BDTT

Week 3

File Types in Big Data Analysisa & Resilient Distributed Datasets (RDDs)

2025

1. Learning Outcomes

- **To learn data types in Python:** Discover Python's data types and how they influence code structure.
- **To know the main file types in Big Data analysis:** Recognise common file formats that underpin large-scale data workflows.
- **To understand CSV and JSON structures in more in-depth:** Examine how CSV and JSON formats are structured and parsed for efficient data handling.
- **How to Create an RDD:** Learn the steps to instantiate Resilient Distributed Datasets within Spark.
- **RDD Operations:** Explore transformations and actions that facilitate data processing across distributed collections.
- **RDD Types:** Familiarise yourself with various RDD categories suited to different data scenarios.
- **RDD & Lazy Execution:** Understand how Spark delays execution for performance optimisation.
- **Functional programming in Spark:** Discover how adopting functional approaches enhances distributed computing.
- **Pair RDDs:** Learn how key-value pair RDDs expand data manipulation capabilities in Spark.

2. Data Types in Python

2.1. Classes and Objects

Python classes define a blueprint or template for creating objects, providing structure for related attributes and methods. They enable you to encapsulate data and behaviour within a logical unit, promoting clarity and maintainability in your code. Objects, meanwhile, are initialised instances of classes that carry their own data, enabling interaction with class methods to achieve specific tasks. This object-oriented approach fosters robust, modular programmes that can be seamlessly extended and adapted to meet evolving requirements.

Python classes act as blueprints that define attributes (data) and methods (behaviour), and objects are individual instances of those classes. For example, imagine a simple

‘Car’ class that stores a car’s colour, brand, and mileage, with a method to simulate driving. By creating multiple objects from the same ‘Car’ class (e.g. my_car and your_car), each object can have its own distinct attributes, yet still share the same attributes and methods defined in the class.

2.2. Mutable vs Immutable Objects

Mutable objects can be compared to a notepad on which you can keep updating your tasks, crossing them out, or adding new ones whenever you like. Each time you write something different, you are changing the same notepad rather than creating a fresh one.

In contrast, **immutable objects** are like a sealed letter: once written and sealed, the receiver can read the contents but you cannot amend them without breaking the seal and essentially rewriting the whole message. This distinction underlines how some things, like your to-do list, can be repeatedly changed, while others, like a sealed letter, must stay the same once finalised.

2.3. Built-in Classes in Python

Numbers are Python’s classes for representing numeric data, encompassing integers, floating-point numbers, and complex numbers. They allow you to perform mathematical operations and handle various computational tasks.

Bool stands for Boolean values, signifying truth or falsity (True or False) and commonly used in conditional expressions or logical tests.

A **Set** is an unordered collection of unique items, meaning no duplicates are allowed. Sets are designed for efficient membership checks and operations such as union, intersection, and difference.

A **Dict** (short for ‘dictionary’) stores data in key-value pairs, allowing fast lookups based on keys. Each key must be unique, whereas values can be of any type.

A **Sequence** represents an ordered series of elements and includes types like lists, tuples, and strings. These structures allow indexing, slicing, and iteration to access or traverse the items they contain.

3. Big Data File Formats

A **file format** is the structure of a file that tells a programme how to display and process its contents. In other words, a file format, sometimes referred to as a file extension, sets out how data is organised within the file. This organisation ensures that your computer or chosen software can correctly interpret and show the information, whether it consists of text, images, audio, or other forms of data. Main Big Data file formats are:

- **CSV (Comma-Separated Values)**
    - Straightforward, human-readable format where each line represents a record, and each value is separated by a comma.
    - Best suited for simpler datasets or scenarios involving data interchange with users who prefer easy-to-edit text files.
- **JSON (JavaScript Object Notation)**
    - Lightweight, text-based structure often used for storing and transmitting data in attribute–value pairs.
    - Excellent for web-based applications and services that require data exchange in a flexible, hierarchical structure.
- **Parquet (Apache Parquet)**
    - Columnar storage format designed for efficient compression and encoding.
    - Ideal for analytical workloads where you only need to query specific columns rather than entire records.
- **AVRO (Apache Avro)**
    - A row-based storage format known for its compact binary representation and schema evolution support.
    - Particularly useful in scenarios where schemas might change over time and backward compatibility is required.
- **ORC (Optimised Row Columnar)**
    - Columnar storage format offering high compression ratios and

performance benefits for analytics.

- Typically used in data warehousing and complex queries where faster reads from specific columns matter.

