# Setup

## Databricks

The Databricks Data Intelligence Platform allows the organisation to use data and AI. It is built on a lakehouse to provide an open, unified foundation for all data and governance and is powered by a Data Intelligence Engine that understands the uniqueness of data. This understanding allows the Databricks Platform to automatically optimise performance and manage infrastructure in ways unique to the business. Moreover, the Data Intelligence Engine understands the organisation’s language, so searching and discovering new data is as easy as asking a question like one normally would to a coworker. Additionally, developing new data and applications is also accelerated through natural language assistance to write code, debug errors, and find answers. Those data and applications require strong governance and security, especially with the birth of generative AI. Databricks provides an end-to-end MLOps and AI development solution built upon a unified approach to governance and security, enabling the customers to pursue all their AI initiatives, from using APIs like OpenAI to custom-built models without compromising data privacy and IP control (Databricks, n.d.).

## PySpark

PySpark is the Python API for Apache Spark. It enables users to perform real-time, large-scale data processing in a distributed environment using Python. It also provides a PySpark shell for interactively analysing the data. PySpark combines Python’s learnability and ease of use with the power of Apache Spark to enable the processing and analysis of data at any size for everyone familiar with Python. PySpark supports Spark’s features such as SparkSQL, DataFrames, Structured Streaming, Machine Learning (MLlib) and SparkCore (Apache Software Foundation, 2025).

## Spark SQL

Spark SQL is Apache Spark’s interface for working with structured and semi-structured data. In particular, Spark SQL provides three main capabilities, including loading data from a variety of structured sources, querying the data using SQL that is located inside a Spark program and from external tools that connect to Spark SQL through standard database connectors (JDBC/ODBC), such as business intelligence tools like Tableau, providing rich integration within a Spark program between SQL and regular Python/Java/Scala code, including the ability to join RDDs and SQL tables, expose custom functions in SQL, and more, which is easier for jobs to write using this combination. Spark SQL implements these capabilities by providing a special type of RDD called SchemaRDD. A SchemaRDD is an RDD of row objects, each representing a record. In addition, they provide new operations not available on RDDs, such as the ability to run SQL queries. SchemaRDDs can be created from external data sources, query results, or regular RDDs (Karau et al., 2015).

## SparkContext

A SparkContext is a main entry point for Spark functionality representing the connection to a Spark cluster and can be used to create RDD and broadcast variables on that cluster (Apache Software Foundation, n.d.).

## SparkSession

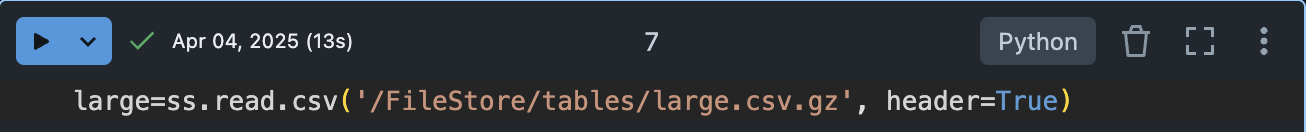
A SparkSession is an entry point to programming Spark with the Dataset and DataFrame API, which can be used to create DataFrame, register DataFrame as tables, execute SQL over tables, cache tables, and read parquet files (Apache Software Foundation, n.d.).

# Problems

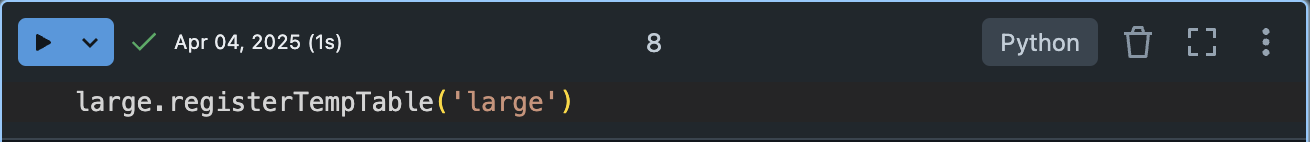
## Question 1

The assumption states that the rows within the sentence column can be duplicated, meaning a sentence can be represented in multiple sources. This duplication happens because a sentence can be referenced multiple times in various documents. Therefore, it requires distinctive counting.

The implementation outline includes reading a large CSV file, registering that CSV file as a temporary table, and querying the number of unique sentences.



