UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA
Dipartimento di Ingegneria "Enzo Ferrari"

Master's Degree in
Computer Engineering
Curriculum: Data Engineering and Analytics

Table Overlap under Order and Contiguity Constraints

Candidate:
Davide Rosario Lupo

Supervisor:
Giovanni Simonini

Co-Supervisor:
Felix Naumann

ACADEMIC YEAR 2024/2025

Abstract

A vast number of tables can be found on the Web, and redundant data often appear as pairs of tables with substantial overlap. This overlap suggests important relationships between tables, such as duplicate or related tables. This thesis investigates the development of an efficient method to determine the largest overlap between two tables under order and contiguity constraints. These constraints limit our ability to freely rearrange rows and columns in order to determine the largest overlap. This makes the overlap problem a more accurate representation of real world scenarios. In fact, in many cases, a meaningful overlap can only be identified when the original order of rows and columns is preserved, ensuring that the structural and semantic relationships within the tables remain intact. To tackle this, two complementary approaches are proposed. The first exploits the classical Longest Common Subsequence and Longest Common Substring algorithms. The second constructs a mapping of all common elements between the two tables and, using pairs of indexes that respect both row and column constraints, identifies the largest common subtable. The methods are evaluated using two real world datasets, Wiki-tables and Git-tables, and demonstrated their efficacy in solving this task. We also observed that in some cases the first approach is more effective, while in other contexts the second approach yields better results. For this reason, we decided to combine the two approaches into a hybrid framework that dynamically selects the most suitable method based on the characteristics of the input tables. The final framework is designed to be flexible and dynamic, allowing users to specify any desired constraints on rows and columns, such as ordering or contiguity, or even to set no constraints at all. In this way, it can adapt to different requirements, ranging from fully unconstrained matching to stricter scenarios.

Contents

Chapter 1

Introduction

The World Wide Web consists of a huge number of unstructured documents, but it also contains structured data in the form of HTML tables. [1] Tables are practical and useful tools in many application scenarios and can be used effectively to collect and organize information from multiple sources. With the help of additional operations, such as sorting, filtering, and joins, this information can be turned into knowledge and can be used to support decision making tasks. Thanks to their convenience and utility, a large number of tables have been produced and made available on the Web in the last two decades. In support of this claim, we cite a study conducted by Cafarella et al. [2] in 2008, in which they were already able to extract 14.1 billion HTML tables from the Web. They also estimated that 154 million of these tables contained high quality relational data. A more recent study, conducted by Lehmberg et al. [3], presents a large public corpus of web tables, consisting of over 233 million tables, extracted from the July 2015 version of CommonCrawl.

More recently, a similar situation is emerging with the increasing popularity of Data Lakes. [4] A Data Lake is a massive collection of datasets that: (1) may be hosted in different storage systems; (2) may vary in their formats; (3) may not be accompanied by any useful metadata; and (4) may change autonomously over time. For data science, data lakes provide a convenient storage layer for experimental data, both the input and output of data analysis and learning tasks. Data Lakes often contain multiple versions of the same table, as new datasets and updated versions are continuously ingested. Furthermore, changes in data extraction processes can produce new variants of existing tables over time. As a result, identifying duplicate or highly similar tables across different versions becomes an important concern for managing and maintaining data lake integrity.

As a final point, we examine the topic of Wikipedia tables, which represent a common case involving a very large number of tables. In the study *The Secret Life of Wikipedia Tables* [5], tables are characterized as dynamic entities rather than static entities. They change shape, they move, they grow, they shrink, and their data change. The authors analyzed all 3.5 million tables from English Wikipedia, tracking their complete history of modifications, which in total has 53.8 million versions. These

statistics are based on Wikidump data from September 1, 2019. Thus, the set of tables on Wikipedia constitutes an example of a highly heterogeneous and semi-structured data lake. Each table can have multiple versions, each version representing an incremental edit of the previous. Since creating tables can be challenging for inexperienced users, it is plausible that they copy and adapt existing ones. As one might expect, large collections of tables often contain pairs that share a significant amount of cell content, these are referred to as overlapping tables.

Arbitrary reordering, while useful for identifying multiple versions of the same table in which the order of the columns or rows has changed, can hide important differences between tables. For example, we could be interested in finding two tables that have evolved over time but have preserved their original structure, so the order of the original columns or rows has remained the same while extra columns or rows may have been added. We can refer to this problem as identifying structurally consistent duplicates.

1.1 Problem Statement

Existing methods allow arbitrary reordering of rows and columns to determine the largest overlap, ignoring the structural and semantic information encoded in the original tables. In many real world scenarios, meaningful overlap can only be identified when the original order of rows and columns is preserved, ensuring that structural relationships within the data remain intact. Thus the problem addressed in this work is to identify the largest overlap between two tables under these conditions. The ability to detect overlapping tables under structural constraints has practical implications in data cleaning, deduplication, data integration, and knowledge discovery from large scale table collections.

