f2822217

August 9, 2023

1 Athens University of Economics and Business

2 School of Business

3 Department of Management Science & Technology

4 Master of Science in Business Analytics

Program:

Full-Time

Quarter:

3rd (Spring Quarter)

Course:

Advanced Topics in Data Engineering

Instructor:

Dr. Giorgos Alexiou

Assignment:

Entity Resolution Assignment

Student (Registration No):

Souflas Eleftherios-Efthymios (f2822217)

4.1 Task A

A **Token Blocking** methodology was implemented in a schema-agnostic manner to create blocks represented as Key-Value (K-V) pairs from the provided `ER-Data.csv` file. The primary goal of this approach was to create distinct Blocking Keys (BK) derived from entities' attribute values, excluding the id column, to facilitate accurate matching and comprehensive analysis.

The required libraries (`pandas` and `html`) were imported and from some other libraries (`nltk`, `pprint`, and `contextlib`) some of their functions.

[1]:
```
import pandas as pd
import nltk
from nltk.corpus import stopwords
from pprint import pprint
import html
from contextlib import redirect_stdout
nltk.download('stopwords')
```

```
[nltk_data] Downloading package stopwords to
[nltk_data]     [redacted]nltk_data…
[nltk_data]   Package stopwords is already up-to-date!
```

[1]:
```
True
```

Two functions are defined: `token_blocking` and `decode_html_special_chars`.

1. `token_blocking(dataframe)`: This function takes a DataFrame as input and returns an inverted index for the data. The inverted index is implemented as a Python dictionary, where the keys are tokens (words) extracted from the DataFrame attributes, and the values are sets of corresponding IDs (unique identifiers) of the DataFrame's entities associated with each token. The function processes the DataFrame row by row, extracting the ID and all other attribute values from each row. For each attribute value, the function converts the text to lowercase, tokenizes it into individual words, and filters out English stopwords using the NLTK library. It then populates the inverted index by associating each token with the set of IDs of entities containing that token. Finally, the function removes tokens from the index that do not meet the minimum threshold of having at least two entities associated with them. The resulting inverted index, excluding the tokens with insufficient entities, is returned as a dictionary.

2. `decode_html_special_chars(text)`: This function takes a text input and decodes any HTML special characters present in the text using the `html.unescape` function from the `html` module. It specifically targets instances of encoded characters like '&', '<', '>', etc., converting them back to their original form. The function is designed to handle both strings and other data types. If the input is a string, it decodes the HTML special characters and returns the updated string. For other data types, it simply returns the original input.

Tokens that belong to the english list of stopwords, like I, you, of, and, etc., are filtered out. If this step was omitted, the token with the most entities as its set of values would be the 'of' token (word) having 30,326 entities associated to it. After the stopwords filtering step, 'j' is the token with the

most values in its block, having 7,599 entities, reducing this way the computation of unnecessary comparisons.

Overall, these functions work together to create an inverted index for a given DataFrame while also providing a utility function to decode any HTML special characters present in the data. The inverted index is a useful data structure for efficient record linkage and entity resolution tasks, allowing for fast lookups of entities associated with specific attributes or tokens.

[2]:
```
# Define the function token_blocking that takes a DataFrame as input and␣
 ↪returns an inverted index
def token_blocking(dataframe):
    # Create an empty dictionary to store the blocking index
    index = {}
    # A list of English stopwords from nltk.corpus
    en_stop = stopwords.words('english')
    # Iterate over DataFrame's rows as Pandas named tuples without returning␣
 ↪the index as the first element of the tuple
    for row in dataframe.itertuples(index=False):
        # Extract the ID and all other attribute values from the row tuple
        id_, *attributes = row
        # Iterate over each attribute value
        for attr in attributes:
            # Check if the attribute value is indeed available (not null)
            if pd.notna(attr):
                # Transform the attribute value to lowercase and split into␣
 ↪individual words
                tokens = str(attr).lower().split()
                # For each token in the attribute's value add the ID to the␣
 ↪corresponding blocking key's value set
                for token in tokens:
                    # Filter out stopwords
                    if token not in en_stop:
                        # If the token is not in the index, create a new entry␣
 ↪with the ID as the value set
                        if token not in index:
                            index[token] = {id_}
                        # If the token already exists in the index, add the ID␣
 ↪to its value set
                        else:
                            index[token].add(id_)
    # Create a list to store keys to delete
    keys_to_delete = []
    # Iterate over each value (set of IDs) in the blocking index
    for key, value in index.items():
        # Each block should contain at least two entities, otherwise add the␣
 ↪key to the keys_to_delete list
        if len(value) < 2:
```

```
            keys_to_delete.append(key)
    # Delete the keys along with their values from the dictionary
    for key in keys_to_delete:
        del index[key]
    # Return the blocking index as a dictionary
    return index


# Function to decode HTML special characters in a cell
def decode_html_special_chars(text):
    return html.unescape(text) if isinstance(text, str) else text
```