**Figure 1.1: Reading a CSV file**



**Figure 1.2: Registering a temporary table**

A black background with white text

Description automatically generated

**Figure 1.3: Querying the number of unique sentences**

The result is 389639 unique sentences, which rejects the assumption.

A screenshot of a computer

Description automatically generated

**Figure 1.4: Question 1’s result**

The data is a CSV file containing a sentence and source header. Therefore, the read.csv method of the SparkSession object is used to read the CSV file by taking the path to the CSV file as a parameter. The ‘/File Store/tables/large.csv.gz’ is the path, where the ‘/File Store/tables’ is the directory, which is the Databricks file system root, and the ‘large.csv.gz’ is the file’s name. The header=True parameter indicates that the first row of the CSV file contains the column names, which are sentence and source. By executing this code, the CSV file will be read, and the data will be loaded into the DataFrame object called large.

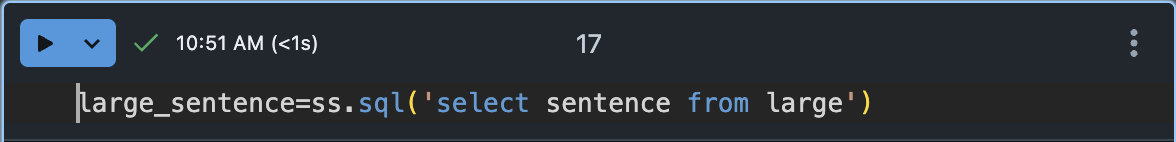
The registerTempTable method is called on the large DataFrame object to register it as a temporary table, which can perform SQL queries using SparkSQL. The registerTempTable method takes a table name as its argument, in this case, ‘large.’ This table name is used to identify the DataFrame in the SparkSQL. Once the DataFrame is registered as a temporary table, it can be referred to by its table name in SQL queries.

The sql method provided by the SparkSession object executes the SQL queries by taking a SQL query as a string parameter. The ‘select count(distinct sentence) large from large’ is the SQL query being executed. It selects the count of distinct sentences from a table name large and aliases the result as large. The show method displays the result of the SQL query in tabular format.

## Question 2

The assumption states that the number of words in the 10 longest sentences is the same, which means the 10 longest sentences have the same length.

The implementation outline includes creating a new DataFrame called a large sentence, writing that DataFrame as a CSV file, reading that CSV File as a text file, splitting the words that are separated by a space, creating a list of tuples containing the number of words in a sentence and a sentence, sorting the list by the number of words in a sentence by in descending order, creating a new list of components containing only the number of words in a sentence in descending order, returning the list of components including the number of words in descending order of the 10 longest sentences.



**Figure 2.1: Creating a new DataFrame**

A screen shot of a computer code

Description automatically generated

**Figure 2.2: Writing a DataFrame as a CSV file**

A screenshot of a computer

Description automatically generated

**Figure 2.3: Reading a CSV file as a text file**

A screenshot of a computer

Description automatically generated

**Figure 2.4: Operating an RDD**

The result includes the unique number of words in the 10 longest sentences, which rejects the assumption.

A screenshot of a computer code

Description automatically generated

**Figure 2.5: Question 2’s result**

The sql method of the SparkSession object executes an SQL query that selects the sentence column from the large table. The result of this query is stored in a DataFrame object called large\_sentence, which contains the data from the sentence column.

The large\_sentence DataFrame is then written to a CSV file using the repartition method to control the number of output files that will be generated, and it takes one as a parameter to ensure that the data is written to a single output file. The write.csv method writes the DataFrame to a CSV file by taking the path where the CSV file will be stored as a parameter. The ‘/FileStore/tables/large\_sentence’ is the path, where the ‘/File Store/tables’ is the directory, the Databricks file system root, and the ‘large\_sentence is the file’s name. The mode=overwrite parameter is used to overwrite the file if it already exists.

The CSV file is read as a text file using the textFile method of the SparkContext object, and this method takes the path to the CSV file as a parameter. The ‘/FileStore/tables/large\_sentence’ is the path, where the ‘/File Store/tables’ is the directory, the Databricks file system root, and the ‘large\_sentence is the file’s name. Once the file is read, it will be represented as a distributed collection of sentences, where each sentence is a separate component, which allows performing distributed operations on the data, such as counting the number of sentences, filtering specific sentences, or applying transformations to each sentence.