1.2 Contributions

This thesis aims to provide a relevant contribution to the research, which can be articulated in the following points:

- A framework for determining the largest overlap between two tables under order and contiguity constraints with respect to rows and columns.
- Integration of an existing method from the literature that allows arbitrary reordering on rows and columns to identify the largest overlap.
- Investigation and discussion of two alternative strategies to solve the problem addressed by the thesis:
    1. using the Longest Common Subsequence and Longest Common Substring algorithms;
    2. using a mapping based strategy to find the common elements.

1.3 Thesis Structure

The document is structured as follows:

- Abstract
- Introduction
- Related Work
- Problem Analysis and Preliminaries
- LCS-Based Largest Overlap Detection
- Mapping-Based Largest Overlap Detection
- Comparison of the Two Approaches
- Experimental Evaluation
- Final Framework
- Conclusion and Future Works

Chapter 2

Related Work

To enable data enrichment and integration, it is essential to identify related tables within Data Lakes or on the Web. Numerous approaches have been introduced to address this need, focusing on the efficient discovery of related tables in various forms: unionable tables, joinable tables, and more generally, related tables. We will discuss this in section 2.1. While a lot of attention has focused on this topic, the task of detecting duplicate tables has remained overlooked in the existing literature, section 2.2. Finally, in section 2.3 we will discuss two very recent studies that addressed the problem of finding the largest overlap between two tables, which are very close to the task that this thesis aims to solve.

2.1 Related Table Discovery

Zhu et al. [6] propose a novel approach for discovering joinable tables in large-scale data lakes. Given a table and a specific join column, the task is to identify other tables that can be joined on the greatest number of distinct values. This challenge can be framed as a set overlap similarity search problem, where columns are treated as sets and matching values correspond to set intersections. In modern data lakes, however, set sizes can be extremely large (up to tens of millions of elements) and dictionaries may contain hundreds of millions of distinct values. To address this, the authors introduce JOSIE (Joining Search using Intersection Estimation), an algorithm designed to reduce the cost of set scans and inverted index lookups when retrieving the top-k matching sets.

Mate [7] is the only join-discovery system capable of identifying multi-column joins. It leverages a specialized hashing function, XASH, to generate a super-key for each table row, which is then used to efficiently verify whether a row may contain a given combination of values. The system requires the user to specify a set of columns as input; based on this, it retrieves all tables with column sets that could potentially join with the user-defined set. However, Mate is not well-suited for detecting the

largest overlap, since users typically do not know in advance which set of columns would yield the maximum overlap.

2.2 Duplicate Table Detection

Although much research has focused on unionable, joinable, and related tables, duplicate tables have received little attention in the literature. Koch et al. [8] address this gap by defining the problem and proposing the use of XASH for detecting duplicate tables in data lakes, designing a pipeline that works both for query tables and for large-scale deduplication within data lakes. Since data lakes are usually only lightly curated, they are particularly prone to data quality issues and inconsistencies, among which duplicate tables are especially common. The aim of duplicate table detection is to identify tables that contain the same data. However, comparing tables directly is computationally expensive, as otherwise identical tables may differ in row and column order. In this work, the authors explore the use of Xash, a hash function originally introduced for discovering multi column join candidates, in the context of duplicate table detection. Xash enables the generation of a super key, which functions similarly to a Bloom filter, allowing the rapid identification of specific cell values. Their results show that leveraging Xash can significantly accelerate the process of detecting duplicate tables.

2.3 Unconstrained Overlap Detection

In this section, we will discuss two very recent studies that focused on solving the problem of finding the largest overlap between two tables. The first, Sloth [9], presents the first formal definition of this problem and proposes an innovative algorithm designed for this task. The proposed algorithm first identifies attribute pairs between two tables that share cell values. Combining these pairs provides a complete overview of all possible non-empty overlaps between tables organized in a lattice structure. These combinations provide an upper bound on the candidate area. The algorithm exploits this limit to prioritize the most promising candidates and find the largest overlap as soon as possible, reducing the number of candidates for which the effective area must be calculated. As the problem is computationally expensive, a greedy variant based on beam search is also proposed, useful for critical cases where the exact algorithm takes too long.

Although Sloth [9] offers a way to compute the exact largest overlap between pairs of tables, it suffers from significant scalability limitations. To tackle the problem of estimating the largest overlap between two tables both effectively and efficiently, Armadillo [10] was proposed. This method leverages an intermediate graph-based representation along with graph neural networks to produce table embeddings that capture not only the content but also the structural relationships. Specifically, how cell values co-occur within rows and columns. With these vector representations,

comparison of tables becomes much more efficient, enabling quick estimation of their overlap ratio.

