**INTERIM PROJECT**

**NAME:** SUDHEER KUMAR CHAKKA

**EMP ID :** 2320418

**COHORT :** CSDAIA24GP002

**DESIGNATION:** GENC INTERN

**TITLE:**

Build an Inverted Index by mapping each word to the set of Document IDs wherever it appears.

**INTRODUCTION:**

**Background:**

 An inverted index is a data structure used to create a full-text search. In simple terms, it is a hashmap like data structure that directs you from a word to a document or a web page. It is the heart of any search engine.

**Importance of Inverted Index:**

 Inverted Index is one of the main components of Search Engines. It is used to retrieve a list of documents that contain a particular word. This is crucial in information retrieval systems and is used to make full-text searches faster.

**Forward Index:**

1.It is a data structure that stores mapping from documents to words i.e.     directs you from document to word.

2.Steps to build Forward index are:

* Fetch the document and gather all the keywords.
* Append all the keywords in the index entry for this document.
* Repeat above steps for all documents

3. Indexing is quite fast as it only append keywords as it move forwards.

4. Searching is quite difficult as it has to look at every contents of index just                to retrieve all pages related to word.

5. Example of forward index:

**Document**                     **Keywords**

doc1                          hello, **sky**, morning

doc2                          tea, coffee, hi

doc3                          greetings, **sky**

1. It stores duplicate keywords in index. Eg: word “sky” is stored multiple times.

2. Real life examples of Forward index:

* Table of contents in book.

**Inverted Index:**

Inverted indexes are widely used in various applications including search engines, text mining, and natural language processing. By mapping words to the documents they appear in, we can quickly locate documents containing a particular word, making it a powerful tool for these applications.

1. It is a data structure that stores mapping from words to documents or set of documents i.e. directs you from word to document.

2. Steps to build Inverted index are:

* Fetch the document and gather all the words.
* Check for each word, if it is present then add reference of document to index else create new entry in index for that word.
* Repeat above steps for all documents and sort the words.

3. Indexing is slow as it first checks that word is present or not.

4. Searching is very fast.

5. Example of Inverted index:

**Word**                       **Documents**

hello                         doc1

**sky**                       doc1, doc3

coffee                               doc2

hi                            doc2

greetings                     doc3

1. It does not store duplicate keywords in index.

2. Real life examples of Inverted index:

* Index at the back of the book.

**OBJECTIVE:**

The objective of this project is to build an inverted index using PySpark RDDs. An inverted index is a data structure that maps each unique word to the set of document IDs in which it appears. This is a fundamental concept in information retrieval systems and is used to speed up full-text searches.

The reason for using Spark and PySpark in this project is because of their ability to process large amounts of data quickly and efficiently. Spark’s core data structure, the Resilient Distributed Dataset (RDD), allows data to be distributed across multiple nodes in a cluster for parallel processing. This is particularly useful when working with large text files for building an inverted index. PySpark provides the Python API for Spark, which makes it easier to implement such a project using Python.

**REAL-TIME USECASE:**

A real-time use case of this project could be a search engine like Google. When a user types a query into the search bar, Google uses an inverted index to find all web pages containing the query words, and then ranks these pages based on various factors like page relevance and importance.

**METHODOLOGY:**

**Understanding Inverted Index:**

An inverted index is a dictionary where each word is associated with a list of documents in which that word appears. This allows for efficient full-text searches.

**Tools and Technologies Used:**

The main tools used in this project are Spark and PySpark. Spark is an open-source, distributed computing system used for big data processing and analytics. PySpark is the Python library for Spark that lets you harness the simplicity of Python and the power of Apache Spark in order to tame Big Data.

**IMPLEMENTATION:**

**Data Collection:**

 The first step in building an inverted index is to collect the data. In this case, the   data is a set of text files.

**Preprocessing:**

 The data is then preprocessed by tokenizing the text (splitting the text into words), normalizing (converting to lowercase), and removing punctuation and stop words (common words like ‘is’, ‘the’, ‘and’, etc. that do not carry much meaning).

**Building the Inverted Index:**

The preprocessed data is then used to build the inverted index. Each word is mapped to the set of document IDs in which it appears.

