



LAB4_Report

CIS612 - Big Data and Parallel Data Processing Systems

Submitted by Aditya Sairam Pullabhatla

CSU ID: 2863159

Part 1: Building the Inverted Index

- **Objective**
- **Steps**
 - **Data Preprocessing in Python**
 - **Generating SQL Inserts in Python**
 - **Creating SQL Tables for Dictionary and Posting**
 - **Populating the Tables**
- **Bonus Part**

Outputs

3.Part 2: Calculating TF-IDF and Cosine Similarity for Query Terms

- **Objective**
- **Steps**
 - **Calculating Total Document Count**
 - **Creating and Populating DocumentVectors Table**
 - **Creating QueryVector Table for Query Terms**
 - **Calculating Cosine Similarity**
 - **Retrieving Top Documents Based on Similarity**

Bonus Part

Outputs

Overview:

This lab focuses on creating an inverted index, calculating TF-IDF values, and implementing cosine similarity to retrieve relevant documents based on query terms. These concepts are fundamental in information retrieval systems, such as search engines.

Key Concepts :

An **inverted index** is a data structure that maps each unique word to a list of documents in which that word appears. It helps quickly locate documents containing specific words, making it essential for search engines.

TF-IDF (Term Frequency-Inverse Document Frequency) measures how important a word is in a document compared to all other documents in a collection. It helps highlight words that are relevant to a specific document while filtering out common words.

Cosine Similarity: A metric used to measure the similarity between two vectors, in this case, between the query and documents in the dataset.

PART 1

Objective :

The objective of Part 1 was to create an inverted index consisting of two main tables:

Dictionary and Posting. This index allows efficient search by storing terms and their document frequencies (DF) and term frequencies (TF) within specific documents.

Steps :

1.Data Preprocessing in Python

We loaded the StateOfUnionAddresses.csv file.

The text data was cleaned and preprocessed by removing stop words and applying stemming.

Term frequencies (TF) and document frequencies (DF) were calculated for each term.

NAME_OF_PRESIDENT	DATE_OF_LINK_TO	TEXT_OF_ADDRESS
George Washington	1790-01-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
George Washington	1790-12-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
George Washington	1791-10-2	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
George Washington	1792-11-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
George Washington	1793-12-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
George Washington	1794-11-1	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
George Washington	1795-12-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
George Washington	1796-12-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
John Adams	1797-11-2	https://www.C:\Users\...Gentlemen of the Senate and Gentlemen of the House of Representatives:
John Adams	1798-12-0	https://www.C:\Users\...Gentlemen of the Senate and Gentlemen of the House of Representatives:
John Adams	1799-12-0	https://www.C:\Users\...Gentlemen of the Senate and Gentlemen of the House of Representatives:
John Adams	1800-11-1	https://www.C:\Users\...Gentlemen of the Senate and Gentlemen of the House of Representatives:
Thomas Jefferson	1801-12-0	https://www.C:\Users\...Fellow Citizens of the Senate and House of Representatives:
Thomas Jefferson	1802-12-1	https://www.C:\Users\...To the Senate and House of Representatives:
Thomas Jefferson	1803-10-1	https://www.C:\Users\...To the Senate and House of Representatives of the United States:
Thomas Jefferson	1804-11-0	https://www.C:\Users\...The Senate and House of Representatives of the United States:
Thomas Jefferson	1805-12-0	https://www.C:\Users\...At a moment when the nations of Europe are in commotion and arming against each other, and when those with whom we have principal intercourse are
Thomas Jefferson	1806-12-0	https://www.C:\Users\...The Senate and House of Representatives of the United States:
Thomas Jefferson	1807-10-2	https://www.C:\Users\...The Senate and House of Representatives of the United States:
Thomas Jefferson	1808-11-0	https://www.C:\Users\...The Senate and House of Representatives of the United States:
James Madison	1809-11-2	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
James Madison	1810-12-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
James Madison	1811-11-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
James Madison	1812-11-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
James Madison	1813-12-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
James Madison	1814-09-2	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:
James Madison	1815-12-0	https://www.C:\Users\...Fellow-Citizens of the Senate and House of Representatives:

Code:

Imported the required libraries and Data Preprocessing.

```
import pandas as pd
import re
from collections import defaultdict
from nltk.corpus import stopwords
from nltk.stem import PorterStemmer
import nltk

# Download stopwords if not already installed
nltk.download('stopwords')

# Load the CSV file into a DataFrame
file_path = 'D:\AdityaFiles\FALL24\Bigdata\LAB4\StateOfUnionAddresses.csv' #
Update to the correct path
df = pd.read_csv(file_path)

# Replace NaN values in the TEXT_OF_ADDRESS column with an empty string
df['TEXT_OF_ADDRESS'] = df['TEXT_OF_ADDRESS'].fillna("")

# Initialize text processing tools
stop_words = set(stopwords.words("english"))
stemmer = PorterStemmer()

# Function to preprocess text
def preprocess(text):
    # Lowercase, remove special characters, and split into words
    text = text.lower()
    text = re.sub(r'\W+', ' ', text)
    words = text.split()
    # Remove stop words and apply stemming
    words = [stemmer.stem(word) for word in words if word not in stop_words]
    return words
```

```

# Step 4: Create Dictionaries for Term Frequency and Document Frequency
term_doc_frequency = defaultdict(int) # Document frequency (DF) per term
term_collection_frequency = defaultdict(int) # Collection frequency per term
posting_list = defaultdict(lambda: defaultdict(int)) # Term frequency (TF)
per document

# Process each document
for _, row in df.iterrows():
    doc_id = f"{row['NAME_OF_PRESIDENT']}_{row['DATE_OF_UNION_ADDRESS']}"
    text = row['TEXT_OF_ADDRESS']
    terms = preprocess(text)
    unique_terms = set(terms)

    # Update term frequencies and document frequencies
    for term in unique_terms:
        term_doc_frequency[term] += 1 # Increment DF for the term
    for term in terms:
        term_collection_frequency[term] += 1 # Increment collection frequency
        posting_list[term][doc_id] += 1 # Increment TF for the term in this
document

# Function to create batched insert statements
def batch_insert_statements(table_name, columns, values_list,
batch_size=10): # Adjust batch_size as needed
    insert_statements = []
    for i in range(0, len(values_list), batch_size):
        batch_values = values_list[i:i+batch_size]
        statement = f"INSERT INTO {table_name} ({', '.join(columns)})
VALUES\n" + ",\n".join(batch_values) + ";"
        insert_statements.append(statement)
    return insert_statements

# Generate SQL inserts for Dictionary Table in batches (using only a limited
number of entries)
dictionary_values = []
for term, doc_freq in list(term_doc_frequency.items())[:10]: # Limit to the
first 10 entries
    collection_freq = term_collection_frequency[term]
    dictionary_values.append(f"('{term}', {doc_freq}, {collection_freq})")

dictionary_inserts = batch_insert_statements(
    "Dictionary", ["Term_PK", "DocFreq", "CollectionFreq"], dictionary_values
)

# Write the batched dictionary inserts to a file
with open('dictionary_insert_batched.sql', 'w') as f:
    f.write("\n\n".join(dictionary_inserts))

```

```

# Generate SQL inserts for Posting Table in batches (using only a limited
number of entries)
posting_values = []
for term, docs in list(posting_list.items())[:10]: # Limit to the first 10
entries
    for doc_id, term_freq in docs.items():
        posting_values.append(f"('{term}', '{doc_id}', {term_freq})")

posting_inserts = batch_insert_statements(
    "Posting", ["Term_FK", "Doc_Id", "TermFreq"], posting_values
)

# Write the batched posting inserts to a file
with open('posting_insert_batched.sql', 'w') as f:
    f.write("\n\n".join(posting_inserts))

# Display a sample of the SQL inserts for verification
print(dictionary_inserts[:1]) # Display the first batched insert for the
dictionary table
print(posting_inserts[:1])    # Display the first batched insert for the
posting table

# ----- Save the Full Data as CSVs -----

# Convert Dictionary table data to DataFrame and save as CSV (all entries)
dictionary_data = {
    'Term_PK': list(term_doc_frequency.keys()),
    'DocFreq': list(term_doc_frequency.values()),
    'CollectionFreq': [term_collection_frequency[term] for term in
term_doc_frequency.keys()]
}
dictionary_df = pd.DataFrame(dictionary_data)
dictionary_df.to_csv("Dictionary.csv", index=False)
print("Dictionary.csv has been created successfully with all entries.")

# Convert Posting table data to DataFrame and save as CSV (all entries)
posting_data = {
    'Term_FK': [],
    'Doc_Id': [],
    'TermFreq': []
}

for term, docs in posting_list.items():
    for doc_id, term_freq in docs.items():
        posting_data['Term_FK'].append(term)
        posting_data['Doc_Id'].append(doc_id)
        posting_data['TermFreq'].append(term_freq)

```

```
posting_df = pd.DataFrame(posting_data)
posting_df.to_csv("Posting.csv", index=False)
print("Posting.csv has been created successfully with all entries.")
```

2. Generating SQL Inserts in Python

The processed data was stored in two lists of values: one for Dictionary and one for Posting.

Python scripts generated batched SQL INSERT statements to populate the Dictionary and Posting tables.