These two methods determine the largest overlap between two tables by allowing the reordering of rows and columns. However, there are scenarios in which it is important to preserve the original semantic order of rows and columns when determining the largest overlap between two tables. This thesis aims to contribute in this direction by proposing a solution for these specific cases, since this problem has not yet been properly explored in the literature.

Chapter 3

Problem Analysis and Preliminaries

This section aims to establish the theoretical foundations for determining the largest overlap between two tables. It presents a formal definition of what we mean by table overlap and describes how to compute its area. We also define the specific structural constraints that are central to our problem: an ordering constraint, which requires preserving the relative order of rows and columns, and a contiguity constraint, which further restricts the overlap to contiguous blocks of rows and columns. Finally, we explain how to leverage the Longest Common Subsequence and the Longest Common Substring algorithms to determine the largest overlap under these constraints. This represents the first method proposed to address the problem discussed in this thesis

3.1 Largest Overlap Definition

The definition of table overlap relies on the concept of attribute mapping, defined as follows.

Attribute Mapping. [9] Given two tables $R(X)$ and $S(Y)$, where X and Y denote their schemas (i.e., their attribute sets), an *attribute mapping* between $R(X)$ and $S(Y)$ is defined as a bijective function M that maps a subset of X to a subset of Y:

$$M : X_M \subseteq X \rightarrow Y_M \subseteq Y$$

Because the mapping is bijective, the attribute sets have the same size ($|X_M| = |Y_M|$). This ensures that no attribute in X is mapped to more than one attribute in Y, and likewise, no attribute in Y is mapped from more than one attribute in X. Each such mapping defines a table overlap. By using the intersection under bag semantics (which allows duplicates), we can formally define the notion of table overlap and its corresponding area as follows.

Table Overlap. [9] Given a mapping M, defined between tables $R(X)$ and $S(Y)$ considering the attribute subsets $X_M \subseteq X$ and $Y_M \subseteq Y$, the *table overlap* O_M is the bag intersection between the bags of the tuples obtained through the projection of $R(X)$ on X_M and $S(Y)$ on Y_M.

Overlap Area. [9] We define the overlap area A_M as the number of cells contained in the overlap O_M:

$$A_M = |X_M| \times |O_M|$$

where $|X_M|$ and $|O_M|$ represent the width and the height of the rectangle of overlapping cells between the two tables, respectively. We also refer to A_M as the area of mapping M.

Largest Overlap. [9] Let O be the set of overlaps determined by all possible mappings between $R(X)$and $S(Y)$. We define the set of the largest overlaps $O' \subseteq O$ as those overlaps that have the maximum area. We refer to the mappings of O' as the top mappings, denoted as M'. The number of top mappings (in most cases just one) is equal to the number of largest overlaps.

3.2 Constraints Definition

In this section, we formally define the two constraints introduced previously to ensure an unambiguous interpretation of them.

Definition. *According to Wikipedia [11], a* ***sequence*** *is an enumerated collection of elements in which repetitions are allowed and the order matters.*

Given a sequence of n distinct elements $S = [s_1, s_2, s_3, .., s_n]$ and let $C = [c_1, c_2, c_3, .., c_m]$ be a sequence of m distinct elements extracted from S:

- We say that C is **ordered** with respect to S if for each $0 < i < m - 1$, $pos_S(c_{i+1}) > pos_S(c_i)$, where $pos_S(x)$ is the position of the element x in S.
- We say that C is **contiguous** with respect to S if for each $0 < i < m - 1$, $pos_S(c_{i+1}) - pos_S(c_i) = 1$, where $pos_S(x)$ is the position of the element x in S.

Figure 3.1 provides a summary of the defined constraints and highlights the different possible cases. The three cases represent increasing levels of restriction on the way columns and rows can be arranged within a table. In the first case, addressed by SLOTH [9], no constraints are imposed on columns and rows. Elements in the largest common subtable between two input tables may appear in any order and do not need to be contiguous. This represents the most flexible scenario. The second case introduces the ordering constraint, imposed as well on both columns and rows: elements must respect a specific sequence, yet they may still be separated by other elements and therefore are not required to be adjacent. Finally, the third case enforces contiguity, which inherently implies ordering as well. Here, elements must

not only follow the correct sequence but also appear consecutively, making this the most restrictive setting.

Case	Constraints Applied	Description
Case 1 (SLOTH)	No constraints	No requirements for ordering or contiguity.
Case 2	Ordering only	Elements must follow a specific order, but they don't need to be contiguous.
Case 3	Contiguity (implies ordering)	Elements must be contiguous, which inherently requires them to be in order.