The `ER-Data.csv` file was loaded into a pandas DataFrame. The file must be placed in the 'Data' folder. The `pandas` library is utilized to read the CSV file into a DataFrame named 'df', using the semicolon (;) separator to parse the CSV file.

[3]:
```
print("Loading data...")
data_path = "../Data/ER-Data.csv"
df = pd.read_csv(data_path, sep=";")
print("Data loaded successfully!\n")
```

```
Loading data…
Data loaded successfully!
```

Then, the task of removing HTML special characters from the DataFrame 'df' is being executed. This includes characters such as `&`, `&hellip;`, `<`, and `&quot;`. The aim is to decode the HTML entities and replace them with their corresponding characters to ensure proper representation of the textual data.

[4]:
```
# Remove HTML special characters, like `&amp;`, `&hellip;`, `&lt;`, `&quot;`
print("Removing HTML special characters...")
# Apply the function to the entire DataFrame
df = df.applymap(decode_html_special_chars)
print("Completed successfully!\n")
```

```
Removing HTML special characters…
Completed successfully!
```

The task of token blocking on the DataFrame `df` is being executed. Token blocking is a schema-agnostic approach used to create blocks in the form of Key-Value (K-V) pairs. Each distinct Blocking Key (BK) is derived from the entities' attribute values, excluding the 'ID' column, which is used only as a reference for the blocking index creation and not included in the blocking process. The `token_blocking(df)` function takes the DataFrame `df` as input and returns an inverted index (blocking index) containing Blocking Keys as keys and sets of corresponding entity 'id's as values.

[5]:
```
# Perform Token Blocking
print("Performing Token Blocking...")
block_index = token_blocking(df)
```

```
print("Completed successfully!\n")
```

```
Performing Token Blocking…
Completed successfully!
```

Finally, the resulting Blocking Index is presented in a neatly formatted manner to the output. The purpose of displaying the index is to provide a clear and organized view of the relationships formed through token blocking. The format of the displayed output is as follows:

```
{'Blocking Key': {Entities}}
----------------------------
{Block_1: {Entity_1, Entity_2, ...},
 Block_2: {Entity_3, Entity_4, ...},
 ...
 Block_N: {Entity_M, Entity_N, ...}}
```

Each 'Blocking Key' is shown as a dictionary key, and the corresponding 'Entities' associated with that key are shown as a set of IDs enclosed in curly braces. The displayed output uses proper indentation and alignment for improved readability. The pprint function is used to format the output, ensuring that the representation remains concise and well-structured even for large blocking indices. Below a sample of 10 mid-sized blocks of the blocking index is printed.