In the BDTT module our focus is on using CSV and JSON files.

Delimiters in a file are special symbols or characters used to separate distinct pieces of data, making it easier for programmes to read and process the information. Common delimiters include commas, semicolons, tabs, spaces, and pipes. For example:

A CSV file might use commas (,), producing something like **John,31,London**

While a tab-delimited file separates values with the tab character, such as

John 31 London

In other cases, pipes (|) or semicolons (;) may also be used, for instance **John|31|London** or **John;31;London**.

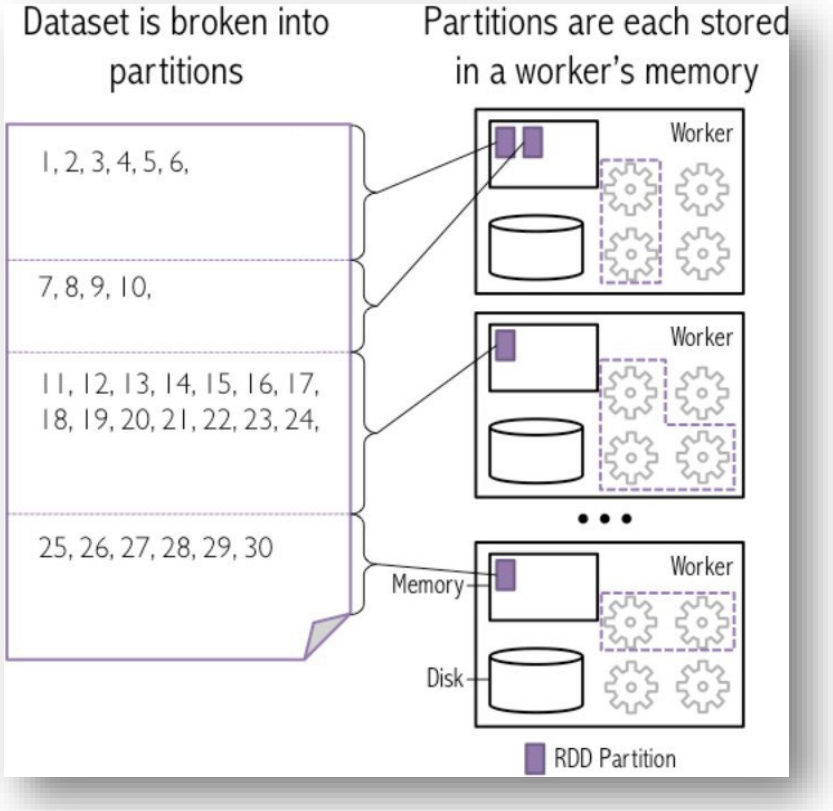
4. What is an RDD?

An RDD **(Resilient Distributed Dataset)** is like having a large container of items (data) shared among multiple helpers (computers). Each helper processes its portion in parallel, then everything is gathered together for a final result. For instance, imagine you've got a massive stack of papers to sort for a project: you pass some papers to each classmate, they all sort at the same time, and afterwards you collect their sorted piles back into one complete, tidy set.

RDDs are the core data structure in Apache Spark, enabling distributed computing across large datasets. Spark builds upon RDDs to provide fault-tolerant, parallelised data operations, ensuring that even if parts of the cluster fail, the dataset and processing can be automatically recovered. This allows developers to focus on their transformations and analyses, whilst Spark coordinates the data distribution and execution details behind the scenes.

Partitioning is the act of splitting data into smaller sections so multiple worker nodes can process them in parallel and speed up the entire operation. It's like cutting a huge

pizza into slices for a group of mates: instead of one person having to eat the whole pizza, everyone gets their own slice to enjoy at the same time. In the same way, Spark takes large datasets and breaks them into smaller partitions that can be dispatched to each worker node. Each node tackles its portion independently, accelerating the overall workload through parallel processing.



5. Three ways to create an RDD:

1. **From a file or set of files**
2. **From data in memory**
3. **From another RDD**

1. **From a file or set of files:**

You can load data stored externally, such as a CSV or text file, to create an RDD. For example, imagine you've saved a large text file containing all your customer orders; Spark can read this file from a distributed system (like HDFS, DBFS) or your local machine, producing an RDD of order records ready for analysis.

2. **From data in memory:**

If you have a small collection of data already available in your programme (like a list of sales figures), you can directly transform that into an RDD. For instance, suppose your application generates a short list of user IDs in memory; you can create an RDD from this list so multiple worker nodes can process the IDs in parallel.