Each sentence in the large\_sentence file is split into individual words, then those words are returned to a list using the split method. The len function in a lambda expression calculates the number of those individual words in a list. The map method maps the number of those individual words in a list and the sentence itself to a tuple. The sortBy method sorts the tuples based on the number of those individual words by taking x[0] as a parameter. In contrast, the ascending=False parameter indicates the type of sorting is in descending order, which means that the biggest number of those individual words will appear first.

The map method maps each tuple to its first element by taking the x[0] parameter, which is the number of those individual words. This transformation returns a list of components containing the number of individual words.

The take method is an action, an RDD’s operation that returns a list of 10 components containing the number of individual words in descending order by taking 10 as a parameter.

## Question 3

The assumption states that the number of bigrams per sentence equals the number of sentences, resulting in an average of 1.

The implementation outline includes splitting the words that are separated by a space, creating a list of components containing the number of words in a sentence that is minus 1, and creating a list of lists containing those number of words and 1, where 1 represents the sentence, summing all those number of words and all the values 1.

A screen shot of a computer program

Description automatically generated

**Figure 3.1: Operating an RDD**

A screenshot of a computer code

Description automatically generated

**Figure 3.2: Calculating the average number of bigrams per sentence**

The result is different than 1, which rejects the assumption.

A screenshot of a computer

Description automatically generated

**Figure 3.3: Question 3’s result**

The question is answered using the split method, which splits the sentence into a list of words separated by a space. The len function calculates the number of words in a list, and then the result is subtracted by one because the number of bigrams is one less than the number of words.

The map method then maps each calculated number of bigrams and the value 1 to a list.

The reduce method sums the number of bigrams and the value 1, where 1 represents the number of sentences. The result is a list containing the sum of all calculated numbers of bigrams and the sum of all the values 1, representing the total number of sentences.

The final list contains two elements: the total number of bigrams and the total number of sentences, where the total number of bigrams is the first element with an index of 0. In contrast, the total number of sentences is the second element with an index of 1. The average number of bigrams per sentence is calculated by dividing the first element by the second element of the list, which is the average\_number\_bigrams\_large\_sentence.

## Question 4

The assumption states that the 10 most frequent bigrams have the same frequency.

The implementation outline includes splitting the words that are separated by a space, creating a list of lists containing the words that have been split, writing a function that loops through the lists within that list and creates bigrams by combining two consecutive words, mapping that function to the list of lists that returns a list of components containing the bigrams, creating a list of tuples containing the bigrams and the value 1 representing the number of times a bigram presents, summing all the values 1, which have the same bigrams, sorting the list by the sum of all the values 1, which represents the total number of times a bigram presenting in descending order, returning the most frequent bigrams as a list of tuples.

A screenshot of a computer

Description automatically generated**Figure 4.1: Splitting the words that are separated by a space**

A screen shot of a computer code

Description automatically generated

**Figure 4.2: Creating the bigrams**

A screenshot of a computer

Description automatically generated

**Figure 4.3: Mapping the function**

A screenshot of a computer code

Description automatically generated

**Figure 4.4: Operating an RDD**

The result returns the 10 most frequent bigrams with unique frequency, which rejects the assumption.

A screen shot of a computer

Description automatically generated**Figure 4.5: Question 4’s result**

The split method splits each sentence into a list of words separated by a space, while the map method returns a list of lists.

The concatenate\_word function is defined to take a list within the list of lists as input and return a list of two consecutive words by first creating an empty list called my\_list to store the two consecutive words, using a for loop to iterate over the indices of the list, from the first to the second to last index. The i variable represents the current index, while the j variable represents the following index by adding 1 to the current index i. Hence, the function only iterates to the second to last index. Otherwise, the value of index j will exceed the index range. Then, it appends the two consecutive words at index i and j in the list to the my\_list, which is separated by a space. Finally, it returns that list.

The list of lists containing the two consecutive words is transformed using the flatMap method into a list of components containing two consecutive words.

The map method within the lambda function maps each two consecutive words with the value 1, representing the number of times a bigram presents to a tuple.

The reduceByKey is a transformation and RDD’s operation that sums all the values 1, which have the same bigrams, resulting in a list of tuples containing the bigrams and the total number of times a bigram presents.

