# DAT535 - Taiwan dengue cases analysis

Yanis Joulou, Sofia Michailidu
University of Stavanger, Norway
y.joulou@stud.uis.no
s.michailidu@stud.uis.no

## ABSTRACT

This report presents the development of a scalable data processing pipeline for analysing dengue fever cases in Taiwan (1994–2024) using Apache Spark, applied to K-Means clustering and Random Forest regression. Profiling tools were utilized to analyse bottlenecks, while various cluster configurations were tested to identify optimal hardware settings for the workload. Key challenges included optimizing Spark configurations for effective resource utilization, managing computational complexity, and ensuring result interpretability. The project highlights Spark's capabilities in handling large-scale datasets efficiently.

The project code is available at https://github.com/YanisJl/dat535JouMic.

## KEYWORDS

Apache Spark, data processing, dengue fever, K-Means clustering, Random Forest regression, profiling, resource optimization, computational complexity, large-scale datasets

## 1 INTRODUCTION

The Taiwan Dengue Cases Analysis project explores the application of data-intensive systems and distributed computing for processing and analysing large-scale datasets. Dengue fever, a significant public health concern in tropical regions, serves as the focal point of this study. Using data from Taiwan, this project leverages modern big data technologies, including Apache Spark, to uncover patterns and insights within over two decades of epidemiological data.

The project's primary objectives include preparing data for identifying groups and patterns within the population affected by dengue to aid in public health interventions, developing scalable and efficient pipelines to handle increasing data volumes, ensuring computational robustness and evaluating the performance of distributed computing frameworks for big data analytics.

This report is structured as follows:

- **Key Concepts of Apache Spark**: A discussion of Spark's abstractions and principles relevant to the project's implementation.
- **Medallion Architecture**: An exploration of the layered approach to data processing adopted for this study.
- **Dataset**: A comprehensive overview of the dataset, including its structure and features.
- **Use Case**: A description of the study's analytical objectives and the algorithms applied.
- **Cloud and Hadoop Cluster Setup**: A description of the cluster setup using OpenStack, critical for ensuring computational efficiency.
- **Data Ingestion**: The methods employed to transfer and store raw data in a distributed environment.
- **Data Cleaning**: The preprocessing steps required to prepare the dataset for clustering and regression, including performance profiling and optimization.
- **Data Serving**: The implementation and profiling of machine learning algorithms for clustering and regression tasks.

By combining scalable tools and a systematic approach, this study demonstrates the potential of distributed systems in tackling real-world challenges in public health and big data analytics.

## 2 KEY CONCEPTS OF APACHE SPARK

Apache Spark[1] [3] is a robust engine designed for processing and managing data in a distributed computing environment. It is particularly well-suited for handling large-scale datasets, leveraging clusters and parallelism to achieve efficient data processing. Key advantages include minimizing network traffic and optimizing resource utilization, making it ideal for big data applications.

The concepts, abstractions, and processing mechanisms described in this chapter were foundational to the development of this project. A deep understanding of Spark's architecture and capabilities was essential in designing an efficient and scalable implementation, as these principles directly informed key decisions throughout the project. The discussed concepts are not only integral to the implementation but are also repeatedly referenced in subsequent chapters, underscoring their importance in achieving the project's objectives.

### 2.1 Efficient Parallel Processing in a Cluster

Spark achieves parallelism by utilizing a cluster-based execution model. When a computation is initiated, the Spark job tracker locates the data, typically stored in a Hadoop Distributed File System (HDFS), as further described in section 2.3. Rather than transferring the data to a central processing unit, Spark sends the computation directly to the nodes where the data is stored. This approach minimizes network traffic and significantly enhances performance. Each node in the cluster processes its respective portion of the data, executing the assigned task locally. This distributed computation model ensures efficient resource utilization and provides scalability.

The results of these computations are stored across the cluster in a manner consistent with the principles of horizontal scaling. Instead of relying on more powerful individual machines to process larger datasets, Spark enables the addition of more nodes to

*Project in Data-intensive Systems and Algorithms (DAT535), IDE, UiS 2024.*

[1] https://spark.apache.org

the cluster. This allows for the efficient processing of much larger datasets without the need to upgrade hardware.

Additionally, Spark abstracts the complexity of distributed file storage by presenting the final output as a single unified file. Although the data remains distributed across multiple machines, this abstraction eliminates unnecessary network traffic during data retrieval and storage, ensuring that operations remain efficient and optimized.

## 2.2 Spark Data Abstractions

In Apache Spark provides multiple data abstractions - RDDs (Resilient Distributed Datasets), Datasets, and DataFrames. Each data model offers distinct advantages depending on the requirements of the data processing task.

*2.2.1 RDD (Resilient Distributed Dataset).* An RDD[2] is a fundamental data structure in Spark, representing an immutable distributed collection of objects that is partitioned across the cluster.