The screenshot shows the SQL Server Enterprise Manager interface with the following SQL script:

```
USE [index]; -- Use brackets if the database is named 'index'
GO

-- Step 2: Create Dictionary Table
CREATE TABLE Dictionary (
    Term_PK NVARCHAR(100) PRIMARY KEY, -- Reduced length to fit within key limits
    DocFreq INT, -- Number of documents containing the term
    CollectionFreq INT -- Total occurrences of the term across all documents
);
GO

-- Step 3: Create Posting Table
CREATE TABLE Posting (
    Term_FK NVARCHAR(100), -- Reduced length to fit within key limits
    Doc_Id NVARCHAR(100), -- Reduced length to fit within key limits
    TermFreq INT, -- Frequency of the term in this specific document
    PRIMARY KEY (Term_FK, Doc_Id), -- Composite key with reduced length
    FOREIGN KEY (Term_FK) REFERENCES Dictionary(Term_PK) -- Foreign key constraint
);
GO

-- Step 4: Insert Data into Dictionary Table in Batches
INSERT INTO Dictionary (Term_PK, DocFreq, CollectionFreq) VALUES
('term1', 5, 100),
('term2', 10, 150),
('term1000', 7, 50);
GO
```

The Results pane shows the data inserted into the Dictionary table:

Term_PK	DocFreq	CollectionFreq
term1	5	100
term1000	7	50
term1001	8	75
term1002	6	60
term2	10	150

The screenshot shows the SQL Server Enterprise Manager interface with the following SQL script:

```
INSERT INTO Dictionary (Term_PK, DocFreq, CollectionFreq) VALUES
('term2', 10, 150),
('term1000', 7, 50);
GO

INSERT INTO Dictionary (Term_PK, DocFreq, CollectionFreq) VALUES
('term1001', 8, 75),
('term1002', 6, 60);
GO

-- Step 5: Insert Data into Posting Table in Batches
INSERT INTO Posting (Term_FK, Doc_Id, TermFreq) VALUES
('term1', 'doc1', 3),
('term1', 'doc2', 1),
('term1000', 'doc5', 2);
GO

INSERT INTO Posting (Term_FK, Doc_Id, TermFreq) VALUES
('term1001', 'doc6', 4),
('term1002', 'doc7', 2);
GO

-- Step 6: Verification Queries
SELECT TOP 10 * FROM Dictionary;
SELECT TOP 10 * FROM Posting;
GO
```

The Results pane shows the data inserted into the Posting table:

Term_FK	Doc_Id	TermFreq
term1	doc1	3
term1	doc2	1
term1000	doc5	2
term1001	doc6	4
term1002	doc7	2

3. Creating SQL Tables for Dictionary and Posting :

Dictionary Table will have Unique Term , Document Frequency (DF) and Collection Frequency

Posting Table will have term frequency (TF) of each term in each document.

4. Populating the Tables

```
posting_insert_batched.sql
1  INSERT INTO Posting (Term_FK, Doc_Id, TermFreq) VALUES
2  ('fellow', 'George Washington_1790-01-08', 2),
3  ('fellow', 'George Washington_1790-12-08', 3),
4  ('fellow', 'George Washington_1791-10-25', 1),
5  ('fellow', 'George Washington_1792-11-06', 1),
6  ('fellow', 'George Washington_1793-12-03', 2),
7  ('fellow', 'George Washington_1794-11-19', 3),
8  ('fellow', 'George Washington_1795-12-08', 2),
9  ('fellow', 'George Washington_1796-12-07', 1),
10 ('fellow', 'John Adams_1797-11-22', 1),
11 ('fellow', 'John Adams_1798-12-08', 1);
12
13 INSERT INTO Posting (Term_FK, Doc_Id, TermFreq) VALUES
14 ('fellow', 'John Adams_1799-12-03', 1),
15 ('fellow', 'Thomas Jefferson_1801-12-08', 3),
16 ('fellow', 'Thomas Jefferson_1802-12-15', 4),
17 ('fellow', 'Thomas Jefferson_1803-10-17', 2),
18 ('fellow', 'Thomas Jefferson_1804-11-08', 2),
19 ('fellow', 'Thomas Jefferson_1805-12-03', 2),
20 ('fellow', 'Thomas Jefferson_1806-12-02', 4),
21 ('fellow', 'Thomas Jefferson_1807-10-27', 1),
22 ('fellow', 'Thomas Jefferson_1808-11-08', 4),
23 ('fellow', 'James Madison_1809-11-29', 2);
24
25 INSERT INTO Posting (Term_FK, Doc_Id, TermFreq) VALUES
26 ('fellow', 'James Madison_1810-12-05', 1),
27 ('fellow', 'James Madison_1811-11-05', 1),
28 ('fellow', 'James Madison_1812-11-04', 3),
29 ('fellow', 'James Madison_1813-12-07', 1),
30 ('fellow', 'James Madison_1814-09-20', 1),
31 ('fellow', 'James Madison_1815-12-05', 1),
32 ('fellow', 'James Madison_1816-12-03', 2),
33 ('fellow', 'James Monroe_1817-12-12', 2),
```

```
dictionary_insert_batched.sql
1  INSERT INTO Dictionary (Term_PK, DocFreq, CollectionFreq) VALUES
2  ('inform', 168, 618),
3  ('sens', 141, 316),
4  ('avoid', 128, 292),
5  ('knowledg', 114, 222),
6  ('circumst', 134, 429),
7  ('relianc', 45, 59),
8  ('free', 187, 1208),
9  ('task', 103, 296),
10 ('saw', 34, 48),
11 ('govern', 215, 7961);
```