**Mapping Words to Document IDs:**

Each word is mapped to the set of document IDs in which it appears.

**DATA COLLECTION:**

There are five text file documents we are considering to build an inverted index.



**Filename : InterimDoc\_1.txt**

A screenshot of a computer

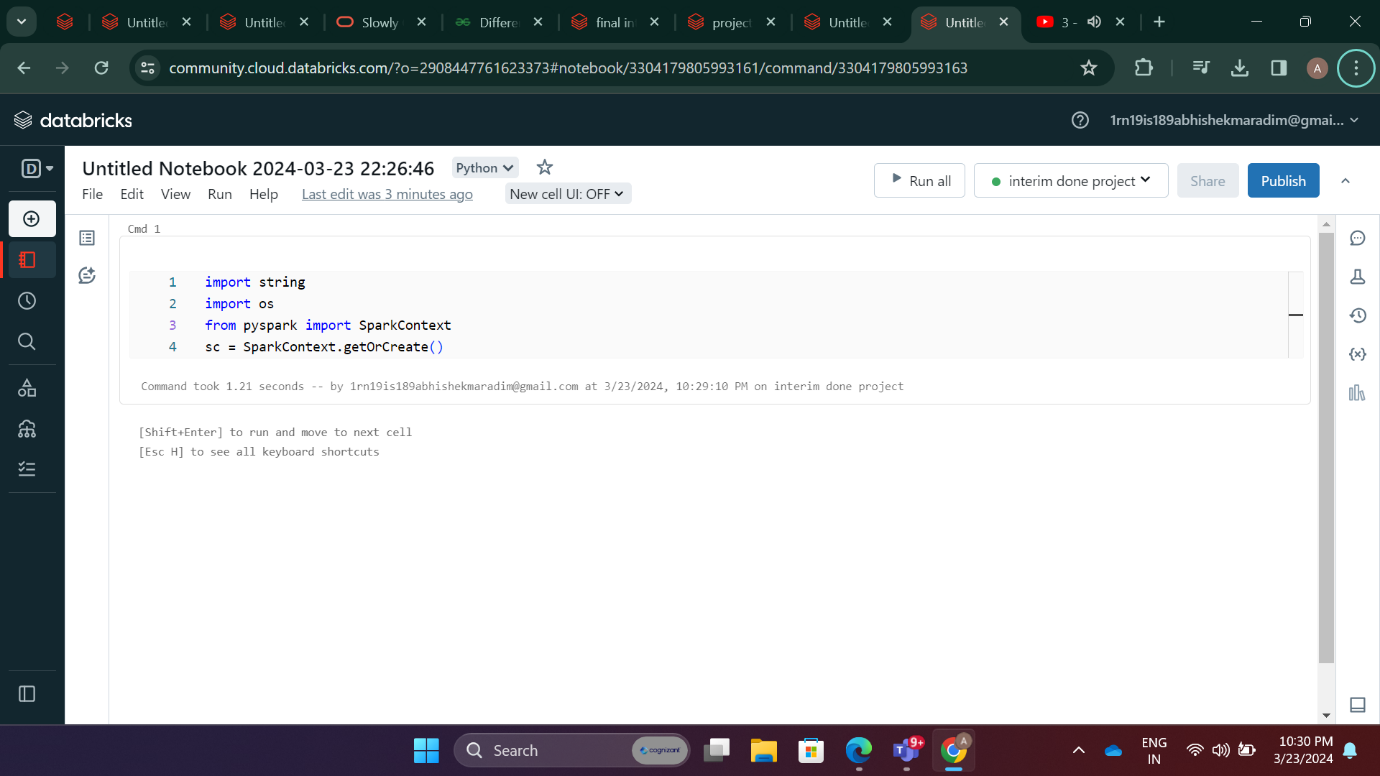
Description automatically generated

**Filename:InterimDoc\_2.txt**

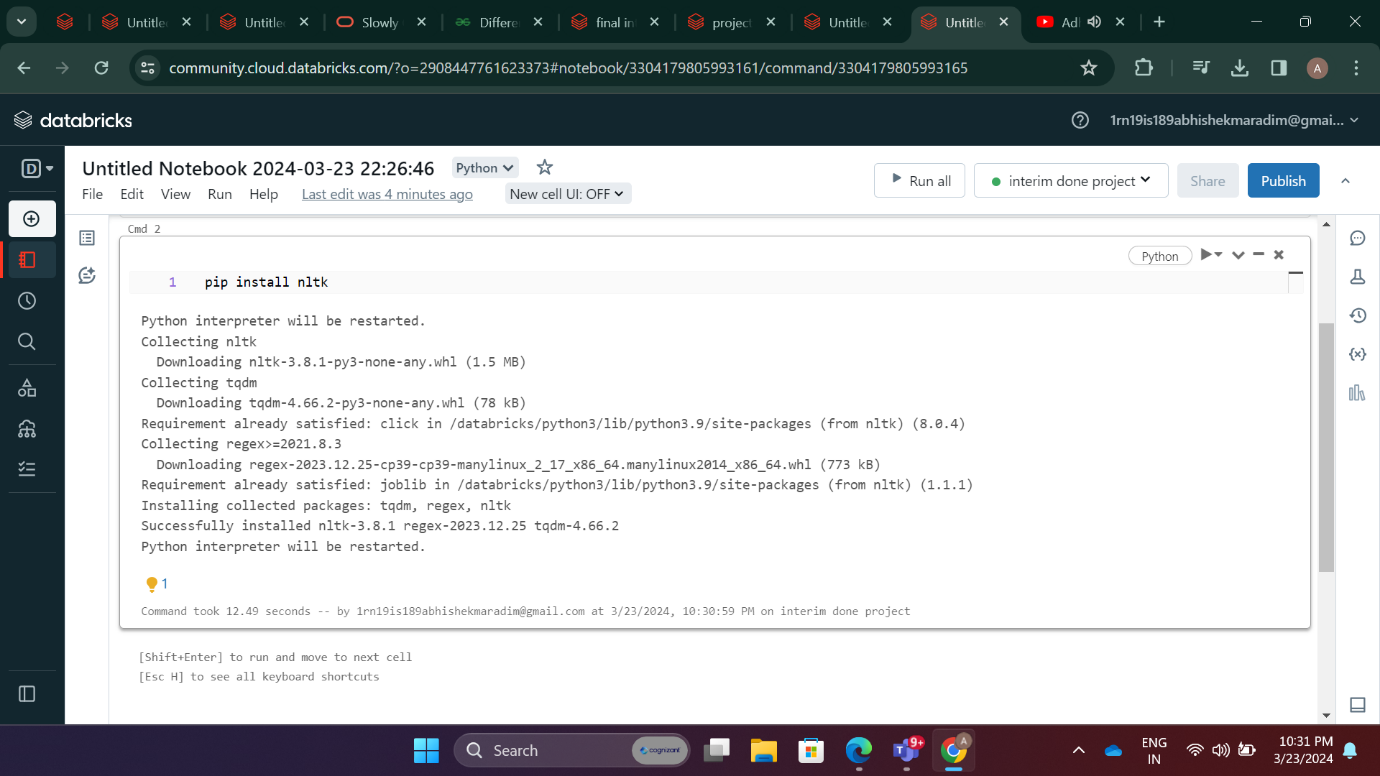
A screenshot of a computer

Description automatically generated

**Filename:InterimDoc\_5.txt**



* **Cell 1:** Here, we’re importing the SparkContext from PySpark and creating an instance of it. This is the entry point for any functionality in Spark.



* **Cell 2:** We’re installing the Natural Language Toolkit (NLTK), a library in Python that provides tools for working with human language data.

A screenshot of a computer

Description automatically generated

* **Cell 3:** We’re defining the file paths of the text files that we’ll be working with.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* **Cell 4-5:** We’re reading each text file into a separate Resilient Distributed Dataset (RDD). Then, we’re using the collect() action to return all the elements of the RDD as an array to the driver program.

A screenshot of a computer

Description automatically generated

* **Cell 6:** We’re transforming each RDD by applying a map function that returns a tuple containing a unique ID for each document and its content..

