
- **Disadvantages:**
    - **Latency**: Not suitable for real-time processing as files need to be written, closed, and then read.
    - **Scalability**: Handling very large files or a high volume of files can be challenging.
    - **Staleness**: Data may become outdated between file updates, especially if the file is large or updated infrequently.

3. Kafka Source

- **Advantages:**
    - **High Throughput**: Designed to handle massive volumes of data with low latency.
    - **Scalability**: Easily scales horizontally across many producers and consumers.
    - **Reliability**: Built-in mechanisms for fault tolerance, data replication, and durability.
    - **Event Streaming**: Ideal for systems that need to handle continuous streams of events in real-time.
- **Disadvantages:**
    - **Complex Setup**: Requires more infrastructure and configuration, including Kafka brokers, topics, and consumers.
    - **Overhead**: May be overkill for smaller applications or when only a small amount of data is involved.
    - **Learning Curve**: Steeper learning curve for developers unfamiliar with distributed systems or Kafka's architecture.

Summary

- **Socket Source** is best for real-time, direct data transfers over the network but can be complex to manage.
- **File Source** is great for ingesting structured data from files, especially when real-time processing is not critical.
- **Kafka Source** is ideal for high-throughput, real-time event streaming in distributed systems but requires more complex infrastructure.

Data sinks

- **Console Sink**: This is a type of data sink where data is output to the console or terminal. It's commonly used for debugging or real-time monitoring during development. For example, when you use Console.WriteLine in C#, it's a console sink that outputs data to the terminal or command line interface.

  ข้อดี (Advantages):

  1. **Immediate Feedback**: Provides real-time output, which is useful for debugging and development.
  2. **Ease of Use**: Simple to implement and requires no configuration for basic usage.
  3. **Visibility**: Helps in monitoring and visualizing logs directly during the execution of the application.

  ข้อเสีย (Disadvantages):

  1. **Volatility**: Logs are lost when the application stops or the console is cleared.
  2. **Performance**: Excessive logging to the console can slow down the application, especially if logs are frequent.
  3. **Limited Persistence**: Not suitable for long-term storage or historical data analysis.

- **File Sink**: This refers to a data sink where data is written to a file. It is used for persistent storage, logging, or later analysis. For instance, logging frameworks often write log entries to a file sink so that you can review logs after the application has run.

  ข้อดี (Advantages):

  1. **Persistence**: Logs are saved to files, making them available for later review, analysis, and archival.
  2. **Size and Performance**: Can handle large amounts of log data efficiently and can be rotated or archived.
  3. **Search and Analysis**: Facilitates searching, filtering, and analyzing logs over time using various tools.

  ข้อเสีย (Disadvantages):

  1. **Disk Usage**: Consumes disk space, which can be significant depending on the volume and verbosity of logs.
  2. **Configuration**: Requires setup for file paths, log rotation, and management.
  3. **Latency**: May introduce a slight delay compared to console output, as writing to disk is slower than to memory.

Similarities:

1. **Purpose**: Both are used to output log or data information. They serve as endpoints where data is directed to be stored or viewed.
2. **Configuration**: Both can be configured to format data, apply filtering, and set logging levels (e.g., info, warning, error).

Differences:

1. **Output Location**:

- **Console Sink**: Outputs data to the console or terminal window. This is typically used for real-time monitoring and debugging while the application is running.
- **File Sink**: Outputs data to a file on the disk. This is useful for persistent storage of logs, which can be reviewed later for analysis, troubleshooting, or auditing.

2. **Persistence**:
   - **Console Sink**: Data is transient; once the console is closed or the application terminates, the data is usually lost unless specifically redirected to a file.
   - **File Sink**: Data is persistent; it remains on the disk even after the application has terminated, and can be accessed or analyzed later.
3. **Usage**:
   - **Console Sink**: Ideal for development, testing, and debugging, where real-time feedback is crucial.
   - **File Sink**: Ideal for production environments where logs need to be archived, reviewed, or monitored over time.

4. **Performance**:
   - **Console Sink**: Typically faster as it directly outputs to the terminal and doesn't involve disk I/O.
   - **File Sink**: May be slower due to file system I/O operations, especially with large volumes of data.