Figure 3.1

Example 1 — Ordered but not contiguous. Let the full sequence be:

$$S = \underset{1}{[a,} \ \underset{2}{b,} \ \underset{3}{c,} \ \underset{4}{d,} \ \underset{5}{e,} \ \underset{6}{f]}$$

Take:

$$C = \underset{1}{[b,} \ \underset{2}{d,} \ \underset{3}{f]}$$

Sequence C consists of three elements: b, d, and f. These elements are denoted as c_1, c_2, and c_3, indicating that they belong to the sequence C and specifying their positions within it, i.e. their indices.

Using the definition of ordering provided above, we can verify that the sequence C satisfies the ordering constraint with respect to the sequence S. Therefore, since the sequence C consists of three elements (m = 3), we must consider each i in the range from 1 *to* $m-1$, thus $i = 1$ and $i = 2$. If the ordering condition, also reported below, is satisfied for all i in this range, we can conclude that C satisfies the ordering constraint with respect to S.

[Ordering condition: $pos_S(c_{i+1}) > pos_S(c_i)$**]**

- $(i = 1) \ \rightarrow \ pos_S(c_2) > pos_S(c_1) = pos_S(d) > pos_S(b) = 4 > 2$
- $(i = 2) \ \rightarrow \ pos_S(c_3) > pos_S(c_2) = pos_S(f) > pos_S(d) = 6 > 4$

As we can see, the ordering condition is satisfied for all values of i within the range considered; therefore, C meets the first constraint.

Let us now proceed to verify if the second constraint is satisfied as well.

$$[\textbf{Contiguity condition: } pos_S(c_{i+1}) - pos_S(c_i) = 1]$$

- $(i = 1) \ \rightarrow \ pos_S(c_2) - pos_S(c_1) = pos_S(d) - pos_S(b) = 4 - 2 = 2 \neq 1$
- $(i = 2) \rightarrow \ pos_S(c_3) - pos_S(c_2) = pos_S(f) - pos_S(d) = 6 - 4 = 2 \neq 1$

Since the contiguity condition is not satisfied for both $i = 1$ and $i = 2$, we can conclude that C does not meet the contiguity constraint with respect to S, but only satisfies the ordering constraint

Example 2 — Ordered and contiguous. With the same sequence:

$$S = \underset{1}{[a,} \ \underset{2}{b,} \ \underset{3}{c,} \ \underset{4}{d,} \ \underset{5}{e,} \ \underset{6}{f]}$$

Take:

$$C = \underset{1}{[d,} \ \underset{2}{e,} \ \underset{3}{f]}$$

In this case as well, sequence C is composed of three elements: d, e and f. Therefore, we must consider each i in the range from 1 *to* $m - 1$, where m = 3. As in Example 1, let us verify whether C satisfies the two constraints. We first consider the ordering constraint.

- $(i = 1) \ \rightarrow \ pos_S(c_2) > pos_S(c_1) = pos_S(e) > pos_S(d) = 5 > 4$
- $(i = 2) \ \rightarrow \ pos_S(c_3) > pos_S(c_2) = pos_S(f) > pos_S(e) = 6 > 5$

The condition is satisfied in both cases for values of i, therefore C is ordered with respect to S.

- $(i = 1) \ \rightarrow \ pos_S(c_2) - pos_S(c_1) = pos_S(e) - pos_S(d) = 5 - 4 = 1$
- $(i = 2) \ \rightarrow \ pos_S(c_2) - pos_S(c_1) = pos_S(f) - pos_S(e) = 6 - 5 = 1$

Even for the contiguity constraint the condition is satisfied for both i, therefore C is contiguous with respect to S. This means that the elements in C are consecutive in S, so there are no gaps. As shown in this example, if C is contiguous in S, then its elements are necessarily ordered as they appear consecutively in S. However, being ordered does not necessarily mean being contiguous, as there may be other elements of S in between.

Example 3 — Not ordered and Not Contiguous. Again with the same sequence:

$$S = \underset{1}{[a,} \ \underset{2}{b,} \ \underset{3}{c,} \ \underset{4}{d,} \ \underset{5}{e,} \ \underset{6}{f]}$$

Take:

$$C = \underset{1}{[e,} \ \underset{2}{a,} \ \underset{3}{c]}$$

- $(i = 1) \ \rightarrow \ pos_S(c_2) > pos_S(c_1) = pos_S(a) > pos_S(e) = 1 > 5$
- $(i = 2) \ \rightarrow \ pos_S(c_3) > pos_S(c_2) = pos_S(c) > pos_S(a) = 3 > 1$

C is not ordered with respect to S, since the condition fails for $i = 1$. This means that at least one element of C appears in S before another element that should precede it, breaking the required order. As ordering is a necessary prerequisite for contiguity, C cannot satisfy the contiguity constraint either. However, we will verify this for completeness.