[6]:
```
# Create a list to store keys to print as sample
keys_to_print = []
# Iterate over each value (set of IDs) in the blocking index
for token_key, ID_value in block_index.items():
    # Each block should contain at least two entities, otherwise add the key to␣
 ↪the keys_to_delete list
    if 10 <= len(ID_value) <= 100:
        keys_to_print.append(token_key)
    if len(keys_to_print) == 10:
        break
# Create a new dictionary containing only the desired keys and values
filtered_dict = {key: value for key, value in block_index.items() if key in␣
 ↪keys_to_print}
# Display the sample index
print("Sample print of 10 mid-size blocks")
print("{'Blocking Key': {Entities}}")
print("----------------------------")
pprint(filtered_dict, width=70, compact=True)
```

```
Sample print of 10 mid-size blocks
{'Blocking Key': {Entities}}
----------------------------
{'advising': {5, 7215, 17067, 17455, 22627, 42038, 46748, 52148,
              52401, 54359, 63637},
 'facilities': {4, 769, 1651, 2425, 4509, 5481, 8833, 9626, 11207,
                14222, 15146, 18392, 29246, 29260, 33357, 34649,
                34828, 35711, 35972, 37287, 38109, 40835, 41463,
```

```
              43646, 46920, 50229, 57607, 59493, 61007, 62757,
              62788, 65983},
'hansen,': {3, 3970, 10387, 13434, 15901, 19173, 20234, 23716, 24927,
            38317, 39985, 41534, 42422, 43897, 49003, 53711, 56426,
            57778, 59313, 62493},
'inc': {1, 923, 2857, 3057, 3486, 4378, 5589, 6339, 7038, 8574, 9368,
        10500, 11596, 15004, 17005, 18337, 21912, 22216, 22275,
        23987, 24308, 26475, 27244, 29327, 30987, 32596, 36256,
        36411, 38590, 39000, 40028, 41393, 42111, 42685, 43073,
        43918, 44647, 44908, 45497, 46763, 47124, 49406, 50987,
        51988, 52505, 53149, 56545, 56805, 57323, 58857, 59169,
        60213, 61288, 61397, 62978, 65238},
'infectious': {4, 1127, 22840, 26202, 38006, 40890, 45665, 48973,
               54199, 64194},
'initiation': {2, 2653, 8485, 15660, 15801, 17496, 21279, 22138,
               25349, 31892, 34902, 37086, 37763, 42392, 42454,
               49508},
'labelling': {3, 12452, 18804, 18928, 20926, 36095, 36814, 42629,
              43414, 51275, 57566},
'road': {1, 588, 835, 1865, 2075, 2414, 7096, 9041, 9470, 10515,
         14530, 14759, 16568, 17834, 18431, 19955, 19961, 25250,
         27089, 28273, 28548, 29054, 29806, 31596, 31828, 32742,
         32876, 36420, 36911, 38165, 38707, 41916, 42167, 42329,
         42867, 43779, 44030, 44123, 45497, 45715, 45920, 48309,
         50056, 50419, 52812, 53149, 54551, 55470, 55873, 58270,
         58680, 62054},
'valley': {1, 7798, 8932, 16285, 21712, 27852, 28464, 28783, 30722,
           32175, 33057, 33085, 33155, 42019, 45850, 46269, 47380,
           57933, 59594},
'wm': {6, 1068, 1519, 1648, 2663, 3089, 3380, 3767, 4507, 5293, 5733,
       6981, 7257, 7420, 8447, 9024, 9893, 10497, 11678, 12668,
       12717, 13252, 13368, 14718, 15951, 16891, 16975, 18268, 19871,
       20224, 20265, 21626, 22312, 22777, 24366, 25126, 25452, 25705,
       25994, 28292, 30260, 30477, 31417, 31632, 33218, 33856, 33990,
       36260, 36275, 36871, 36995, 39378, 40622, 40908, 41309, 41789,
       41951, 42049, 42096, 42741, 44810, 45082, 46466, 47826, 49342,
       49556, 49629, 50084, 50544, 51046, 52188, 52204, 53325, 53581,
       53800, 54774, 57315, 57981, 58080, 58915, 59847, 59873, 60511,
       62104, 62290, 62402, 62842}}
```

The resulting blocking index, obtained after performing Token Blocking on a DataFrame, is also written to a file named `blocking_index_print.txt`. The purpose of writing to this file is to store the entire output of the blocking index, which may be quite extensive, and allow users to view it conveniently at a later time, if needed. The resulting output is directed to the file `blocking_index_print.txt` using the `redirect_stdout` context manager to capture the standard output and store it in the file. Finally, the file is closed, and the process is completed, confirming that the data has been successfully written to the file.