What are the differences between Transformations and Actions in Low level Spark? And where are they similar? (Transformations และ Action ใน Low level Spark แตกต่างกันตรงไหน และเหมือนตรงไหนบ้าง)

Similarities:

1. **Work with RDDs**: Both transformations and actions operate on RDDs. Transformations create a new RDD from an existing RDD, while actions perform computations on RDDs and return results.
2. **Used in Data Processing**: Both types of operations are crucial for building and processing data in Spark. Transformations are used to create new RDDs from existing ones, and actions are used to compute and retrieve results from RDDs.

Differences:

1. **Transformations**:
   - **Not Executed Immediately**: Transformations create a new RDD from an existing RDD but do not execute any computation until an action is called.
   - **Lazy Evaluation**: Transformations are lazily evaluated, meaning that the computation only happens when an action is triggered.
   - **Examples**: map, filter, flatMap, groupByKey, reduceByKey
2. **Actions**:
   - **Executed Immediately**: Actions trigger the computation of transformations and return a result. They initiate the execution of the computation plan.
   - **End of Pipeline**: Actions are the operations that actually compute and return results or output data to storage or to the console.

- **Examples**: collect, count, saveAsTextFile, first

In summary, transformations are used to create or modify RDDs without triggering immediate computation, while actions trigger the computation and provide the results or output.

Directed Acyclic Graph (DAG)

Advantages

1. **Organizes Data**: DAGs help in organizing data or tasks in a sequence that cannot loop back, providing a clear order.
2. **Dependency Analysis**: They are useful for managing and analyzing dependencies between events or steps, such as task scheduling or dependency management.
3. **Parallel Processing**: DAGs are used in systems where tasks or operations can be processed in parallel, improving efficiency in computations that can be executed concurrently.

Disadvantages

1. **Complex Management**: Managing and computing with large or complex DAGs can be challenging, especially as the number of nodes and edges increases.
2. **Difficulty in Changes**: Modifying the structure of a DAG, such as adding or removing nodes, may require significant adjustments to the graph, making management more difficult.
3. **Data Storage Usage**: DAGs can consume a substantial amount of storage space, particularly with a high number of nodes and edges.

DAGs are commonly used in various fields, including database management, task scheduling in computing systems, and dependency analysis. (DAG ใช้ใน ด้าน ใดบ้าง)

Batch Processing

Advantages

1. High Efficiency: Batch processing is often highly efficient for handling large volumes of data at once, as Spark can group operations together and reduce redundant work.
2. Handling Large Data Sets: It is well-suited for processing large datasets, such as transactional data or aggregated data collected over a period.
3. Resource Management: Batch processing allows for efficient resource allocation, as it can leverage cluster resources during off-peak times.
4. Repeatability: Batch processes are repeatable and predictable, as operations are scheduled and executed at set intervals.

Disadvantages

1. Latency: Batch processing can introduce delays because it processes data at scheduled intervals, so data arriving after the cutoff time must wait until the next batch.
2. Not Ideal for Real-Time Data: It is less suitable for real-time or streaming data that requires immediate processing and response.
3. Error Handling: Managing errors in batch processing can be complex, especially when dealing with large volumes of data at once.
4. State Management: Recovering and managing state can be challenging if errors occur during batch processing.

Resilient Distributed Datasets (RDDs)

Advantages of RDDs:

1. **Fault Tolerance**: RDDs are fault-tolerant because they maintain lineage information, which allows them to recompute lost data from the original source in case of failures.
2. **Parallel Processing**: RDDs support parallel processing, which significantly improves performance by distributing tasks across multiple nodes.
3. **Immutable**: RDDs are immutable, meaning once created, they cannot be modified. Any transformation creates a new RDD, which helps avoid race conditions and bugs.
4. **Lazy Evaluation**: RDDs use lazy evaluation, meaning computations are only performed when an action (e.g., collect, save) is triggered, which can optimize performance by minimizing unnecessary work.

Disadvantages of RDDs:

1. **Memory Usage**: RDDs can be memory-intensive because they require storing data in-memory across the cluster.
2. **Complexity**: Working with and tuning RDDs can be complex, especially for those unfamiliar with distributed computing concepts.
3. **Transformation Overhead**: Although RDDs employ lazy evaluation, complex transformations can still introduce significant computational overhead.