- $(i = 1) \rightarrow pos_S(c_2) - pos_S(c_1) = pos_S(a) - pos_S(e) = 1 - 5 = -4 \neq 1$
- $(i = 2) \ \rightarrow \ pos_S(c_2) - pos_S(c_1) = pos_S(c) - pos_S(a) = 3 - 1 = 2 \neq 1$

As mentioned above, the sequence C could not be contiguous. This is evident from the condition that is not satisfied for both i.

3.3 Overlap Definitions with Constraints

Given two tables $R(X)$ and $S(Y)$, the overlap is defined as a common rectangular subtable obtained by selecting subsets of rows and columns from both tables that match on their values.

Definition 1. *Under the* ***ordering constraint****, the selected rows and columns forming the largest common subtable must preserve their relative order with respect to both original tables, R and S. In other words, the sequences of selected rows and columns must satisfy the definition of* ***ordered subsequence*** *introduced in Section 3.2.*

The example shown in Figure 3.3 illustrates the application of the ordering constraint in the search for the largest overlap between two tables. Table R contains six rows and five columns, while table S contains five rows and five columns. The ordering constraint requires that the selected rows and columns in the final table maintain the same relative order in both tables. For this reason, the columns Brand / Supplier,

	Brand	Category	Country	Rating	Color
1	Apple	Electronics	USA	4.8	White
2	Adidas	Shoes	China	4.2	Blue
3	Samsung	Electronics	South Korea	4.6	Black
4	Nike	Shoes	China	4.5	Black
5	Puma	Shoess	USA	4.1	Red
6	Sony	Electronics	Japan	4.7	Silver

(a) Table R(X)

	Supplier	Country	Type	Stars	Color
1	Samsung	South Korea	Electronic	4.6	Black
2	Nike	China	Shoes	4.	Black
3	Reebok	UK	Shoes	4.0	Green
4	Apple	USA	Electronics	4.8	White
5	H&M	Sweden	Clothing	3.5	Red

(b) Table S(Y)

1	Samsung	South Korea	Black
2	Nike	China	Black

Figure 3.3: Largest Overlap Under Ordering Constraint

Country, Color are chosen, as they are present in both tables and respect the same relative sequence. Similarly, the selected rows correspond to Samsung and Nike, which appear in both tables in the correct order. In contrast, although common to both tables, the Apple row is not included in the overlap because in R it precedes Samsung, while in S it follows it, thus violating the ordering constraint. Similarly, the Category/Type column cannot be considered, as it appears before Country in one table and after it in the other, resulting in a reversal of the relative order. Figure 3.3 thus shows the largest common rectangular subtable that respects the constraints, consisting of two rows and three columns, that is the largest overlap between R and S.

Definition 2. *Under the* ***contiguity constraint****, the selected rows and columns forming the largest common subtable must appear as contiguous blocks within both tables, R and S. This means that, according to the definition of* ***contiguous subsequence*** *provided in Section 3.2, the chosen rows and columns must not only preserve their relative order but also occur consecutively, without gaps, in each table.*

	Brand	Category	Country	Rating	Color
1	Apple	Electronics	USA	4.8	White
2	Adidas	Shoes	China	4.2	Blue
3	Samsung	Electronics	South Korea	4.6	Black
4	Nike	Shoes	China	4.5	Black
5	Puma	Shoess	USA	4.1	Red
6	Sony	Electronics	Japan	4.7	Silver

(a) Table R(X)

	Supplier	Country	Type	Stars	Color
1	Samsung	South Korea	Electronic	4.6	Black
2	Nike	China	Shoes	4.	Black
3	Reebok	UK	Shoes	4.0	Green
4	Apple	USA	Electronics	4.8	White
5	H&M	Sweden	Clothing	3.5	Red

(b) Table S(Y)

In the example shown in Figure 3.5, the contiguity constraint is applied in the

Figure 3.5: Largest Overlap Under Contiguous Constraint

search for the largest overlap between the two tables. This constraint requires that the selected rows and columns not only preserve their relative order across both tables, but also form contiguous subsequences. As a result, only the rows corresponding to Samsung and Nike are retained, since they appear consecutively in both tables and respect the required ordering. Although other rows such as Apple are present in both tables, they cannot be included because their position breaks the contiguity condition: in table R, Apple is separated from Samsung and Nike by intervening rows, while in table S it follows them. Similarly, additional columns that might otherwise contribute to the overlap are excluded whenever their sequence would disrupt contiguity. Consequently, the largest valid overlap under this stricter constraint, reported in Figure 3.5, consists of two consecutive rows aligned across both tables.

In the context of computing the largest overlap between two tables, the notions of ordering and contiguity introduced above play a crucial role in determining which portions of rows and columns can be matched. These constraints can be naturally mapped to two well-known algorithmic problems: Longest Common Subsequence (LCS) and Longest Common Substring (LCStr). The following two sections provide a detailed examination of these two fundamental string matching problems.