3. **From another RDD:**

Once you have an existing RDD, you can apply a transformation, such as filtering or mapping, to produce a new one. For example, if you have an RDD of website traffic logs, you might create a second RDD containing only visits from a specific country by filtering out all other entries.

6. RDD Operations:

RDD operations are commands or instructions you apply to Resilient Distributed Datasets in order to manipulate or retrieve the data they hold. They help you shape your datasets by creating new versions or by extracting results once the data is ready for final analysis.

- **Transformations**
  These create new RDDs from existing ones but do not immediately produce a final outcome. For example, if you have a list of fruits, you could transform it by filtering out everything except apples, resulting in a new RDD that contains only apples.

- **Actions**
  These produce the final result or return a value to the driver. Sticking with the fruit analogy, once you've filtered to apples, you might run a count action to

determine how many apple entries are in the new RDD. When the action is executed, Spark completes all the needed work and returns the result.

7. RDD and Lazy Execution:

RDD & Lazy Execution means Spark waits to do the heavy lifting until a final action is called. Think of it like planning out your weekly grocery run: you decide which shops you'll visit and what items you need beforehand, but you only hop into the car and start the actual shopping when you absolutely need the groceries. Until then, it's all just planning on paper.

When you apply **transformations** to an RDD, Spark records these operations in a directed acyclic graph (DAG), effectively mapping out how to obtain the required results. However, it does not carry out any data processing at this stage. Only when an **action** (such as counting or collecting results) is requested does Spark active the DAG, executing the transformations in the most efficient order possible. This delay in computation is what we call "lazy execution." It allows Spark to combine multiple transformations, optimise the tasks, and run them once—rather than reprocessing data each time a transformation is defined.

8. Functional Programming in Spark

Functional Programming in Spark emphasises the use of functions and immutable data, allowing for more predictable and maintainable code when working with distributed datasets. Rather than mutating shared state, Spark transformations produce new Resilient Distributed Datasets (RDDs) at each step, which streamlines parallel processing and fault tolerance. This approach ensures cleaner, more modular programmes, laying the groundwork for robust, scalable data analysis. There are two types of functions in python; named functions and anonymous (lambda) functions.

Named functions in Python are defined with an explicit identifier, making them easy to reuse and reference throughout your programme. For example, you might create a function called *calculate_discount*, which takes an item's price and returns the reduced amount after applying a discount. You can then call *calculate_discount* whenever you need to handle similar calculations.

Anonymous (lambda) functions, on the other hand, are defined without a name. They are designed for short, one-off tasks like a quick arithmetic expression or a simple check in a list of numbers. Because they're concise and usually written in a single line, they're often passed directly into functions that expect a quick action, such as filtering out odd numbers or adding up two values on the fly.

9. Main RDD Operations:

In Apache Spark, transformations and actions form the essence of data manipulation on Resilient Distributed Datasets. Transformations specify how data should be shaped, filtered, or combined, while actions finally execute the plan and produce a result. This design fits neatly into functional programming principles, emphasising pure, stateless functions and immutable data throughout each step. Many of these operations accept functions defined as lambdas, enabling concise yet powerful data processing. By imposing the heavy processing to Spark only when required, these operations streamline performance and maintain clarity in distributed data workflows.

Transformations

- **map** – Applies a function to each element, returning a new RDD.
- **filter** – Keeps only those elements that meet a specified condition.
- **flatMap** – Similar to map, but flattens nested structures into a single list.
- **distinct** – Removes duplicate elements, producing an RDD of unique records.
- **union** – Combines two RDDs into one, preserving all elements from both.
- **intersection** – Returns only the elements present in both RDDs.
- **reduceByKey** – Aggregates values by key (often used for summing or averaging).
- **groupByKey** – Groups values by key, producing a collection of values for each.
- **sortByKey** – Sorts the RDD based on keys in ascending or descending order.
- **join** – Merges two key-value RDDs, matching pairs by shared keys.

Actions

- **collect** – Retrieves all data from the RDD to the driver as a single collection.
- **count** – Counts the total number of elements in the RDD.
- **first** – Returns the first element of the RDD.
- **take** – Fetches a specific number of elements from the RDD.
- **reduce** – Aggregates all elements of the RDD using a specified operation.
- **countByValue** – Counts how often each unique element occurs in the RDD.
- **foreach** – Applies a function to each element without returning a new RDD.
- **saveAsTextFile** – Stores the RDD's contents as text files in the specified location.