RDDs in Spark also support lazy evaluation, meaning that transformations, further described in 2.4.1, on RDDs are not executed immediately. Instead, Spark builds a Directed Acyclic Graph (DAG) of all operations, and the actual computation is triggered only when an action (e.g., collect, count, save) is performed. This enables Spark to optimize the execution plan by analyzing the entire workflow before executing any steps, thus minimizing unnecessary computation and optimizing performance.

As described in [2], RDDs provide resilience not through data redundancy but by utilizing a minimal set of operations to recover from failures. When a node in the cluster fails, Spark uses the DAG to recompute the lost partition.

RDDs offer more flexibility by allowing low-level transformations and actions, such as `map`, `filter`, and `reduce`. These operations provide control over data manipulation but require a deeper understanding of the underlying system.

*2.2.2 Dataset.* A Dataset[3] is also a distributed collection of data, it represents a higher-level abstraction built on top of RDDs, providing benefits like strong typing and the ability to use advanced features such as lambda functions. Datasets also provide integration with the optimized execution engine of Spark SQL. Datasets are available in Scala and Java, but the Dataset API is not supported in Python.

*2.2.3 DataFrame.* A DataFrame[3] is essentially a Dataset organized into named columns. DataFrames abstract away many of the low-level details of working with RDDs and are suitable for tasks that resemble working with relational database tables or other tabular data.

## 2.3 HDFS (Hadoop Distributed File System)

HDFS is Spark's default storage layer, designed for distributed environments. It provides a unified storage solution across the cluster by distributing the data across multiple nodes. The primary components of HDFS are the name nodes and the data nodes. Each worker has its own file system, but HDFS acts as a shared layer, enabling seamless data access throughout the cluster. This architecture reduces unnecessary data movement, network traffic, and serialization overhead.

When a file is created, it is divided into blocks, which are distributed across the data nodes in the cluster. To ensure reliability and fault tolerance, each block is typically replicated three times, which is a redundancy level referred to as the replication factor. This design not only protects against data loss from hardware failures but also enables parallel processing by allowing multiple parts of a file to be read simultaneously.

## 2.4 Functional Abstraction in Spark

As described in [2], Spark employs a functional programming model to define its operations. A core principle of functional programming is immutability. Data or program states are not modified, which eliminates side effects and ensures that external variables remain unchanged. This immutability makes functional code safer for parallel execution, as it avoids race conditions and related concurrency issues.

An important feature of functional programming is the use of higher-order functions, which are those that accept other functions as arguments. Spark leverages this principle with operations such as `map` and `reduce`, which are foundational to its processing paradigm. Higher-order functions enhance flexibility and abstraction in program design while maintaining the simplicity and efficiency required for distributed computation.

In Spark, ensuring the correct and efficient use of higher-order functions like `map` and `reduce` is essential. These functions provide the building blocks for distributed processing, enabling seamless parallelism and optimization of large-scale data transformations.

*2.4.1 Transformations and Actions.* Spark distinguishes between two types of operations: transformations and actions[4].

**Transformations**, such as `map` and `filter`, are lazy operations that define a new dataset from an existing one. They do not execute immediately but instead construct a logical execution plan in the form of a DAG. This enables Spark to optimize execution by analyzing the dependencies and reducing redundant computations.

In contrast, **actions**, such as `count` or `collect`, trigger the execution of transformations. When an action is invoked, Spark evaluates the DAG and performs the necessary computations to produce the requested result.

This separation between transformations and actions is a core design feature of Spark, allowing it to achieve efficient execution while minimizing resource usage.

## 2.5 MapReduce

MapReduce is a programming model widely used in distributed systems to process large datasets. It is based on two core higher-order functions: `map` and `reduce`. These functions play complementary roles in the data processing pipeline.

*2.5.1 The Map Function.* This function applies a user-defined operation to each element in a dataset, producing a new set of key-value pairs. The `map` function is inherently parallel, meaning that its execution can be independently performed on chunks of

[2] https://spark.apache.org/docs/latest/rdd-programming-guide.html
[3] https://spark.apache.org/docs/latest/sql-programming-guide.html
[4] https://spark.apache.org/docs/latest/rdd-programming-guide.html

the dataset without affecting the results. This characteristic makes it highly suitable for distributed processing across multiple nodes in a cluster.

*2.5.2 The Reduce Function.* The `reduce` function aggregates values associated with the same key, combining them into a single result. Unlike `map`, `reduce` is inherently sequential, as its operation depends on processing the entire dataset step by step. However, as written in [2], when the operation satisfies two conditions—being associative and allowing data partitioning by distinct keys—the reduction process can also be parallelized.

*2.5.3 MapReduce Execution in Spark.* In Spark, `map` operations are straightforward to parallelize and distribute across the dataset. However, efficient execution of `reduce` operations requires additional considerations, particularly to minimize the computational and network overhead associated with data shuffling.

The shuffle phase is a critical step in MapReduce, where data is grouped by key and sorted before being sent to the appropriate nodes for reduction. Sorting during the shuffle phase is computationally expensive and often a bottleneck in the process. Furthermore, shuffling involves transferring data between nodes, which incurs significant network traffic. The performance of this phase can also be impacted by variations in the completion times of tasks on different nodes, leading to uneven workload distribution.