The sortBy method sorts the list based on the total number of times a bigram is presented in descending order by specifying the ascending=False parameter.

The take method returns a list of tuples containing the 10 most frequent bigrams by taking 10 as a parameter.

## Question 5

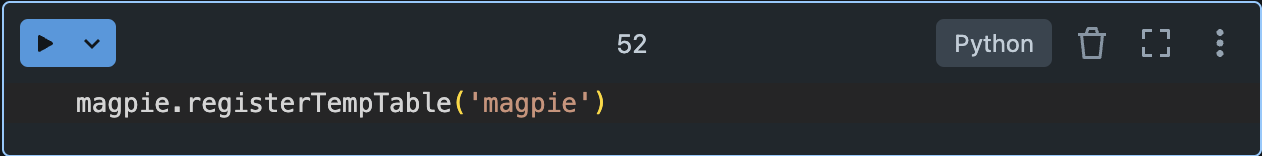
The assumption states that no bigrams from the Wikipedia subset appear in the list of idioms contained in the MAGPIE subset.

The implementation outline includes reading the MAGPIE file, which formats as a JSON, registering that file as a temporary table, splitting the words that are separated by a space, creating a list of lists containing the words that have been split, writing a function that loops through the lists within that list and creates bigrams by combining two consecutive words, mapping that function to the list of lists that returns a list of components containing the bigrams, creating a list of lists containing the bigrams, creating a DataFrame with one column named bigrams that contains the values are the bigrams within those lists, registering that DataFrame as a temporary table, querying the bigrams that have been extracted from the Wikipedia subset appear in the list of idioms contained in the MAGPIE subset.

A black background with white text

Description automatically generated

**Figure 5.1: Reading a JSON file**



**Figure 5.2: Registering a temporary table**

A screenshot of a computer

Description automatically generated

**Figure 5.3: Splitting the words that are separated by a space**

A screen shot of a computer code

Description automatically generated

**Figure 5.4: Creating the bigrams**

A screenshot of a computer

Description automatically generated

**Figure 5.5: Mapping the function**

A screen shot of a computer

Description automatically generated

**Figure 5.6: Operating an RDD**

A screenshot of a computer

Description automatically generated

**Figure 5.7: Querying the bigrams**

The result returns 67 bigrams extracted from the Wikipedia subset and appears in the list of idioms contained in the MAGPIE subset, which rejects the assumption.

A screenshot of a computer

Description automatically generated

**Figure 5: Question 5’s result**

The read.json method reads a JSON file and then returns the file as a DataFrame object.

This DataFrame object is then registered as a temporary table, which SQL can query to interact with the data.

The split method splits each sentence into a list of words separated by a space, while the map method returns a list of lists.

The concatenate\_word function is defined to take a list within the list of lists as input and return a list of two consecutive words by first creating an empty list called my\_list to store the two consecutive words, using a for loop to iterate over the indices of the list, from the first to the second to last index. The i variable represents the current index, while the j variable represents the following index by adding 1 to the current index i. Hence, the function only iterates to the second to last index. Otherwise, the value of index j will exceed the index range. Then, it appends the two consecutive words at index i and j in the list to the my\_list, which is separated by a space. Finally, it returns that list.

The list of lists containing the two consecutive words is transformed using the flatMap method into a list of components containing two consecutive words.

The map method maps each bigram to a list, which creates a list of lists containing the bigrams. The toDF method creates a DataFrame containing one column named bigrams that contains the bigrams within those lists. That DataFrame is then registered as a temporary table named large\_bigrams.

The sql method of the SparkSession object takes a string as an argument, which represents the SQL query to be executed. The ‘select count(distinct bigrams) large from large\_bigrams where bigrams in (select idiom from magpie)’ is the SQL query being executed, which counts the unique bigrams within the temporary table and filters the bigrams from the Wikipedia subset that appears in the list of idioms contained in the MAGPIE subset.

The show method returns the result of the SQL query.

## Question 6

The assumption is that the bigrams appear on Wikipedia and MAGPIE. Moreover, the 10 bigrams, from rank 2500 to rank 2510, have the same frequency.