[7]:
```
# Write the resulting index to a file
print("\nWriting to a file, named `blocking_index_print.txt`, to be able to see␣
 ↪the entire output if needed...")
f = open('blocking_index_print.txt', 'w', encoding="utf-8")
with redirect_stdout(f):
    pprint(block_index, width=120, compact=True)
f.close()
print("Completed!")
```

```
Writing to a file, named `blocking_index_print.txt`, to be able to see the
entire output if needed…
Completed!
```

4.2 Task B

Task B involves the **computation of all possible comparisons** required to resolve duplicates within the blocks created in Task A. To achieve this, the number of comparisons for each block in the `block_index` is calculated. A function named `calculate_comparisons` is defined, which takes a blocking index as input. The purpose of this function is to compute the total number of pairwise comparisons required for the entities within the given blocking data structure.

The function employs the combination formula (`comb` function) from the `math` module to calculate the number of unique pairs that can be formed within each block. It iterates through the values (i.e., sets of IDs) within the blocking index to determine the number of distinct entities present in each block and by computing their lengths, it effectively counts the unique entities. For each block, the number of pairwise comparisons possible among its entities is determined using the combination formula. The formula calculates the number of ways to choose two entities from the block without repetition. These pairwise combinations represent the possible comparisons that can be made within the block.

The function accumulates the number of pairwise comparisons within each block and aggregates them to obtain the total number of comparisons required for the entire blocking data structure. This total count represents the final result of the function.

[8]:
```
# Use the combination formula from the math module
from math import comb


# Define the function calculate_comparisons that takes a blocking index as input
def calculate_comparisons(blocking_index):
    # Initialize a variable to keep track of the total number of comparisons
    total_comparisons = 0
    # Iterate over each value (set of IDs) in the blocking index
    for value in blocking_index.values():
        # Calculate the number of unique entities in the block by getting the␣
 ↪set's length
```

```
        num_entities = len(value)
        # Calculate the number of pairwise comparisons that can be made within␣
↪the block
        # using the combination formula from the math module (math.comb)
        comparisons_in_block = comb(num_entities, 2)
        # Add the number of comparisons within this block to the total number␣
↪of comparisons
        total_comparisons += comparisons_in_block
    # Return the total number of comparisons across all blocks in the blocking␣
↪index
    return total_comparisons
```

After the `calculate_comparisons` function is executed with the given blocking index `block_index`, the total number of pairwise comparisons is obtained and printed (392,309,099 comparisons).

[9]:
```
# Calculate the number of comparisons for the given blocking index (block_index)
print("Calculating all pair-wise comparisons...")
num_comparisons = calculate_comparisons(block_index)
# Print the result, which represents the total number of pair-wise comparisons
print("Number of comparisons:", num_comparisons)
```

```
Calculating all pair-wise comparisons…
Number of comparisons: 392309099
```

4.3 Task C

In this Task, a custom (self-created) **Meta-Blocking Graph** was created from the Block Collection created in the first Task. The Graph was stored in a `SQLite` database (disk) instead of creating it in memory using libraries, like `NetworkX` or `iGraph`, because when executing the latter (with `NetworkX`), after 30 minutes of run-time, a memory error was being produced as the entire graph could not fit in memory.

The Meta-Blocking procedure contained the following phases:

1. Graph Building: It took 10 minutes to complete the insertion of all nodes (entities) and undirected edges (pairs of co-occurring entities) to the SQLite ("graph") database for every block of the Block Colection created in Task A.
2. Edge Weighting: It took approximately 14 minutes to apply the attribute-agnostic CBS (Common Blocks Scheme) Weighting Scheme in order to put weights on the previously created edges, based on the number of common blocks (edges) that entities per comparison have in common.
3. Graph Pruning: An Edge-centric pruning algorithm was implemented by deleting all edges having weight < 2.
4. Block Collecting: The pruned blocking graph was transformed into a new block collection. Every retained edge created a block of minimum size. The final number of comparisons of the new block collection (after edge pruning) was calculated and printed (53,626,150 comparisons).

Firstly, a class, named `SQLiteGraph`, is defined, that facilitates interactions with an SQLite database to represent a graph. The class contains different methods for facilitating all phases of the Meta-Blocking procedure:

1. The class includes a constructor (`__init__`) that takes a parameter representing the path to an SQLite database file. This establishes a connection to the specified database.