To mitigate these challenges, Spark introduces an intermediate combine phase. During this phase, partial aggregation of data is performed locally on each node before the shuffle phase begins. By reducing the volume of data transferred during shuffling, the combine phase significantly improves the efficiency of the overall process. The combine phase is an optimization step rather than a mandatory processing stage.

### 2.6 Job, Stage, and Task

In Spark, the execution of a computation is organized hierarchically into jobs, stages, and tasks, which together define the workflow and execution model of a Spark application.

*2.6.1 Job.* A job is created whenever an action, such as collect or count, is invoked on an RDD or a higher-level abstraction like a Dataset or DataFrame. A job represents the entire computation required to produce the result of that action.

*2.6.2 Stage.* A stage represents a subset of the job that can be executed without requiring a data shuffle. It corresponds to a sequence of transformations that can be performed in a single pass. When a job is divided, it is broken down into multiple stages based on the dependencies between transformations and the need for shuffling.

*2.6.3 Task.* A task is the smallest unit of execution in Spark. Each stage is divided into tasks, where each task processes a partition of data and runs on a single machine within the cluster. Tasks are distributed across the cluster's nodes, enabling parallel execution and efficient utilization of resources.

## 3 MEDALLION ARCHITECTURE

As described in [1], the Medallion Architecture is a data design pattern that emphasizes modularity by segmenting data processing into distinct stages. It employs a layered structure to organize and manage data logically. As data progresses through these layers, the goal is to continuously enhance its quality and suitability for specific use cases. The architecture comprises three primary layers: Bronze, Silver, and Gold, ensuring data integrity throughout the transformation process.

### 3.1 Bronze Layer

The Bronze layer represents the raw, ingested data directly imported from its source. This data typically retains the original structure and format of the source, often including unstructured data, such as logs. At this stage, no significant transformations or validations have been applied to the data and it is stored in its native form. This layer serves as the foundation for subsequent processing and transformations. This data is considered to be true and complete.

### 3.2 Silver Layer

In the Silver layer, the focus shifts to data cleaning and validation. Operations performed at this stage include handling missing or null values, removing invalid or duplicate records, and resolving errors and inconsistencies. It may involve connecting or integrating related datasets to provide a more unified view of the information. The data structure is also improved to ensure a higher level of usability.

While the Silver layer results in cleaner and better-organized data, it is not yet optimized for direct business use or reporting purposes. At this stage, the data may be standardized into a unified format based on a predefined schema, ensuring consistency across datasets. The Silver layer acts as a general-purpose intermediary, preparing the data for more targeted transformations in the Gold layer.

### 3.3 Gold Layer

The Gold layer consists of data that has been refined and transformed to meet the requirements of specific business use cases. This is the final stage of the Medallion Architecture, where advanced algorithms and processing techniques are applied to refine the data, making it ready for producing insights.

The data in the Gold layer is curated and structured to directly support business processes, reporting and analytics. It often involves filtering out irrelevant data to focus on what is most relevant for the intended use case. It is ready to meet the needs of end-users and is often the basis for dashboards or other business applications.

### 3.4 Application of Medallion Architecture in This Project

In this project, the Medallion Architecture is implemented as the foundation for designing the data processing pipeline. By organizing the data into the layers that were described, its quality and structure is enhanced progressively. The Bronze layer, described in 3.1, handles raw data ingestion. The Silver layer, covered in 3.2, focuses on structuring, cleaning and standardizing the data for two use cases. Finally, the Gold layer, discussed in 3.3, applies two different machine learning algorithms to the data, preparing it to reveal insights and support further interpretation.

## 4 DATASET

The dataset used in this project contains information about all reported dengue cases in Taiwan from 1994 to 2024. It originates from Ta-Wei Lo's contribution on the Kaggle platform [5]. The dataset contains approximately 107,000 records and includes 30 features. These features encompass physical information about individuals (such as age group and sex), temporal details (including onset, confirmation, and notification dates), and geographical data (specifying both the place of residence and the location of contamination). While the dataset includes additional features, they are not relevant to this study and will not be utilized.

For our project, the dataset was transformed into an unstructured format where all values were represented as strings, forming the input file for the data cleaning and processing pipeline.

To test the robustness and efficiency of our program under varying levels of technical stress, the dataset has been expanded by duplicating rows to create larger files. This process produced datasets that are 10, 50, and 100 times larger than the original file, enabling us to better identify computational bottlenecks and time-consuming operations. The 100 times larger dataset, approximately 3.5 GB in size, is used as the primary input for the analyses and descriptions provided in this report.

## 5 USE CASE

The project focused on two primary analytical techniques: K-Means clustering and Random Forest regression, applied to the dataset of dengue cases.

The first method, K-Means clustering, was used to group the data based on demographic and geographical features, aiming to uncover patterns and similarities within the dataset. This technique helped identify regions or population segments that shared similar characteristics, providing insights into potential correlations between these factors and the incidence of dengue cases.

