**Task 2, Part B**In this section, we aim to compare the performance of different methods for matching records between two datasets, **ACM** and **DBLP2**, using techniques such as **Shingling**, **MinHashing**, and **Locality-Sensitive Hashing (LSH)**. Our objective is to evaluate both **precision** and **execution time** for this method and compare it with methods used in the previous part of the assignment.

We will utilized three datasets:

* **ACM.csv**
* **DBLP2.csv**
* **DBLP-ACM\_PerfectMapping.csv**

These datasets are available through this [website](https://dbs.uni-leipzig.de/research/projects/benchmark-datasets-for-entity-resolution). The **PerfectMapping** file contains the true mapping between ACM and DBLP2 records, which will be used to evaluate the precision of our method.

**Dataset Import and Initial Preprocessing**

1. We began by importing the datasets **df\_ACM** and **df\_DBLP2**. During the import, we noticed an encoding issue in the **df\_DBLP2** dataset. This issue was resolved without causing any loss of data integrity.
2. Next, we processed both datasets by concatenating their rows into single strings. Before concatenation, we ensured that missing values were removed, and we joined the remaining values using spaces as delimiters.
3. To ensure consistent processing, we converted all alphabetical characters in both datasets to lowercase. This normalization step eliminates any discrepancies due to case differences between records.
4. In this step, we replaced multiple consecutive spaces with a single space. This ensured that the data was clean and uniformly formatted before further processing.
5. After cleaning and normalizing both datasets, we needed to combine them into a comprehensive list for further analysis. Using the zip function, we paired each record’s ID with its corresponding concatenated string from both datasets. This ensured that the original IDs from df\_ACM and df\_DBLP2 were preserved during the comparison process.

**Data Preparation for Comparison**

1. **Shingling**  
   We implemented **k-shingling** (with k=5) to break down each text into overlapping groups of five characters. This method helps capture the essence of each record, even in cases where there are slight variations in the text.
2. **Building a Vocabulary**  
   A vocabulary was created, assigning a unique number to each unique group of five characters (shingle) found across all records.
3. **Binary Vector Representation**  
   For each record, we generated a binary vector that indicated the presence or absence of each shingle in the vocabulary. If a specific group of characters (shingle) was present in a record, its corresponding position in the vector was marked as 1, otherwise as 0.

**MinHashing and Locality-Sensitive Hashing (LSH)**

1. **MinHashing with 500 Hash Functions**  
   We used **500 hash functions** to generate MinHash signatures for each record. The MinHash technique enabled us to condense the binary vectors into compact representations while preserving the similarity between records. This method greatly reduces computational time while ensuring accuracy in comparing large datasets.
2. **Locality-Sensitive Hashing (LSH)**  
   We applied **LSH** to identify likely matching records between **ACM** and **DBLP2**. LSH works by grouping similar MinHash signatures into the same bucket, allowing us to focus on the most promising candidate pairs for comparison. From this step, we extracted the top **2224 candidates**.

**Precision and Matching Evaluation**

1. **Loading the Perfect Mapping**  
   To evaluate the accuracy of our method, we imported the **PerfectMapping** dataset, which contains the true mappings between **ACM** and **DBLP2** records.
2. **Candidate Filtering and Adjustments**  
   We then pinpointed the boundary between the **ACM** and **DBLP2** datasets by identifying where the ACM dataset ends. This boundary allowed us to filter the candidate pairs, ensuring that one record came from **ACM** and the other from **DBLP2**. After filtering, we retained the top **2224 candidates** for comparison.
3. **Precision Calculation**  
   The extracted candidates were compared against the true mappings from the **PerfectMapping** dataset. We calculated the precision as the ratio of correctly identified pairs to the total number of extracted candidates. Our method achieved a precision of **98.88%**, with **443 correctly identified pairs**.

Interestingly, we found that increasing the number of hash functions can impact both precision and the number of correctly identified pairs. For example, using **750 hash functions** increased precision but reduced the number of correctly identified pairs to **44**.

**Execution Time and Comparison**

1. **Measuring Execution Time**  
   Using the **time** package, we measured the execution time of the code. With the current configuration (500 hash functions and 2224 candidates), the total running time was **14.73 seconds**.
2. **Comparison with Previous Methods**  
   In comparison with earlier methods used in Part 1, which took **X minutes** and yielded a precision of **X**, the current method demonstrates significant improvements. The earlier method required a longer elapsed time and returned many more matches, but at a much lower precision. In contrast, the method used in this part offers a much higher precision of **96.05%** within a substantially shorter time frame.