Bonus Part :

The bonus part included calculating the term frequencies and document frequencies more efficiently using batch inserts and verifying the data by displaying the top entries in both tables.

```
# Function to create batched insert statements
def batch_insert_statements(table_name, columns, values_list,
batch_size=10): # Adjust batch_size as needed
    insert_statements = []
    for i in range(0, len(values_list), batch_size):
        batch_values = values_list[i:i+batch_size]
        statement = f"INSERT INTO {table_name} ({', '.join(columns)})
VALUES\n" + ",\n".join(batch_values) + ";"
        insert_statements.append(statement)
    return insert_statements

# Generate SQL inserts for Dictionary Table in batches (using only a limited
number of entries)
dictionary_values = []
for term, doc_freq in list(term_doc_frequency.items())[:10]: # Limit to the
first 10 entries
    collection_freq = term_collection_frequency[term]
    dictionary_values.append(f"('{term}', {doc_freq}, {collection_freq})")

dictionary_inserts = batch_insert_statements(
    "Dictionary", ["Term_PK", "DocFreq", "CollectionFreq"], dictionary_values
)

# Write the batched dictionary inserts to a file
with open('dictionary_insert_batched.sql', 'w') as f:
    f.write("\n\n".join(dictionary_inserts))
```

outputs:

```
SELECT TOP (1000) [Term_PK]
, [DocFreq]
, [CollectionFreq]
FROM [StateOfUnionIndex].[dbo].[Dictionary]
```

Term_PK	DocFreq	CollectionFreq
1	1	1
2	1	1
3	1	1
4	1	1
5	5	21
6	10	167
7	159	2832
8	1	1
9	1	1
10	4	4
11	6	6
12	4	4
13	2	3
14	5	5
15	1	1
16	6	7
17	3	3
18	6	8
19	9	11
20	8	9
21	5	7

```
SELECT TOP (1000) [Term_FK]
, [Doc_Id]
, [TermFreq]
FROM [StateOfUnionIndex].[dbo].[Posting]
```

Term_FK	Doc_Id	TermFreq
1	Abraham Lincoln_1862-12-01	1
2	Harry S. Truman_1946-01-21	1
3	Harry S. Truman_1946-01-21	1
4	Harry S. Truman_1946-01-21	1
5	Benjamin Harrison_1891-12-09	4
6	Grover Cleveland_1894-12-03	2
7	Harry S. Truman_1946-01-21	6
8	Herbert Hoover_1932-12-06	8
9	John Quincy Adams_1826-12-05	1
10	Benjamin Harrison_1890-12-01	1
11	Chester A. Arthur_1881-12-06	12
12	Chester A. Arthur_1882-12-04	16
13	Chester A. Arthur_1883-12-04	21
14	Chester A. Arthur_1884-12-01	1
15	Rutherford B. Hayes_1880-12-...	1
16	William H. Taft_1910-12-06	56
17	William J. Clinton_1998-01-27	2
18	William J. Clinton_1999-01-19	56
19	William McKinley_1899-12-05	1
20	Abraham Lincoln_1861-12-03	15
21	Abraham Lincoln_1862-12-01	19

PART -2 (Calculating TF-IDF and Cosine Similarity for Query Terms)

Objective: The objective of Part 2 was to use the inverted index to calculate the TF-IDF scores for each term in each document and calculate cosine similarity for a sample query.