2. The method `add_nodes_and_edges_from_token_blocking_dict` is responsible for populating the database with nodes and edges. It receives a blocking index as input and iterates over the index's values (which represent sets of entity IDs). For each set of entity IDs, the method calculates pairs of unique combinations within the set using `itertools.combinations` and inserts these pairs as edges into the 'edgelist' table of the database. Additionally, the unique entity IDs are inserted into the 'nodelist' table. Throughout this process, a progress bar is used to track the insertion progress. The method updates the progress bar as pairs of edges are inserted and nodes are added to the database. Once all data is inserted, the method commits the changes to the database and closes the progress bar.

3. The method `query_graph(self, query)` facilitates running SELECT queries on the SQLite database. It accepts a SQL query as input, establishes a connection to the database, executes the query, fetches all the rows from the query result, and returns them as a list.

4. The method `is_num_edges_equal_to(self, number_comparisons)` checks whether the number of edges (comparisons) in the graph (retrieved from the 'edgelist' table) is equal to a specified number (`number_comparisons`). It does so, by executing a SELECT query to count the number of rows in the 'edgelist' table and comparing it with the provided number. The method returns a boolean indicating the result of the comparison.

5. The method `apply_cbs_weighting_scheme(self)` applies the Common Blocks Scheme (CBS) weighting to the edges in the graph. It establishes a connection to the database, creates a new table named 'edgelist_weighted' by aggregating and calculating the weights of the edges from the original 'edgelist' table. The weights are computed based on the count of similar edges (common blocks). The original 'edgelist' table is then dropped, and the new 'edgelist_weighted' table is renamed to 'edgelist' to replace it. The method provides progress tracking using a progress bar.

6. The method `prune_edges_with_weight_lower_than(self, limit)` prunes edges in the graph whose weight is lower than a specified limit. It establishes a connection to the database, defines a query to delete edges with weight below the given limit, and executes the query. The changes are then committed to the database.

7. The method `number_of_edges(self)` retrieves the total number of edges in the graph. It defines a SELECT query to count the number of rows (edges) in the 'edgelist' table, executes the query using the `query_graph` method, and returns the count of edges extracted from the query result.

8. The method `number_of_nodes(self)` retrieves the total number of nodes (entities) in the graph. It defines a SELECT query to count the number of rows (nodes) in the 'nodelist' table, executes the query using the `query_graph` method, and returns the count of nodes extracted from the query result.

9. The method `close_connection(self)` closes the connection established with the SQLite database and associated with the instance of the `SQLiteGraph` class.

This class serves as a tool to efficiently manage the process of constructing a graph in an SQLite database by adding nodes and edges based on a provided blocking index. It encapsulates the necessary database interactions and progress tracking, enabling querying the graph, comparing the number of edges, applying a CBS weighting scheme to the edges within the SQLite database, allowing the retrieval of edge and node information from the SQLite database and providing a mechanism to close the database connection.