3.4 Longest Common Subsequence

A Longest Common Subsequence (LCS) [12] is defined as the longest sequence that appears in the same relative order within all sequences of a given set, most commonly between two sequences. The LCS does not require the elements to occupy consecutive positions, but only to preserve their order. For example, consider the sequences [A,B,C,D] and [A,C,B,A,D].

- They have five length 2 common subsequences: [A,B], [A,C], [A,D], [B,D], and [C,D]
- two length 3 common subsequences: [A,B,D] and [A,C,D];
- and no longer common subsequences.

So, [A,B,D] and [A,C,D] are their *longest common subsequences*.

The computation of LCS is a classical problem in computer science and serves as the foundation for a wide range of applications. For instance, it supports tools that highlight differences between text files and is extensively employed in version control systems to reconcile concurrent modifications of shared documents. Beyond software engineering, LCS has proven essential in computational linguistics, where it is used for text similarity and plagiarism detection, and in bioinformatics, where it supports DNA and protein sequence alignment by identifying conserved subsequences across different organisms.

From a computational perspective, the LCS problem is usually solved by dynamic programming, which guarantees polynomial-time solutions, typically $O(n * m)$ for sequences of length n and m. However, given the high computational cost for very large sequences, numerous optimizations and approximate algorithms have been proposed, including parallel methods, compressed data structures, and heuristic-based approaches, making LCS still an active area of research.

3.4.1 Ordering constraint and LCS

When only ordering is required, the overlap table must preserve the relative sequence of rows and columns across both original tables, but non contiguous selections are allowed. This condition corresponds directly to the Longest Common Subsequence problem, which aims to identify the longest sequence of elements appearing in both input sequences in the same order. Applying the LCS algorithm to the columns of two tables allows us to identify the largest overlap under the ordering constraint. This can be achieved either by comparing pairs of individual columns across the two tables, or by combining multiple columns into larger composite sequences, thus generating different configurations from which the largest overlap can be determined.

3.5 Longest Common Substring

A Longest Common Substring (LCStr) [13] is defined as the longest sequence of characters (or, more generally, elements) that appears contiguously in all sequences under comparison, most often between two strings. In contrast to the Longest Common Subsequence (LCS), a substring requires the elements to occupy consecutive positions in the original sequences. For example, consider the sequences [A,B,A,B,C], [B,A,B,C,A].

- They have only one longest common substring, [A,B,C] of length 3.
- Other common substrings are [A], [A,B], [B], [B,A], [B,C] and [C] of length 1 or 2.

The LCStr problem is also a fundamental task in computer science with wide range of applications. It is widely used in string matching algorithms, plagiarism detection,

and information retrieval, where identifying exact shared fragments is crucial. In bioinformatics, LCStr is employed for discovering patterns in DNA and protein sequences, providing insights into functional or evolutionary relationships. Similarly, in data compression, LCStr helps detect repeated substrings that can be encoded more efficiently.

From a computational perspective, LCStr can be solved efficiently using dynamic programming in $O(n * m)$ for sequences of length n and m. More advanced methods exploit suffix trees or suffix arrays, enabling faster solutions in linear or near-linear time.

3.5.1 Contiguity constraint and LCStr

When both ordering and contiguity constraints are imposed, the overlap between two tables must consist of compact blocks of rows and columns, with no gaps. This situation corresponds to the Longest Common Substring (LCStr) problem, which identifies the longest contiguous subsequence shared by the inputs. Applied to tables, the LCStr algorithm can operate on pairs of individual columns or on sequences formed by combining multiple columns, thus detecting the largest contiguous overlap where elements are required to be both ordered and adjacent.

3.6 Using LCS and LCStr Algorithms in Practice

In this section, we examine the practical application of the Longest Common Subsequence (LCS) and Longest Common Substring (LCStr) algorithms for comparing two tables and determine the largest overlap between them.

3.6.1 Comparing Single Columns

In the context of finding the largest overlap between two tables, we leverage the fact that the LCS algorithm can take as input two arbitrary lists of elements, e.g. strings, int. This makes it particularly suitable when comparing one column from each two tables: the values of the two columns can be directly represented as sequences of elements, and the largest overlap under the ordering constraint reduces to the problem of computing their Longest Common Subsequence.