The second method was Random Forest regression. While the regression model was not directly aimed at predicting specific outcomes, it can serve as a tool for completing the dataset.

Both use cases required the efficient processing of a large-scale dataset, but real-time performance was not a primary consideration, as the system was not intended for deployment in an environment requiring immediate or real-time processing. Instead, the focus was on handling large volumes of data effectively and ensuring that the algorithms could process the dataset within a reasonable time frame.

## 6 CLOUD AND HADOOP CLUSTER SETUP

The cloud environment was established using OpenStack on virtual machines provided by the university. Four virtual machines were created for the project, each equipped with 8 GB RAM and 4 CPU cores. A private network was configured, and a router connected this network to the external network. An interface was added to enable communication, and security group rules were carefully managed to allow secure SSH access to the virtual machines.

The Hadoop cluster was set up in a distributed mode, comprising one main node and three worker nodes. Essential software packages, including OpenJDK 8, Python, and `mrjob` for running MapReduce jobs, were installed on all nodes. Hadoop 3.2.1 was installed, and hostnames were assigned to ensure seamless communication between nodes. HDFS and YARN services were started from the main node to complete the cluster setup.

[5] https://www.kaggle.com/datasets/taweilo/taiwan-dengue-daily-confirmed-cases-1998-2024

## 7 DATA INGESTION

The dataset selection and the use case were previously described in earlier chapters, providing the context for the project. This section focuses on the data ingestion process, detailing the methods used to prepare the data for subsequent processing within the cloud and Hadoop cluster environment.

### 7.1 Ingestion Method

Since the use case and dataset did not necessitate a specialized ingestion pipeline, the data was manually downloaded and transferred to the main node. The unstructured dataset, provided in a text file, was uploaded to the virtual machine using `scp`, a secure method for quick file transfers, and loaded into HDFS. While this manual process is sufficient for smaller-scale operations, it could be automated using a script for scalability and efficiency in larger deployments.

### 7.2 Overview of the Ingested Data

The ingested data in the Bronze layer is unstructured and occupies approximately 3.5 GB of storage. The dataset comprises raw, unprocessed information directly sourced from its origin. The input data example is shown below:

```
... '1998/01/02' '' '1998/01/07' 'M' '40-44' 'Pingtung
    County' 'Pingtung City' 'None' 'None' 'A1320-0136-00' '
    120.505898941' '22.464206650' 'A1320-04-008' 'A1320-04'
     'None' 'None' 'None' 'None' 'N' 'None' '1' 'None' '
    10013' '1001301' ...
```

This unstructured data serves as the foundation for further cleaning, structuring, and processing in the subsequent Silver layer.

## 8 DATA CLEANING

The objective of the data cleaning process is to prepare the input data for gold-level use cases, ensuring consistency and reliability for machine learning models. Machine learning workflows demand structured and numerical data, which poses a challenge when the input data is unstructured or contains columns with unsuitable data types. Thus, cleaning and transformation steps are crucial to convert raw input into a format suitable for algorithmic processing.

### 8.1 Leveraging Spark for Data Cleaning

The data cleaning process utilized key Spark concepts, such as MapReduce (detailed in section 2.5) and the advantages of RDDs (discussed in section 2.2.1). These tools provided scalability, flexibility, and fault tolerance, essential for handling large, unstructured datasets efficiently. The distributed nature of Spark allowed parallel processing of data transformations, reducing computational time and optimizing resource usage.

## 8.2 Logical Division of the Workflow

Given that multiple machine learning algorithms were considered in this project, the cleaning process was logically divided into two distinct parts:

(1) **Common Cleaning:** This stage involved general preprocessing tasks to address inconsistencies in the raw data. Key steps included handling missing values, converting string data to numerical formats, and removing duplicate entries. These transformations ensured that the dataset met the foundational requirements for machine learning models, creating a consistent base for further, algorithm-specific processing.
(2) **Algorithm-Specific Cleaning:** After the common preprocessing, the data was further refined for the specific needs of each algorithm and use case:
   (a) **Clustering-Specific Cleaning:** For clustering, specific columns from the dataset were retained and transformed. The focus was on preparing features that would emphasize group similarities and differences and making the values format more suitable for clustering algorithm.
   (b) **Regression-Specific Cleaning:** For regression, a time-series dataset was created.

Each of the parts is described in the following sections.

## 8.3 Common Part: Preprocessing for Both Algorithms

The common part of the data cleaning workflow serves as the foundation for both clustering and regression preprocessing. Its primary goal is to create a structured and efficient dataset from the raw, unstructured input, ensuring consistency and preparing the data for algorithm-specific transformations.