[10]:
```
import sqlite3
from itertools import combinations
from tqdm.auto import tqdm


class SQLiteGraph:

    # Constructor for the SQLiteGraph class
    def __init__(self, database_file):
        # Initialize a connection to an SQLite database using the provided file␣
↪path
        # The database_file parameter represents the path to the SQLite␣
↪database file
        # This connection will be used to interact with the SQLite database
        self.conn = sqlite3.connect(database_file)

    # Function to add nodes and edges to the SQLite database from a blocking␣
↪index
    def add_nodes_and_edges_from_token_blocking_dict(self, blocking_index):
        # Establish a connection to the SQLite database
        conn = self.conn
        c = conn.cursor()
        # Drop the 'nodelist' and 'edgelist' tables if they exist
        c.execute('''DROP TABLE IF EXISTS nodelist''')
        c.execute('''DROP TABLE IF EXISTS edgelist''')
        # Create the 'nodelist' table if it doesn't exist
        c.execute('''CREATE TABLE IF NOT EXISTS nodelist (id INTEGER)''')
        # Create the 'edgelist' table if it doesn't exist
        c.execute('''CREATE TABLE IF NOT EXISTS edgelist
                     (source INTEGER, target INTEGER)''')
        # Begin a transaction
        c.execute('BEGIN TRANSACTION')
        # Progress bar to track the insertion progress
        pbar = tqdm(total=num_comparisons, desc='Progress')
        # Initialize an empty set to store unique nodes (entities)
        node_set = set()
        # Iterate over each value (list of IDs) in the blocking index
        for value in blocking_index.values():
            # Get the union (distinct IDs) of the sets node_set and the IDs␣
↪contained in each block
```

```
            # and store it in the node_set
            node_set.union(value)
            # Use itertools.combinations to generate unique pairs within the␣
↪current block
            block_pairs = set(combinations(value, 2))
            # Insert the pairs into the 'edgelist' table in the database
            c.executemany('INSERT INTO edgelist (source, target) VALUES (?, ?
↪)', block_pairs)
            # Update the progress bar to reflect the number of inserted pairs
            pbar.update(len(block_pairs))
            # Clear the block_pairs set for memory efficiency
            block_pairs.clear()
        # Insert the unique nodes into the 'nodelist' table in the database
        c.executemany('INSERT INTO nodelist (id) VALUES (?)', node_set)
        # Commit the changes to the database
        conn.commit()
        # Set the progress bar description to 'Committing...' before closing it
        pbar.set_description('Committing...')
        # Close the progress bar
        pbar.set_description('Completed')
        pbar.close()

    # Function to run SELECT query on the SQLite database
    def query_graph(self, query):
        # Establish a connection to the SQLite database
        conn = self.conn
        c = conn.cursor()
        # Execute the SELECT query
        c.execute(query)
        # Fetch all rows from the query result
        rows = c.fetchall()
        # Return the query result
        return rows

    # Function to check if the number of edges in the graph (comparisons) is␣
↪the same as a specified number
    def is_num_edges_equal_to(self, number_comparisons):
        # Define the SELECT query to count the number of rows (edges) in the␣
↪'edgelist' table
        query = 'SELECT COUNT(*) FROM edgelist'
        # Call the function to run the SELECT query on the graph database and␣
↪retrieve the result
        result = self.query_graph(query)
        # Return a boolean indicating whether the number of edges in the␣
↪database is equal to 'number_comparisons'
        return result[0][0] == number_comparisons
```

```
    # Function to apply Common Blocks Scheme (CBS) weights to the edges
    def apply_cbs_weighting_scheme(self):
        # Establish a connection to the SQLite database
        conn = self.conn
        c = conn.cursor()
        # Progress bar initialization and update
        pbar = tqdm(total=4, desc='Progress', mininterval=0.01)
        pbar.update(1)
        # Define the SELECT query to create a new table 'edgelist_weighted' with weighted edges
        c.execute('CREATE TABLE edgelist_weighted AS \
        SELECT source, target, COUNT(*) as weight FROM edgelist GROUP BY source, target')
        pbar.update(1)
        # Drop the existing 'edgelist' table to replace it with the weighted version
        c.execute('DROP TABLE edgelist')
        pbar.update(1)
        # Rename the 'edgelist_weighted' table to 'edgelist'
        c.execute('ALTER TABLE edgelist_weighted RENAME TO edgelist')
        pbar.update(1)
        # Commit the changes to the database
        pbar.set_description('Committing...')
        conn.commit()
        # Close the progress bar
        pbar.set_description('Completed')
        pbar.close()

    # Function to prune edges if their weight is lower than a specified limit
    def prune_edges_with_weight_lower_than(self, limit):
        # Establish a connection to the SQLite database
        conn = self.conn
        c = conn.cursor()
        # Define query to delete edges with weight lower than the specified 'limit'
        c.execute('DELETE FROM edgelist WHERE weight < ?', (limit,))
        # Commit changes to the database
        conn.commit()

    # Function to retrieve the graph's number of edges
    def number_of_edges(self):
        # Define SELECT query to count the number of rows in the 'edgelist' table
        query = 'SELECT COUNT(*) FROM edgelist'
        # Call the function to run the SELECT query on the database and retrieve the result
```

```
        result = self.query_graph(query)
        # Return the number of edges, which is extracted from the query result
        return result[0][0]

    # Function to retrieve the graph's number of nodes
    def number_of_nodes(self):
        # Define SELECT query to count the number of rows in the 'nodelist'␣
↪table
        query = 'SELECT COUNT(*) FROM nodelist'
        # Call the function to run the SELECT query on the database and␣
↪retrieve the result
        result = self.query_graph(query)
        # Return the number of nodes, which is extracted from the query result
        return result[0][0]

    # Method to close the connection established with the SQLite database
    def close_connection(self):
        # Close the connection to the SQLite database
        self.conn.close()
```