1. Calculating Total Document Count :

Importing the required libraries and calculating the total document count

Code

```
import pandas as pd
import math
from collections import defaultdict
```

```

# Load the CSV files
dictionary_file_path = r'D:\AdityaFiles\FALL24\Bigdata\LAB4\Dictionary.csv'
posting_file_path = r'D:\AdityaFiles\FALL24\Bigdata\LAB4\Posting.csv'

# Load the Dictionary and Posting tables into DataFrames
dictionary_df = pd.read_csv(dictionary_file_path)
posting_df = pd.read_csv(posting_file_path)

# Step 1: Drop NaN values in the Term_PK column in dictionary_df
dictionary_df = dictionary_df.dropna(subset=['Term_PK'])

# Step 2: Calculate the total number of documents
total_docs = posting_df['Doc_Id'].nunique()

# Step 3: Create a list to store document vectors for each term-document pair
document_vectors = []

# Step 4: Calculate TF-IDF for each term in each document
for term in dictionary_df['Term_PK']:
    # Check if the term exists in the dictionary_df to avoid IndexError
    if term in dictionary_df['Term_PK'].values:
        # Fetch Document Frequency (DF) from the dictionary
        doc_freq = dictionary_df.loc[dictionary_df['Term_PK'] == term,
'DocFreq'].values[0]
        idf = math.log(total_docs / doc_freq) # Calculate Inverse Document
Frequency (IDF)

        # Get all occurrences of this term in different documents from the
posting data
        term_postings = posting_df[posting_df['Term_FK'] == term]
        for _, row in term_postings.iterrows():
            doc_id = row['Doc_Id']
            term_freq = row['TermFreq']
            tf_idf = term_freq * idf # Calculate TF-IDF

            # Add the calculated TF-IDF to the document_vectors list
            document_vectors.append({
                'Term': term,
                'Document': doc_id,
                'TF_IDF': tf_idf
            })
        else:
            print(f"Warning: Term '{term}' not found in Dictionary.")

# Step 5: Convert the document vectors list to a DataFrame
document_vectors_df = pd.DataFrame(document_vectors)

# Save the result to a CSV file

```

```

output_file_path = r'D:\AdityaFiles\FALL24\Bigdata\LAB4\DocumentVectors.csv'
document_vectors_df.to_csv(output_file_path, index=False)
print(f"Document vectors saved to {output_file_path}")

# Optional: Generate SQL INSERT statements for DocumentVectors table (for SQL
database import)
insert_statements = []
for _, row in document_vectors_df.iterrows():
    term = row['Term']
    document = row['Document']
    tf_idf = row['TF_IDF']
    insert_statements.append(f"INSERT INTO DocumentVectors (Term, Document,
TF_IDF) VALUES ('{term}', '{document}', {tf_idf});")

# Write SQL statements to a file
with open('document_vectors_insert.sql', 'w') as f:
    f.write("\n".join(insert_statements))

print("SQL insert statements saved to 'document_vectors_insert.sql'")

```

```

-- Step 1: Calculate the total number of documents
DECLARE @total_docs INT;
SELECT @total_docs = COUNT(DISTINCT Doc_Id) FROM Posting;

```

Creating and Populating DocumentVectors Table:

```

IF OBJECT_ID('DocumentVectors', 'U') IS NOT NULL
    DROP TABLE DocumentVectors;
GO

-- Step 1.2: Create DocumentVectors table to store TF-IDF scores
CREATE TABLE DocumentVectors (
    Term NVARCHAR(255),
    Document NVARCHAR(255),
    TF_IDF FLOAT
);
GO

-- Step 1.3: Insert TF-IDF scores into DocumentVectors table
INSERT INTO DocumentVectors (Term, Document, TF_IDF)
SELECT
    Term_FK AS Term,
    Doc_Id AS Document,
    TermFreq * LOG(@total_docs * 1.0 / (SELECT DocFreq FROM Dictionary WHERE Term_PK = Term_FK)) AS TF_IDF
FROM Posting;
GO

```

Calculating TF_IDF Scores:

```
-- Step 1.2: Insert TF-IDF scores into DocumentVectors table
INSERT INTO DocumentVectors (Term, Document, TF_IDF)
SELECT
    Term_FK AS Term,
    Doc_Id AS Document,
    TermFreq * LOG(@total_docs * 1.0 / (SELECT DocFreq FROM Dictionary WHERE Term_PK = Term_FK)) AS TF_IDF
FROM Posting;
SELECT * FROM DocumentVectors;
```

	Term	Document	TF_IDF
1	000	Abraham Lincoln_1861-12-03	4.66489726979423
2	000	Abraham Lincoln_1862-12-01	5.90886987507269
3	000	Abraham Lincoln_1863-12-08	1.55496575659808
4	000	Abraham Lincoln_1864-12-06	2.17695205923731
5	000	Andrew Jackson_1829-12-08	1.24397260527846
6	000	Andrew Jackson_1830-12-06	1.24397260527846
7	000	Andrew Jackson_1831-12-06	2.79893836187654
8	000	Andrew Jackson_1832-12-04	4.66489726979423
9	000	Andrew Jackson_1833-12-03	2.17695205923731
10	000	Andrew Jackson_1834-12-01	4.66489726979423
11	000	Andrew Jackson_1835-12-07	6.84184932903153
12	000	Andrew Jackson_1836-12-05	5.59787672375307
13	000	Andrew Johnson_1865-12-04	2.79893836187654
14	000	Andrew Johnson_1866-12-03	3.10993151319615
15	000	Andrew Johnson_1867-12-03	15.8606507173004
16	000	Andrew Johnson_1868-12-09	22.3915068950123
17	000	Benjamin Harrison_1889-12	7.46383563167076