The input data, being unstructured, is parsed to create a structured format of rows and columns. This is achieved using a delimiter-based splitting process implemented with the map function. Additional steps involve removing unnecessary characters, such as quotation marks. The code implementing this functionality including the loading of the file can be seen below:

```
data = sc.textFile(
    "hdfs:///project/unstructured_dengue.txt"
)

def remove_quotation_marks(row: List[Any]):
        row[0] = row[0].strip("'")
        row[-1] = row[-1].strip("'")
        return row

data = data \
        .map(lambda row: row.split("' '")) \
        .map(lambda row: remove_quotation_marks(row))
```

Once the data has been structured, irrelevant columns are removed to minimize the size of the dataset and improve processing efficiency. Only the columns relevant to subsequent algorithms are retained. Also the first row, which contains column names, is removed.

Each column is converted to the appropriate data type, ensuring compatibility with the algorithms. Date columns are converted to `datetime` objects. Numeric columns are converted to integers and categorical columns are kept as strings. These transformations are achieved through the `map` function as shown in the following code example:

```
def convert_datatypes(
    row: List[Any],
    convert_function: Callable[[Any], Any],
    idxs: List[int]
) -> List[Any]:
    for idx in idxs:
        if row[idx] != '':
            row[idx] = convert_function(row[idx])
        else:
            row[idx] = None
    return row


def convert_to_datetime(
    input_str: str,
    format_str: str
) -> datetime:
    return datetime.strptime(input_str, format_str)

data = data.map(
    lambda row: convert_datatypes(
        row,
        partial(
            convert_to_datetime,
            format_str='%Y/%m/%d'
        ),
        [0, 1]
    ).map(
        lambda row: convert_datatypes(row, int, [-1])
    )
```

*8.3.1 Resulting Data Structure.* After the common data cleaning process, the data is structured in RDD with rows and columns and all columns are in their appropriate data types. The structure can be seen below:

```
[
    [datetime.datetime(1998, 1, 2, 0, 0),
     datetime.datetime(1998, 1, 7, 0, 0),
     'M', '40-44', 'Pingtung County', 'N', 'None', 1],
    [datetime.datetime(1998, 1, 2, 0, 0),
     datetime.datetime(1998, 1, 7, 0, 0),
     'M', '40-44', 'Pingtung County', 'N', 'None',1],
    ...
]
```

## 8.4 Preprocessing for Clustering

The clustering-specific data cleaning process focuses on transforming the dataset into a format that is suitable for clustering algorithms.

The first step in the clustering preprocessing pipeline involves manipulating the time-related features. The goal is to convert date columns into a numerical format that can be utilized by clustering algorithms. For instance, the `Date_Onset` and `Date_Confirmation` columns are used to calculate the number of days since the onset date. The transformation is performed using the `map` function.

Several categorical columns are transformed into numerical formats suitable for clustering. For binary columns, such as `Sex` or `Imported`, the values are converted to 0 and 1 values. For numerical columns where the ordering is significant, such as `Age`, the column

is split and the lower boundary of the range is used as the numeric value.

Additionally, categorical columns like `County_living` and `Country_infected` are reduced to fewer categories. This is done using the functions shown below, which identifies and retains the most frequent categories while consolidating the rest.

```
def restrict_to_values(
    row: List[Any],
    idx: int,
    values: List[str]
) -> List[Any]:
    if row[idx] not in values:
        row[idx] = 'Other'
    return row


def reduce_number_of_categories_for_column(
    data: PipelinedRDD,
    idx: int,
    limit: int = 500
) -> Tuple[Any, List[str]]:
    counts = data.map(
        lambda row: (row[idx], 1)
    ).reduceByKey(
        lambda a, b: a + b
    )

    values_to_be_kept = (
        counts.map(
            lambda row: ('Other', row[1])
            if row[1] < limit or row[0] == 'None'
            else row
        )
        .reduceByKey(lambda a, b: a + b)
        .map(lambda row: row[0])
        .collect()
    )

    data = data.map(
        lambda row: restrict_to_values(
            row,
            idx,
            values_to_be_kept
        )
    )

    return data, values_to_be_kept
```

The missing values are replaced with a default value. For these categorical columns, one-hot encoding is applied.

After all categorical data has been encoded and time-based columns have been transformed, the data is normalized to ensure that all features are on the same scale. This is crucial for clustering algorithms such as K-means, which rely on distance metrics.

After all transformations and feature engineering steps are complete, the RDD is converted into a DataFrame.

*8.4.1 Resulting Data Structure.* The final dataset is ready for further clustering steps, where rows represent individual data points, and columns represent the processed features, all in the appropriate data types for clustering algorithms. The data structure example is shown below:

```
[
    [0.5120144173007609, 0.01824817518248175, 0.0,
     0.5714285714285714, 0.0, 0.5, 0, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0],
    [0.5120144173007609, 0.01824817518248175, 0.0,
     0.5714285714285714, 0.0, 0.5, 1, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
     0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0]
     ...
]
```

## 8.5 Preprocessing for Regression

The regression preprocessing pipeline focuses on preparing time-series data for model training. `Date_Onset`, is converted from a date-time format to a numerical representation using `time.mktime()` to obtain timestamps. Next, the data is aggregated to count the number of new cases per day using `reduceByKey` function as seen in the code below:

```
data = data.map(lambda x: [x, 1])
data_regression = data.reduceByKey(
    lambda sum, current: sum + current
)
```

Irrelevant columns are removed, leaving only the date and the number of new cases. The cleaned data is split into training and testing DataFrames.