Then, the environment to create and work with an SQLite-based graph is set by initializing an instance of the `SQLiteGraph` class, defining the path to the SQLite Graph Database file as 'my_graph_database.db' in the same directory as the code.

[11]:
```
# Define the SQLite Graph Database file path
db_file = 'my_graph_database.db'
print("Creating the Meta-Blocking Graph...\n")
# Create the graph by initializing an instance of the SQLiteGraph class
graph = SQLiteGraph(db_file)
```

```
Creating the Meta-Blocking Graph…
```

Then, the Graph Building Phase begins, which involves creating the graph structure by adding nodes and edges based on the information obtained from the token blocking operation. This phase runs for approximately 10-20 minutes depending on computer's performance.

[12]:
```
# Phase 1: Graph Building
print("1. Graph Building Phase: Adding all nodes (entities) and edges␣
↪(comparisons) to the graph...")
# Add nodes and edges to the graph using the blocking index generated from␣
↪token blocking
graph.add_nodes_and_edges_from_token_blocking_dict(block_index)
```

```
1. Graph Building Phase: Adding all nodes (entities) and edges (comparisons) to
the graph…
```

```
Completed: 100% 392309099/392309099 [09:57<00:00, 777950.02it/s]
```

Then, in order to check that the previous phase completed successfully, the number of edges (com-

parisons) in the graph is compared with the previously calculated `num_comparisons` and the result of the comparison is printed as a boolean value, indicating whether the number of graph edges is equal to the number of comparisons in Task B. If the value is True, then the previous phase succeeded inserting all edges (comparisons) to the graph.

[13]:
```
# Check if the number of graph edges is the same as the number of comparisons␣
  ↪in Task B
print('\nCheck: Is the number of graph edges the same as the number of␣
  ↪comparisons of Task B?:')
print(graph.is_num_edges_equal_to(num_comparisons))
```

```
Check: Is the number of graph edges the same as the number of comparisons of
Task B?:
True
```

Then, the commencement of the Edge Weighting Phase is indicated, which invokes the `apply_cbs_weighting_scheme()` method on the `graph` instance to apply the Common Blocks Scheme (CBS) Weighting Scheme to the graph's edges. This phase runs for approximately 15-30 minutes depending on computer's performance.

[14]:
```
# Phase 2: Edge Weighting
print("\n2. Edge Weighting Phase: Applying CBS Weighting Scheme...")
# Apply the CBS Weighting Scheme to the graph
graph.apply_cbs_weighting_scheme()
```

```
2. Edge Weighting Phase: Applying CBS Weighting Scheme…
```

Completed: 100% 4/4 [13:50<00:00, 161.86s/it]

Then, the Graph Pruning Phase starts, which invokes the `prune_edges_with_weight_lower_than(threshold)` method on the `graph` instance to remove edges from the graph that have a weight lower than the defined `threshold`, which in our case is 2.

[15]:
```
# Phase 3: Graph Pruning
# Set a threshold for pruning edges with weight below this value
threshold = 2
print("\n3. Graph Pruning Phase: Pruning edges with weight <", threshold, "...")
# Remove edges from the graph that have weight less than the specified threshold
graph.prune_edges_with_weight_lower_than(threshold)
print("Completed!")
```

```
3. Graph Pruning Phase: Pruning edges with weight < 2 …
Completed!
```

Then, in order to check that the previous phase completed successfully, the number of edges (comparisons) in the graph is again compared with the previously calculated `num_comparisons`. The

value must be False, if the previous phase succeeded pruning all edges (comparisons) of the graph that have weight < 2.