3. Creating QueryVector Table for Query Terms

```
WHILE @@FETCH_STATUS = 0
BEGIN
    -- Get Document Frequency (DF) for the term
    SELECT @doc_freq = DocFreq FROM Dictionary WHERE Term_PK = @term;

    -- Calculate TF-IDF for the query term
    SET @query_tf_idf = LOG(@total_docs * 1.0 / @doc_freq);

    -- Insert into the temporary QueryVector table
    INSERT INTO #QueryVector (Term, TF_IDF) VALUES (@term, @query_tf_idf);

    FETCH NEXT FROM cur INTO @term;
END
CLOSE cur;
DEALLOCATE cur;

-- View the contents of the #QueryVector table for verification
SELECT * FROM #QueryVector;
```

	Term	TF_IDF
1	freedom	0.330041346290462
2	peace	0.330041346290462

Calculating Cosine Similarity and Retrieving Top Documents Based on Similarity

```
-- Step 4: Calculate Cosine Similarity between query and documents
-- Drop and create CosineSimilarityResults table if it exists
IF OBJECT_ID('CosineSimilarityResults', 'U') IS NOT NULL
    DROP TABLE CosineSimilarityResults;

CREATE TABLE CosineSimilarityResults (
    Document NVARCHAR(255),
    CosineSimilarity FLOAT
);

-- Insert cosine similarity calculations
INSERT INTO CosineSimilarityResults (Document, CosineSimilarity)
SELECT
    DocumentVectors.Document,
    SUM(DocumentVectors.TF_IDF * #QueryVector.TF_IDF) /
    (MAX(#DocumentMagnitudes.Magnitude) * @query_magnitude) AS CosineSimilarity
FROM DocumentVectors
JOIN #QueryVector ON DocumentVectors.Term = #QueryVector.Term
JOIN #DocumentMagnitudes ON DocumentVectors.Document = #DocumentMagnitudes.Document
GROUP BY DocumentVectors.Document;
```

100 %

Results Messages

	Document	CosineSimilarity
1	Dwight D. Eisenhower_1959-01-09	0.0547017308755134
2	Ronald Reagan_1985-02-06	0.0520830537659014
3	Dwight D. Eisenhower_1960-01-07	0.046634007143188
4	Ronald Reagan_1988-01-25	0.0450032299114084
5	Harry S. Truman_1951-01-08	0.0427882597589155

Bonus Part

Storing and Querying Similar Documents Efficiently

The goal is to store the results in a way that allows easy querying, such as retrieving documents with the highest similarity scores or performing additional analysis on these scores

```

-- Step 1: Normalize the Cosine Similarity scores
-- Create a normalized version of the CosineSimilarityResults table to make querying easier
IF OBJECT_ID('NormalizedCosineSimilarityResults', 'U') IS NOT NULL
    DROP TABLE NormalizedCosineSimilarityResults;
GO

CREATE TABLE NormalizedCosineSimilarityResults (
    Document NVARCHAR(255),
    CosineSimilarity FLOAT,
    NormalizedScore FLOAT
);
GO

-- Insert normalized scores into the table
-- NormalizedScore = CosineSimilarity / (sum of all cosine similarities for the query terms)
INSERT INTO NormalizedCosineSimilarityResults (Document, CosineSimilarity, NormalizedScore)
SELECT

```

Document	NormalizedScore
Dwight D. Eisenhower_1959-01-09	0.0340041683326866
Ronald Reagan_1985-02-06	0.0323763233665585
Dwight D. Eisenhower_1960-01-07	0.0289890393511207
Ronald Reagan_1988-01-25	0.0279753013465822
Harry S. Truman_1951-01-08	0.0265984122296976

OUTPUT :

Part 1 (Building the Inverted Index)

Dictionary table

Term_PK	DocFreq	CollectionFreq
1	1	1
2	1	1
3	1	1
4	1	1
5 0	5	21
6 00	10	167
7 000	159	2832
8 0000	1	1
9 0001	1	1
10 001	4	4
11 002	6	6
12 003	4	4
13 004	2	3
14 005	5	5
15 006	1	1
16 007	6	7
17 008	3	3
18 009	6	8
19 01	9	11
20 010	8	9
21 011	5	7
22 0111	1	1
23 012	4	6
24 013	4	6

Posting Table :

Term_FK	Doc_Id	TermFreq
1	Abraham Lincoln_1862-12-01	1
2	Harry S. Truman_1946-01-21	1
3	Harry S. Truman_1946-01-21	1
4	Harry S. Truman_1946-01-21	1
5	Benjamin Harrison_1891-12-09	4
6	Grover Cleveland_1894-12-03	2
7	Harry S. Truman_1946-01-21	6
8	Herbert Hoover_1932-12-06	8
9	John Quincy Adams_1826-12-05	1
10	Benjamin Harrison_1890-12-01	1
11	Chester A. Arthur_1881-12-06	12
12	Chester A. Arthur_1882-12-04	16
13	Chester A. Arthur_1883-12-04	21
14	Chester A. Arthur_1884-12-01	1
15	Rutherford B. Hayes_1880-12-...	1
16	William H. Taft_1910-12-06	56
17	William J. Clinton_1998-01-27	2
18	William J. Clinton_1999-01-19	56
19	William McKinley_1899-12-05	1
20	Abraham Lincoln_1861-12-03	15
21	Abraham Lincoln_1862-12-01	19
22	Abraham Lincoln_1863-12-08	5
23	Abraham Lincoln_1864-12-06	7
24	Andrew Jackson_1829-12-08	4