*8.5.1 Resulting Data Structure.* The final dataset consists of time points (dates) and the target variable (new cases per day), formatted for the regression algorithm. The example of the data is shown below:

```
[
    (1449014400.0, 2170),
    (1449100800.0, 1550),
    ...
]
```

## 8.6 Error Handling and Workflow

The workflow began with the development and testing of the data processing pipeline in Jupyter Notebook. This environment allowed for step-by-step validation, where intermediate outputs were inspected to ensure each part of the process was functioning correctly. Errors were identified by reviewing the outputs and leveraging Spark's UI logs to monitor and debug any transformation failures.

After ensuring the individual components performed as expected, the notebook was converted into a Python script for more comprehensive execution and profiling. This transition allowed for a holistic view of the entire process.

## 8.7 Profiling and Optimization

After the functionality was implemented, the focus shifted to profiling and optimization to enhance the performance of the data

cleaning pipeline. By analysing execution times and system resource usage, we identified and addressed bottlenecks to improve processing efficiency.

*8.7.1 Tools Used for Profiling.* Key tools used for profiling included the Spark UI for monitoring job execution and `htop` and `ps` for tracking system resources like CPU and memory usage. These tools helped assess job performance, stage distribution, and resource consumption.

*8.7.2 Job Execution.* The process consisted of 6 jobs, with execution times and stages analysed to identify performance bottlenecks. Ideally, minimizing the number of jobs and stages is preferred to reduce data shuffling and overhead, as discussed in 2.5. However, certain parts of the cleaning process, for example the use of `reduceByKey` function in reducing the number of categories in column, necessitated shuffle operations or job terminations that could not be avoided without affecting the preprocessing objectives. The configuration for the execution included 3 executors, each with 2 cores and 2GB of memory, and 29 partitions were set automatically. The total execution time was 5.2 minutes.

*8.7.3 Challenges and Caching Optimization.* Upon inspecting the Spark UI, it became clear that the data was being loaded from the text file multiple times, causing the process to restart each time and resulting in repetitive common cleaning steps. The data loading operation, in particular, was time-consuming. This redundancy, driven by Spark's lazy evaluation mechanism discussed in 2.4.1, became a significant challenge, as the same cleaning process was applied multiple times for different algorithm-specific cleaning parts.

To address this, caching[6] was implemented in parts of the code where it was most beneficial. Caching was applied after the common cleaning stage, as the same data was reused for subsequent operations, such as creating indexed data for joining clustering results with the unprocessed dataset and for both algorithm-specific cleaning steps. This optimization reduced the execution time to 2.8 minutes. Additional caching was applied after the `zipWithIndex` operation, which showed minimal benefit due to the overhead of caching. This step slightly increased the execution time to 2.9 minutes, so it was not kept in the implementation. Further improvements were seen during the clustering cleaning stage, where the same RDD was reused for calculating the most frequent categories, reducing the execution time to 2.3 minutes after applying the caching.

*8.7.4 Resource Usage: Memory and Processing Time.* Due to the unstructured nature of the input data, it was not possible to selectively load only the necessary columns as one could with a columnar storage format such as parquet. Despite this, the entire dataset fit into memory and the data was processed efficiently, with memory usage monitored using the Spark UI. The overhead from garbage collection was minimal. Serialization time was also low, eliminating the need to use alternative serialization methods like Kryo [7], which would have introduced additional complexity and limitations with advanced data types.

[6] https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.RDD.cache.html
[7] https://github.com/EsotericSoftware/kryo

Garbage collection time was minimal, with most execution time spent on executor computing. Shuffle read/write and serialization times were also low, indicating efficient memory use. Job skew was absent, as each executor handled a balanced share of tasks, confirmed by the Spark UI.

# 9 DATA SERVING

After the cleaning phase, the prepared dataset was utilized in two distinct applications: clustering and regression for time series analysis. These use cases involved transforming the cleaned data into formats suitable for machine learning models while ensuring interpretability and efficiency in linking results back to the original dataset. To implement the data serving part of this project, Spark MLlib[8] was used. This section outlines the methodologies, challenges, and results for both applications.

## 9.1 Clustering

The objective of the clustering task was to group the dataset into meaningful clusters, enabling the identification of patterns in the data. K-Means clustering was chosen for this purpose, with the number of clusters set to $k = 10$.

*9.1.1 Methodology.* The clustering process involved several key steps:

(1) Transforming the data into a suitable format using `VectorAssembler`, which assembled all input features into a single column for model training.
(2) Training the K-Means model and generating predictions for each data point.
(3) Linking clustering results back to the original dataset to ensure interpretability.
(4) Exporting the final results to Parquet format for efficient storage and analysis.