[16]:
```
# Check if the number of graph edges (after pruning) is the same as the initial␣
↪number of comparisons (Task B)
print('Check: Is the number of graph edges (after pruning) the same as the␣
↪initial number of comparisons (Task B)?:')
print(graph.is_num_edges_equal_to(num_comparisons))
```

```
Check: Is the number of graph edges (after pruning) the same as the initial
number of comparisons (Task B)?:
False
```

Lastly, the start of the Block Collecting Phase is indicated, which invokes the `number_of_edges()` method on the `graph` instance to count the number of edges remaining in the graph after the pruning process. The final number of comparisons (after edge pruning) is 53,626,150.

[17]:
```
# Phase 4: Block Collecting
print("\n4. Block Collecting Phase: Count every retained edge...")
# Count the number of edges after pruning to get the final number of comparisons
print('Final number of comparisons (after edge pruning):', graph.
↪number_of_edges())
```

```
4. Block Collecting Phase: Count every retained edge…
Final number of comparisons (after edge pruning): 53626150
```

Finally, the `close_connection()` method on the `graph` instance is invoked to terminate the connection to the graph's SQLite database.

[18]:
```
# Close the connection to the graph
print("\nClosing connection with the graph...\n")
graph.close_connection()
print("Closed!")
```

```
Closing connection with the graph…

Closed!
```

4.4 Task D

In Task D, a function named `jaccard_similarity` is defined that calculates the **Jaccard similarity** between two entities based on their titles in a given DataFrame. The function begins by accepting three inputs: a DataFrame (`dataframe`), and two entity IDs (`id1` and `id2`). This function employs the following steps:

1. Title Extraction and Preprocessing: The function extracts the titles of the entities associated with the provided IDs from the DataFrame. It converts the titles to lowercase and tokenizes

them by splitting them into sets of words.

2. Intersection and Union Calculation: The function calculates the intersection of the two sets (common words) and the union of the sets (total unique words). These values are essential for computing the Jaccard similarity.

3. Jaccard Similarity Computation: The Jaccard similarity is computed by dividing the intersection count by the union count. This ratio quantifies the similarity between the sets of words, which correspond to the titles of the two entities.

4. Result Return: The calculated Jaccard similarity value is returned as the output of the function, representing the extent of similarity between the titles of the two entities ranging from 0 (no similar word) to 1 (identical).

Overall, the function enables the assessment of title-based similarity between entities by systematically preprocessing titles, computing set-based measures, and generating a Jaccard similarity value as the result.

[19]:
```
# Function that calculates the Jaccard similarity between two entities in a␣
 ↪DataFrame based on their titles,
# given their IDs
def jaccard_similarity(dataframe, id1, id2):
    # Extract the titles of entities with ID1 and ID2 from the DataFrame
    title1 = dataframe.loc[dataframe['id'] == id1, 'title']
    title2 = dataframe.loc[dataframe['id'] == id2, 'title']
    # Convert the titles to lowercase and split them into sets of words
    set1 = set(str(title1.values[0]).lower().split())
    set2 = set(str(title2.values[0]).lower().split())
    # Calculate the intersection of the two sets (common words)
    intersection = len(set1.intersection(set2))
    # Calculate the union of the two sets (total unique words)
    union = len(set1.union(set2))
    # Calculate the Jaccard similarity by dividing the intersection by the union
    jaccard_sim = intersection / union
    # Return the Jaccard similarity value
    return jaccard_sim
```

The `jaccard_similarity` function is then tested by calculating the Jaccard similarity of entities with IDs 10 and 810. The result of the computation is displayed using the `print` statement. This action demonstrates the usage of the function to determine the Jaccard similarity between the titles of the entities associated with the given IDs (10 and 810) in the provided DataFrame (`df`).

[20]:
```
# Test the function by calculating the Jaccard similarity of entities with ID␣
 ↪10 and 810
print("The Jaccard similarity of entities with ID 10 and 810 is:")
print(jaccard_similarity(df, 10, 810))
print("\n--------------------------")
print("END")
print("BYE-BYE")
```

```
The Jaccard similarity of entities with ID 10 and 810 is:
0.2

--------------------------
END
BYE-BYE
```