For example, given the two columns in Figure 3.6 extracted from tables R and S, can be represented as sequences of elements as follows:

- col_R = ['USA', 'China', 'South Korea', 'China', 'USA', 'Japan']
- col_S = ['South Korea', 'China', 'UK', 'USA', 'Sweden']

	Country
1	USA
2	China
3	South Korea
4	China
5	USA
6	Japan

	Country
1	South Korea
2	China
3	UK
4	USA
5	Sweden

Figure 3.6

By applying the LCS algorithm, pseudocode shown in Figure 3.8, to the two lists, col_R and col_S, the algorithm systematically compares elements from both lists to identify the longest sequence of items that appear in the same order. In this case, it determines ['South Korea', 'China', 'USA'] as the longest common subsequence, as illustrated from the output shown in Figure 3.7. In this setting, the application is straightforward since each element in the sequence corresponds to a single row entry in the column. However, when the comparison involves multiple columns simultaneously, the problem becomes more complex, as rows must be treated as structured objects rather than simple values. In such cases, additional techniques are required to encode and align multi attribute rows before applying the algorithms. We will discuss this in the next section.

```
col_R = ['USA', 'China', 'South Korea', 'China', 'USA', 'Japan']
col_S = ['South Korea', 'China', 'UK', 'USA', 'Sweden']
length, indices = longest_common_subsequence(col_R, col_S)
print("Length of LCS:", length)
print("Indices in col_R:", indices)
✓ 0.0s
Length of LCS: 3
Indices in col_R: [2, 3, 4]
```

Figure 3.7

For the implementation, we used a standard dynamic programming approach to the Longest Common Subsequence problem, widely documented in the literature. The function shown in Figure 3.8 is an adaptation of this standard method: it not only computes the length of the LCS between two sequences, but also returns the indices of the matched elements in the first sequence, thanks to the backtracking function. In Figure 3.7, the output of the algorithm is shown, consisting of the length and the corresponding list of indices. In this case, the indices [2, 3, 4] indicate the elements in the first sequence that are part of the longest common subsequence. This

feature is particularly useful in our context, as it allows us to reconstruct the actual subsequence when needed, rather than being limited to its length. In the case of tables, this means that we can directly identify which rows participate in the overlap, thus bridging the gap between the abstract sequence alignment and its concrete application to tabular data.

Algorithm 3 LongestCommonSubsequence function()

Input: Two sequences $seq1[1 \dots m]$, $seq2[1 \dots n]$
Output: Length of LCS and list of indices in $seq1$

```
f[0...m][0...n] ← 0
for i = 1 to m do
    for j = 1 to n do
        if seq1[i] = seq2[j] then
            f[i][j] ← f[i − 1][j − 1] + 1
        else
            f[i][j] ← max(f[i − 1][j], f[i][j − 1])
        end if
    end for
end for

/* Backtracking to recover indices of LCS in seq1 */
i ← m, j ← n, indices_seq1 ← [ ]
while i > 0 and j > 0 do
    if seq1[i] = seq2[j] then
        indices_seq1.append(i − 1)
        i ← i − 1
        j ← j − 1
    else if f[i − 1][j] > f[i][j − 1] then
        i ← i − 1
    else
        j ← j − 1
    end if
end while
return (f[m][n], indices_seq1)
```

Figure 3.8

In relation to the second constraint, and still considering the case of single columns, the problem of identifying the largest overlap can be reformulated as the computation of the longest common substring between the two columns. Consequently, the reasoning developed for the ordering constraint remains valid. The only difference concerns the adoption of the LCStr algorithm in place of the LCS. For the sake of completeness, an illustrative example is provided below.

In Figure 3.9, which reports the first two columns extracted from tables R and S, the largest overlap, highlighted in green, consists of only two elements: ['South Korea', 'China']. The element 'USA' is not included, as in the right hand column the presence of 'UK' between 'China' and 'USA' interrupts the sequence, thus violating the contiguity constraint.

	Country
1	USA
2	China
3	South Korea
4	China
5	USA
6	Japan

	Country
1	South Korea
2	China
3	UK
4	USA
5	Sweden

Figure 3.9

In this case, we also relied on a dynamic programming solution for the implementation of the Longest Common Substring algorithm. The pseudocode of this standard algorithm is presented in Figure 3.10. It outputs both the length and the starting index in the first sequence, which can then be used to reconstruct the identified longest common substring.

Algorithm 4 LongestCommonSubstring function()

Input: Two sequences $seq1[1 \dots m]$, $seq2[1 \dots n]$
Output: Length of the longest common substring and its starting index in $seq1$

```
dp[0...m][0...n] ← 0
max_len ← 0
end_seq1 ← 0
for i = 0 to m − 1 do
    for j = 0 to n − 1 do
        if seq1[i] = seq2[j] then
            dp[i + 1][j + 1] ← dp[i][j] + 1
            if dp[i + 1][j + 1] > max_len then
                max_len ← dp[i + 1][j + 1]
                end_seq1 ← i + 1
            end if
        else
            dp[i + 1][j + 1] ← 0
        end if
    end for
end for
/* Recover the starting index in seq1 */
start_seq1 ← end_seq1 − max_len
return (max_len, start_seq1)
```

Figure 3.10

3.6.2 Comparing Multiple columns

When extending the overlap computation to multiple columns, the problem requires treating the rows not as single values but as tuples of attributes. To adapt the LCS (or the LCStr) algorithm to this setting, we encode each row into a single sequence element by concatenating the values of the selected columns. For example, when comparing the first two columns of the two tables, we first extract the corresponding columns for each table, shown in Figure 3.11. That is, *Brand* and *Country* for R and *Supplier* and *Country* for S. From these, we build two transformed columns, where each row is represented by the concatenation of the attribute values. The result of this process is shown in Figure 3.12. The LCS algorithm can then be applied directly to these new encoded sequences, *combined_R* and *combined_S*.