After transforming the data, the use of Spark MLlib made the model training straightforward:

```
model = KMeans(k=10, seed=1)
model = model.fit(df)
predictions = model.transform(df)
```

Once predictions were generated, they were linked back to the unprocessed dataset using an index column. This ensured that the clustering results could be interpreted alongside original attributes such as `Sex`, `Age_Group`, and `County_Living`, as further discussed in the following section 9.1.2. The final dataset was exported in Parquet format, which provided the advantages of efficient storage due to columnar compression, broad interoperability and good read/write performance.

*9.1.2 Challenges and Solutions.* A key challenge during clustering was linking the results back to the original unprocessed data. Due to Spark's distributed architecture, data order is not guaranteed across partitions, making it impossible to align processed data with unprocessed data row by row.

Two solutions were considered:

[8] https://spark.apache.org/mllib/

(1) Storing both processed and unprocessed data together in the same row. While this approach would simplify the linking process, it would increase storage requirements due to redundancy.
(2) Adding an index to the dataset and using it to join the processed and unprocessed data. Although this method introduced additional computational overhead for the join operation, it ensured accuracy and flexibility.

The second solution was chosen despite the computational cost of the join operation. By indexing the data, clustering predictions were effectively linked back to the original dataset without duplicating information.

## 9.2 Regression

The regression task aimed to predict time series values using the cleaned dataset using a Random Forest Regressor.

*9.2.1 Methodology.* The process involved the following steps:

(1) Splitting the dataset into training (90%) and testing (10%) sets, sorted by the time attribute to preserve temporal order.
(2) Assembling the feature column for model training using `VectorAssembler`.
(3) Fitting the Random Forest Regressor on the training data.
(4) Generating predictions on the test set.
(5) Exporting the regression results to Parquet format for storage and further analysis.

Model training and prediction were implemented as follows:

```
lr = RandomForestRegressor()
model = lr.fit(train)
regression_results = model.transform(test)
```

Similar to the clustering task, the results were exported in Parquet format, providing efficient storage and facilitating further analysis.

## 9.3 Profiling and Comparison

K-Means clustering is often computationally intensive due to its iterative nature and the substantial computation required for optimizing cluster centroids and calculating distances between each data point and the cluster centers. This can become particularly challenging for high-dimensional data and large datasets.

Similarly, Random Forest training could be computationally demanding because it involves constructing a large number of decision trees and each tree must be stored during the training process.

These characteristics make comparing the two algorithms interesting. This comparison allows us to evaluate how effectively Spark handles these computational demands within the context of our specific project.

*9.3.1 Comparison of algorithms.* The performance comparison between the clustering and regression algorithms accounts for both computation time and the time spent on data cleaning and exporting results to Parquet. To ensure consistency, the code was adjusted to first run the entire data cleaning process, followed by executing just one of the algorithms along with its export to a Parquet file. This approach ensured that the data cleaning steps were the same for both algorithms, eliminating them as a factor in any performance differences. When exporting to Parquet, it was expected that the clustering algorithm would require more time due to the larger dataset being exported, as it involves more columns compared to the regression algorithm. As a result, the Parquet export time was anticipated to be longer for clustering than for the time series regression. The settings of the number of cores and memory per executor were kept the same as for the cleaning profiling.

For the **K-Means** algorithm, the total execution time was 5.7 minutes, including the data cleaning. When excluding the cleaning time, which took approximately 2.3 minutes, the actual computation time for clustering was around 3.4 minutes. A total of 40 jobs were executed, of which 6 were related to the cleaning process, leaving 34 jobs for the clustering algorithm itself. This includes 3 jobs related to saving the data to Parquet and possibly performing final operations. Notably, the last three jobs, which involved writing the data to Parquet, took 38, 60, and 8 seconds, respectively. The extended duration of these jobs could be attributed not only to the data writing process but also to additional tasks such as data gathering and final preparations.

There were a few jobs that took longer than expected, with durations in the lower to middle tens of seconds. For example, one job in the early stages of the clustering process took 50 seconds, which is notably longer than most other tasks. This suggests that the job might have been more computationally intensive, possibly related to data preparation or initialization tasks for the clustering algorithm.

In contrast, the **Random Forest Regressor** algorithm had a total execution time of 2.6 minutes, with only 0.3 minutes dedicated to the core algorithm itself and its export to Parquet. The regression process was considerably faster, with most jobs completing in under 0.5 seconds, indicating that the algorithm is computationally lighter than clustering in this context. The most time-consuming aspect of the regression process was the saving of data, taking approximately 2 seconds. In total, 20 jobs were run, 11 of which were related to the algorithm's execution and 3 of these involved saving the data to Parquet.

When comparing the two algorithms, it's clear that the regression algorithm is much faster, with jobs typically completing in under 0.5 seconds and fewer jobs overall. This suggests that the regression process is less resource-intensive and more efficient than the clustering algorithm. However, this difference could also be partially due to the clustering algorithm handling a larger amount of input data, which contributes to its increased computational complexity. On the other hand, the clustering algorithm involves more complex and resource-heavy operations, leading to longer job durations. Significant time is taken by the clustering algorithm's final Parquet saving jobs. It's important to note that these observations are specific to our use case and the particular datasets used for testing.