Use RDDs (ใช้ในด้านไหนบ้าง)

- **Recommendation Systems**: Platforms like Netflix or Amazon use big data processing to offer personalized recommendations based on user preferences.
- **Social Media Analytics**: Analyzing data from social media to understand trends and user behavior, which can influence marketing and content strategies.
- **Financial Services**: Processing and analyzing large financial datasets for market predictions, fraud detection, and other financial insights.

RDBMS (Relational Database Management System)

RDBMS is used for managing structured data organized in tables, with relationships between tables established through keys. Examples of daily applications include:

1. **Banking and Financial Systems**: Banks and financial institutions use RDBMS to manage customer data, transactions, bank accounts, and credit card information. These systems require high accuracy and consistency, which makes RDBMS suitable.
2. **Inventory Management Systems**: Large businesses, like retail stores, use RDBMS to track inventory, manage orders, and maintain supplier information. These systems need to keep data consistent and up-to-date.
3. **School Management Systems**: Schools and universities use RDBMS to manage student information, class schedules, grades, and teacher data. This data is interconnected and requires consistent updates, making RDBMS a good fit.

NoSQL

NoSQL databases do not store data in table formats like RDBMS, making them suitable for handling unstructured data or data that changes frequently. Examples of daily applications include:

1. **Social Media Platforms**: Platforms like Facebook and Twitter use NoSQL to manage user profiles, posts, comments, and likes. These platforms handle massive amounts of data that change frequently.
2. **Big Data Management Systems**: Companies like Google and Amazon use NoSQL to manage large volumes of data from various sources, such as user behavior, purchase history, and search queries.
3. **Recommendation Systems**: Websites like Netflix and Amazon use NoSQL to store and process data related to user viewing and purchasing habits, enabling them to recommend movies or products tailored to individual users.

Similarities:

1. **Data Storage**: Both RDBMS and NoSQL are systems used for storing data, allowing users to access and manage that data effectively.
2. **Support for Large Data**: Both systems can handle large volumes of data and scale to accommodate increasing amounts of information.
3. **Data Access Support**: Both RDBMS and NoSQL provide mechanisms for querying and retrieving the needed data through various commands.
4.

5.

MapReduce Count

```
class MapReduceCount(MRJob):
    def mapper(self, _, line):
        data = line.split(',')
        status_type = data[1].strip()
        if status_type == 'photo':
            yield 'Photo', 1
        .....
    def reducer(self, key, value):
        yield key, sum(value)
if (__name__ == '__main__'):
    MapReduceCount.run()
```

NOTE1: (_, line) is used to ignore the key and uses each line of the file as a value.

NOTE2: strip() is used for removing extra whitespaces.

NOTE3: yield keeps data in memory at the time it reads.

MapReduce Top-N

- Map function uses a field as a key to group-by and uses a value required for computing top-N (Bahga and Madisetti, 2019).
- Reduce function (Bahga and Madisetti, 2019)
  - Reduce function uses a list of values grouped by key received from the Map function.
  - It then sorts the values and finds the top-N in each group.

How Apache Spark Works?

(Aven, 2018; Antolínez García, 2023; Chambers and Zaharia, 2018)

- Client submits an application in Spark on the Driver (master node).
- The Driver creates a SparkSession (Spark 2.0) or SparkContext (Spark 1.0) to start the application.
- The Driver communicates with the Executor (slave node) to provide resources required for the application and execute the application code.
- The Driver also communicates with the Cluster Manager to conduct scheduling to keep track of resources.

RDDs from Parallelised Collections

(Antolínez García, 2023)

```
from pyspark.sql import SparkSession
spark = SparkSession.builder.getOrCreate()
alphabet = ['a', 'b', 'c', 'd', 'e', 'f', 'g', 'h']
rdd = spark.sparkContext.parallelize(alphabet, 4)
print("Number of partitions: " + str(rdd.getNumPartitions()))
```

Use textFile() to create RDD from file.

```
rdd2 = spark.sparkContext.textFile('fb_live_thailand.csv', 5)
print("Number of partitions: " + str(rdd2.getNumPartitions()))
```