Bonus part output :

SQL insert Statements :

Name	Date modified	Type	Size
pycache	04-11-2024 10:43	File folder	
Dictionary	08-11-2024 22:23	Microsoft Excel Co...	179 KB
dictionary_insert_batched	08-11-2024 22:23	Microsoft SQL Ser...	1 KB
inverted_index	07-11-2024 22:21	Data Base File	16 KB
invertedindex	08-11-2024 16:05	Python File	5 KB
Posting	08-11-2024 22:23	Microsoft Excel Co...	10,037 KB
posting_insert_batched	08-11-2024 22:23	Microsoft SQL Ser...	89 KB
querycheck	08-11-2024 16:22	Python File	3 KB
StateOfUnionAddresses	05-11-2024 21:29	Microsoft Excel Co...	9,932 KB
DocumentVectors	08-11-2024 16:28	Microsoft Excel Co...	14,492 KB
document_vectors_insert	08-11-2024 16:28	Microsoft SQL Ser...	33,005 KB

Part 2 : Calculating TF-IDF and Cosine Similarity for Query Terms

DocumentVectors Table

SELECT TOP (1000) [Term]

100 %

Results Messages

	Term	Document	TF_IDF
1		Harry S. Truman_1946-01-21	5.37989735354046
2	00	Benjamin Harrison_1890-12-01	3.07731226054641
3	00	Chester A. Arthur_1881-12-06	36.927747126557
4	00	Chester A. Arthur_1882-12-04	49.2369961687426
5	00	Chester A. Arthur_1883-12-04	64.6235574714747
6	00	Chester A. Arthur_1884-12-01	3.07731226054641
7	00	Rutherford B. Hayes_1880-12-06	3.07731226054641
8	00	William H. Taft_1910-12-06	172.329486590599
9	00	William J. Clinton_1998-01-27	6.15462452109283
10	00	William J. Clinton_1999-01-19	172.329486590599
11	00	William McKinley_1899-12-05	3.07731226054641
12	005	Andrew Jackson_1829-12-08	3.77045944110636
13	005	Millard Fillmore_1851-12-02	3.77045944110636
14	005	Theodore Roosevelt_1906-12-03	3.77045944110636
15	005	William H. Taft_1910-12-06	3.77045944110636
16	005	William J. Clinton_1999-01-19	3.77045944110636
17	01	Benjamin Harrison_1889-12-03	3.18267277620424
18	01	Chester A. Arthur_1883-12-04	3.18267277620424
19	01	Grover Cleveland_1886-12-06	3.18267277620424
20	01	Grover Cleveland_1888-12-03	3.18267277620424
21	01	Grover Cleveland_1894-12-03	3.18267277620424
22	01	James Buchanan_1859-12-19	3.18267277620424
23	01	Ulysses S. Grant_1870-12-05	3.18267277620424
24	01	William H. Taft_1910-12-06	6.36534555240847

CosineSimilarityResults Table:

SELECT TOP (1000) [Document]

100 %

Results Messages

	Document	CosineSimilarity
1	Abraham Lincoln_1862-12-01	0.00564155430228733
2	Abraham Lincoln_1863-12-08	0.00529534948955656
3	Andrew Jackson_1829-12-08	0.00232040183054018
4	Andrew Jackson_1830-12-06	0.00179052266025876
5	Andrew Jackson_1832-12-04	0.00307416028732756
6	Andrew Jackson_1834-12-01	0.0033308425424282
7	Andrew Jackson_1835-12-07	0.00581310893463317
8	Andrew Johnson_1865-12-04	0.0160125642558192
9	Andrew Johnson_1867-12-03	0.0031975943941226
10	Andrew Johnson_1868-12-09	0.00165150284401277
11	Benjamin Harrison_1890-12-01	0.00158970831199178
12	Benjamin Harrison_1892-12-06	0.000895933484098...
13	Calvin Coolidge_1923-12-06	0.00581012165611936
14	Calvin Coolidge_1924-12-03	0.00808016442956607
15	Calvin Coolidge_1925-12-08	0.00589286788711004
16	Calvin Coolidge_1926-12-07	0.00612165727282854
17	Calvin Coolidge_1927-12-06	0.00216358990902244
18	Calvin Coolidge_1928-12-04	0.00214720008941829
19	Chester A. Arthur_1883-12-04	0.00208111198115522
20	Chester A. Arthur_1884-12-01	0.00185594639372129
21	Dwight D. Eisenhower_1953-...	0.04200496514313
22	Dwight D. Eisenhower_1954-...	0.0363524593968167
23	Dwight D. Eisenhower_1955-...	0.0251351374224384
24	Dwight D. Eisenhower_1956-...	0.0119672646599821