*9.3.2 Cluster Configuration Experiments.* To enhance performance, various configurations were tested by adjusting the CPU and memory allocations across three executors. These experiments aimed to identify the optimal balance between resource usage and execution time. The results, including the total execution time (encompassing the cleaning phase), are summarized in Table 1.

| Memory | 2 Cores | 3 Cores | 4 Cores |
|---|---|---|---|
| 512 MB | 9.2 | 7.3 | 6.5 |
| 1 GB | 8.1 | 6.7 | 6.3 |
| 2 GB | 5.9 | 5.0 | 4.4 |
| 3 GB | 5.9 | 4.9 | 4.4 |

**Table 1: Execution Time (minutes) for Different Cluster Configurations by Memory per Executor and Number of Cores per Executor (3 executors in total)**

Increasing the CPU count per executor consistently reduced execution time, demonstrating the benefit of additional parallelism. Increasing executor memory showed diminishing returns after 2 GB, suggesting that the dataset size was manageable within this range. The best performance was achieved with 4 CPUs and 2–3 GB of memory per executor, offering an optimal balance between resource usage and execution time.

*9.3.3 Challenges - DataNode utilization in spark cluster.* A challenge encountered during the project was the uneven utilization of DataNodes in the cluster. Despite having three DataNodes configured, only two of them were actively used during computation. This imbalance led to suboptimal resource usage and reduced the overall performance of the cluster.

To resolve this issue, we adjusted the Spark configuration to explicitly specify the number of executor instances, ensuring that all DataNodes were utilized effectively. The configuration was updated as follows:

```
conf = SparkConf() \
    .setAppName("project") \
    .set("spark.master", "yarn") \
    .set("spark.deploy.mode", "cluster") \
    .set("spark.executor.instances", "3") \
    .set("spark.executor.cores", "3") \
    .set("spark.executor.memory", "2G") \
```

In this configuration, the parameter *spark.executor.instances* allows us to force the cluster into using all 3 DataNodes.

Additionally, we made sure that the data was distributed and available across all the DataNodes.

These changes ensured that the computational workload was evenly distributed across the three DataNodes, leading to improved resource utilization and significantly reduced processing times.

## 10 CONCLUSION

This project implemented a scalable pipeline for processing and analyzing a dataset of dengue fever cases in Taiwan (1994–2024) using Apache Spark. The workflow was divided into three main phases: data ingestion, data cleaning, and data serving, with each phase addressing distinct technical challenges while leveraging distributed computing techniques to handle large-scale data efficiently. The ingestion phase established a cloud-based cluster setup with three DataNodes and a main node, providing a solid foundation for subsequent processing. Data cleaning involved preprocessing steps tailored for K-Means clustering and Random Forest regression, emphasizing low-level MapReduce routines to better understand Spark's distributed computation capabilities. Finally, the data serving phase applied these algorithms and exported the results in Parquet format.

A key challenge encountered was uneven utilization of DataNodes, which initially resulted in inefficient resource usage. Adjustments to the Spark configuration ensured that all DataNodes were utilized effectively, leading to improved performance and balanced workloads. Another challenge arose from Spark's lazy evaluation model, where redundant computations during data cleaning caused delays. Introducing caching mechanisms resolved this issue, significantly reducing processing times by reusing intermediate results. Through profiling and tuning, the project also demonstrated the relationship between hardware configurations and performance, showing the benefits of additional CPU cores and memory up to a certain threshold.

The project underscored the importance of optimization when leveraging Spark's distributed architecture, particularly for large-scale data processing tasks. It also highlighted the value of profiling and tuning, with tools like the Spark UI proving essential for identifying bottlenecks and validating configuration improvements.

While the primary goal was exploring Spark's scalability and computational efficiency, the project successfully demonstrated its ability to handle large datasets. Future work could include testing additional machine learning algorithms, such as a classification algorithm that could be used to fill in the missing data in the dataset, or further improvements of the profiling and tuning of the implementation.

This project highlighted the strengths and challenges of using Spark for distributed data processing.

## REFERENCES

[1] Mssaperla. [n. d.]. What is the Medallion Lakehouse Architecture? - azure databricks. https://learn.microsoft.com/da-dk/azure/databricks/lakehouse/medallion

[2] Tomasz Wiktorski. 2019. *Data-intensive Systems: Principles and Fundamentals using Hadoop and Spark.* Springer Cham. https://doi.org/10.1007/978-3-030-04603-3

[3] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, and Ion Stoica. 2016. Apache Spark: a unified engine for big data processing. *Commun. ACM* 59, 11 (Oct. 2016), 56–65. https://doi.org/10.1145/2934664

*This report benefited from the use of OpenAI's ChatGPT[9] for formal editing of the text. ChatGPT was utilized to enhance the clarity and professionalism of the text, ensuring that language and style met formal academic standards. Additionally, it was used to assist in resolving the issue with not all DataNodes being utilized as described in 9.3.3.*

[9] chat.openai.com