The implementation outline includes splitting the words that are separated by a space, creating a list of lists containing the words that have been split, writing a function that loops through the lists within that list and creates bigrams by combining two consecutive words, mapping that function to the list of lists that returns a list of components containing the bigrams, creating a list of tuples containing the bigrams and the value 1 representing the number of times a bigram presents, summing all the values 1, which have the same bigrams, creating a DataFrame with two columns named bigram and frequency that contains the values are the bigrams within those tuples and the frequency of that bigram, registering that DataFrame as a temporary table, querying the 10 bigrams that appear in Wikipedia and not in MAGPIE starting from rank 2500 to rank 2510 in descending frequency order.

A screenshot of a computer

Description automatically generated

**Figure 6.1: Splitting the words that are separated by a space**

A screen shot of a computer code

Description automatically generated

**Figure 6.2: Creating the bigrams**

A screenshot of a computer

Description automatically generated

**Figure 6.3: Mapping the function**

A screenshot of a computer program

Description automatically generated

**Figure 6.4: Operating an RDD**

A screenshot of a computer code

Description automatically generated

**Figure 6.5: Querying the bigrams**

The result returns the 10 bigrams with the same frequency that appear in Wikipedia and not in MAGPIE starting from rank 2500 to rank 2510 in descending frequency order, which rejects the assumption that states the 10 bigrams starting from rank 2500 to rank 2510 appear in Wikipedia and MAGPIE. However, it can not reject the assumption that the 10 bigrams, starting from rank 2500 to rank 2510, have the same frequency.

A screenshot of a computer program

Description automatically generated

**Figure 6: Question 6’s result**

The split method splits each sentence into a list of words separated by a space, while the map method returns a list of lists.

The concatenate\_word function is defined to take a list within the list of lists as input and return a list of two consecutive words by first creating an empty list called my\_list to store the two consecutive words, using a for loop to iterate over the indices of the list, from the first to the second to last index. The i variable represents the current index, while the j variable represents the following index by adding 1 to the current index i. Hence, the function only iterates to the second to last index. Otherwise, the value of index j will exceed the index range. Then, it appends the two consecutive words at index i and j in the list to the my\_list, which is separated by a space. Finally, it returns that list.

The list of lists containing the two consecutive words is transformed using the flatMap method into a list of components containing two consecutive words.

The map method within the lambda function maps each two consecutive words with the value 1, representing the number of times a bigram presents to a tuple.

The reduceByKey is a transformation and RDD’s operation that sums all the values 1, which have the same bigrams, resulting in a list of tuples containing the bigrams and the total number of times a bigram presents.

The toDF method creates a DataFrame containing two columns named bigram and frequency, which contains the bigrams within those tuples and the frequency of that bigram. That DataFrame is then registered as a temporary table named large\_bigrams.

The sql method of the SparkSession object takes a string as an argument, which represents the SQL query to be executed. The ‘select \* from (select bigram, frequency, dense\_rank() over(order by frequency desc, bigram asc) rank from large\_bigram where bigram not in (select idiom from magpie)) where rank between 2500 and 2510’ is the SQL query being executed, which selects the bigram, frequency, and the rank that is calculated using the dense\_rank method. The dense\_rank method assigns a rank to each row based on the frequency of a bigram in descending order, and the bigrams that have the same frequency are ordered based on the alphabet. The where clause filters the bigrams that appear in Wikipedia, not MAGPIE. Moreover, the other where clause filters the bigrams that rank between 2500 and 2510.

The show method displays the 10 bigrams with the same frequency that appear in Wikipedia and not in MAGPIE, starting from rank 2500 to rank 2510 in descending frequency order and alphabetical order for the bigrams with the same frequency.

# References

* Databricks. (n.d.). Data intelligence platform. Databricks. <https://www.databricks.com/product/data-intelligence-platform>
* Apache Software Foundation. (2025, February 23). PySpark documentation (Version 3.5.5). <https://spark.apache.org/docs/latest/api/python/index.html>
* Apache Software Foundation. (n.d.). pyspark.SparkContext — PySpark 3.5.0 documentation. <https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.SparkContext.html>
* Apache Software Foundation. (n.d.). pyspark.sql.SparkSession — PySpark 3.5.0 documentation. <https://spark.apache.org/docs/latest/api/python/reference/pyspark.sql/api/pyspark.sql.SparkSession.html>
* Karau, H., Konwinski, A., Wendell, P., & Zaharia, M. (2015). Learning Spark: Lightning-fast big data analysis. O’Reilly Media, Inc.