A screenshot of a computer

Description automatically generated

* **Cell 7:** We’re creating a new RDD (documents) that contains the data from all the previous RDDs. This is done using the union() transformation.

A screenshot of a computer

Description automatically generated

* **Cell 8:** We’re downloading the Punkt Tokenizer Models. This is a pre-trained unsupervised machine learning tokenizer. It divides a text into a list of sentences.

A screenshot of a computer

Description automatically generated

* **Cell 9:** We’re tokenizing the content of each document using the Punkt tokenizer.

A screenshot of a computer

Description automatically generated

* **Cell 10:** We’re normalizing the tokens by removing punctuation and converting all the words to lowercase.

A screenshot of a computer

Description automatically generated

* **Cell 11:** We’re downloading the list of English stop words from NLTK, which are common words that do not contain important meaning and are usually removed from texts.

A screenshot of a computer

Description automatically generated

* **Cell 12:** We’re lemmatizing the words (converting them to their base form, or lemma) and removing stop words.

A screenshot of a computer

Description automatically generated

* **Cell 13:** We’re creating pairs of words and their corresponding document IDs.

A screenshot of a computer

Description automatically generated

* **Cell 14:** We’re grouping the word-document pairs by word.

A screenshot of a computer

Description automatically generated

* **Cell 15:** We’re creating the inverted index by mapping each word to the set of document IDs it is associated with.

A screenshot of a computer

Description automatically generated

* **Cell 16:** We’re replacing the document IDs in the inverted index with their corresponding filenames.

A computer screen shot of a chat

Description automatically generated

* **Cell 17:** We’re printing the inverted index.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

* **Cell 18:** We’re defining a function that allows a user to search for a word in the inverted index and returns the files where the word is found.

**RESULTS:**

**Efficiency of the Inverted Index:**

The efficiency of the inverted index can be evaluated by its speed and accuracy            in retrieving the relevant documents that contain a particular word.

**Use Cases:**

The inverted index is a fundamental component in many applications including search engines, text mining, and natural language processing.

**CONCLUSION:**

The project successfully implemented an inverted index using PySpark and RDDs. The efficiency of the inverted index was evident in its ability to quickly map words to the documents they appear in, thereby facilitating fast and efficient full-text searches. This is a testament to the power and flexibility of Spark and PySpark, which allow for distributed processing of large datasets across a cluster of machines.

The use of RDDs was particularly beneficial in this project. RDDs, or Resilient Distributed Datasets, are a fundamental data structure in Spark that inherently provide fault tolerance through lineage information and can be processed in parallel across a cluster. This allowed for efficient manipulation and transformation of data when building the inverted index.

**FUTURE WORK:**

While the current implementation of the inverted index serves its purpose well, there are several ways in which this project could be extended or improved in the future:

1. **Support for More Complex Queries:**

 The inverted index could be extended to support more complex queries, such as multi-word or phrase queries. This would involve modifying the indexing process to consider phrases or groups of words, rather than just individual words.

1. **Document Ranking:**

Currently, the inverted index simply returns the set of documents that contain a particular word. A potential improvement could be to rank the documents by relevance to the query. This could be achieved by implementing a ranking algorithm, such as TF-IDF (Term Frequency-Inverse Document Frequency) or BM25.

1. **Use of DataFrames:**

While RDDs were used in this project, future work could explore the use of Spark DataFrames, which offer a more structured and powerful interface for large-scale data processing. DataFrames also have better optimization through Catalyst and Tungsten, which could potentially lead to performance improvements.

1. **Scalability Testing:**

Further testing could be done to evaluate the scalability of the solution as the size of the dataset increases. This would provide valuable insights into the performance and efficiency of the inverted index at scale.

1. **Integration with a Front-End Application:**

The inverted index could be integrated with a front-end application to provide a user-friendly interface for querying the index.

These improvements would not only enhance the functionality of the inverted index but also provide an opportunity to delve deeper into the capabilities of Spark, PySpark, RDDs, and DataFrames. They represent exciting avenues for future exploration and learning.