Bonus Part Output :(Normalized Scores and Ranking:

NormalizedCosineSimilarityResults Table:

100 %

Results Messages

	Document	Cosine Similarity	Normalized Score
1	Abraham Lincoln_1862-12-01	0.00564155430228733	0.00350695232276177
2	Abraham Lincoln_1863-12-08	0.00529534948955656	0.00329174145938938
3	Andrew Jackson_1829-12-08	0.00232040183054018	0.00144242847862941
4	Andrew Jackson_1830-12-06	0.00179052266025876	0.00111304035481962
5	Andrew Jackson_1832-12-04	0.00307416028732756	0.00191098640242004
6	Andrew Jackson_1834-12-01	0.0033308425424282	0.00207054747061218
7	Andrew Jackson_1835-12-07	0.00581310893463317	0.00361359561362611
8	Andrew Johnson_1865-12-04	0.0160125642558192	0.00995387022820103
9	Andrew Johnson_1867-12-03	0.0031975943941226	0.0019877165913606
10	Andrew Johnson_1868-12-09	0.00165150284401277	0.00102662164086766
11	Benjamin Harrison_1890-12-01	0.00158970831199178	0.000988208383457892
12	Benjamin Harrison_1892-12-06	0.000895933484098...	0.000556938007638202
13	Calvin Coolidge_1923-12-06	0.00581012165611936	0.00361173863543155
14	Calvin Coolidge_1924-12-03	0.00808016442956607	0.00502286247658288
15	Calvin Coolidge_1925-12-08	0.00589286788711004	0.00366317607118483
16	Calvin Coolidge_1926-12-07	0.00612165727282854	0.0038053981299787
17	Calvin Coolidge_1927-12-06	0.00216358990902244	0.00134494968060023
18	Calvin Coolidge_1928-12-04	0.00214720008941829	0.00133476129760316
19	Chester A. Arthur_1883-12-04	0.00208111198115522	0.00129367902978096
20	Chester A. Arthur_1884-12-01	0.00185594639372129	0.00115370962816813
21	Dwight D. Eisenhower_1953-...	0.04200496514313	0.0261114937804465
22	Dwight D. Eisenhower_1954-...	0.0363524593968167	0.0225977337252751
23	Dwight D. Eisenhower_1955-...	0.0251351374224384	0.0156247239401414
24	Dwight D. Eisenhower_1956-...	0.0119672646599821	0.0074391957158709

Ranked Documents Table:

SELECT TOP (100) [Document]

100 %

Results Messages

	Document	Rank	Normalized Score
1	Dwight D. Eisenhower_1959-01-09	1	0.0340041683326866
2	Ronald Reagan_1985-02-06	2	0.0323763233665585
3	Dwight D. Eisenhower_1960-01-07	3	0.0289890393511207
4	Ronald Reagan_1988-01-25	4	0.0279753013465822
5	Harry S. Truman_1951-01-08	5	0.0265984122296976
6	George H.W. Bush_1991-01-29	6	0.0262943568304006
7	Franklin D. Roosevelt_1941-01-06	7	0.0262237222355923
8	Dwight D. Eisenhower_1953-02-02	8	0.0261114937804465
9	Harry S. Truman_1950-01-04	9	0.0248555837482398
10	George W. Bush_2005-02-02	10	0.0234220363920743
11	Lyndon B. Johnson_1965-01-04	11	0.022899997322102
12	Ronald Reagan_1987-01-27	12	0.0228669016737035
13	Dwight D. Eisenhower_1954-01-07	13	0.0225977337252751
14	Dwight D. Eisenhower_1957-01-10	14	0.0183363911858151
15	Ronald Reagan_1984-01-25	15	0.0182771603081576
16	John F. Kennedy_1963-01-14	16	0.0180194475030645
17	George W. Bush_2002-01-29	17	0.0165842858945362
18	Harry S. Truman_1952-01-09	18	0.0158179516368999
19	Ronald Reagan_1986-02-04	19	0.0156798740102595
20	Franklin D. Roosevelt_1942-01-06	20	0.0156282348198707
21	Dwight D. Eisenhower_1955-01-06	21	0.0156247239401414
22	Harry S. Truman_1948-01-07	22	0.0152354236303258
23	Franklin D. Roosevelt_1944-01-11	23	0.0137786052764322
24	Ronald Reagan_1983-01-25	24	0.0137351304785621