- *combined_R* = [AppleUSA, AdidasChina, SamsungSouthKorea, NikeChina, PumaUSA, SonyJapan]
- *combined_S* = [SamsungSouthKorea, NikeChina, ReebokUK, AppleUSA, H&MSweden]

The algorithm then yields the subsequence [SamsungSouthKorea, NikeChina] as the longest ordered overlap between the two columns from both tables, as illustrated in Figure 3.13. In this way, the concatenation step effectively generalizes the one

dimensional case to multi attribute rows, allowing the LCS algorithm to operate seamlessly while ensuring that overlaps respect all selected columns simultaneously.

	Brand	Country
1	Apple	USA
2	Adidas	China
3	Samsung	South Korea
4	Nike	China
5	Puma	USA
6	Sony	Japan

	Supplier	Country
1	Samsung	South Korea
2	Nike	China
3	Reebok	UK
4	Apple	USA
5	H&M	Sweden

Figure 3.11

1	AppleUSA
2	AdidasChina
3	SamsungSouth Korea
4	NikeChina
5	PumaUSA
6	SonyJapan

1	SamsungSouth Korea
2	NikeChina
3	ReebokUK
4	AppleUSA
5	H&MSweden

Figure 3.12

```
    length, index = compute_long_comm_subseq(r_tab_col_list, s_tab_col_list, seed_ids=(0,3), seeds=seeds)
    print('Length of Longest Common Subsequence:', length)
    print('Index in r_tab_col_list:', index)
✓ 0.0s
combined_r: ['AppleUSA', 'AdidasChina', 'SamsungSouth Korea', 'NikeChina', 'PumaUSA', 'SonyJapan']
combined_s: ['SamsungSouth Korea', 'NikeChina', 'ReebokUK', 'AppleUSA', 'H&MSweden']
Length of Longest Common Subsequence: 2
Index in r_tab_col_list: [2, 3]
```

Figure 3.13

Figure 3.14 presents the implementation of the previously illustrated example. Specifically, the algorithm takes as input the tables and, through seed based column selection, constructs reduced tables restricted to the relevant attributes. Since the standard Longest Common Subsequence algorithm is defined for one dimensional sequences, it cannot be applied directly in the multi column case. Therefore, a preprocessing step is required, where the selected column values are concatenated row wise into string sequences, enabling the LCS algorithm to operate correctly over multi attribute rows.

Algorithm 5 LCSubseq_overCombinedColumns function()

Input: Tables $R(X)$ and $S(Y)$; *seed_ids*: list of target seeds; *seeds*: complete list of detected seeds

Output: Length and indices of the longest common subsequence

```
/* Extract column pairs from seeds using given indices */
col_pairs ← [ seeds[i][0] | i ∈ seed_ids ]

/* Build reduced tables with only selected target columns */
new_r_tab ← [ r_tab[m[0]] | m ∈ col_pairs ]
new_s_tab ← [ s_tab[m[1]] | m ∈ col_pairs ]

/* Combine entries row-wise into string sequences */
combined_r ← [ concat(row) | row ∈ zip(new_r_tab) ]
combined_s ← [ concat(row) | row ∈ zip(new_s_tab) ]

/* Finally apply the original algorithm */
(length, idx) ← LongestCommonSubsequence(combined_r, combined_s)
return (length, idx)
```

Figure 3.14

Regarding the application of the alternative algorithm, LCStr, to the multi column case, the underlying logic remains identical: the same preprocessing steps, including column selection and row-wise concatenation, are employed. For this reason, we avoid repeating the entire procedure. However, what is important to highlight is that in the implementation shown in Figure 3.14, the Longest Common Substring algorithm is applied instead of the Longest Common Subsequence algorithm.

Chapter 4

LCS-Based Largest Overlap Detection

In this section, we present the algorithm used as the foundation for our approach to detect the largest overlap between two tables. This algorithm was first introduced in the related work SLOTH by Luca Zecchini et al. [9], which also tackles the problem of identifying the largest overlap, but in that context no constraints were imposed. Both rows and columns could be freely reordered to maximize the overlap. In contrast, our setting introduces stricter requirements, as overlaps must respect both ordering and contiguity constraints. For this reason, we adopt the original algorithm as a baseline and adapt it to meet these new conditions.

4.1 Algorithm Overview

The algorithm considers as input two tables, denoted as $R(X)$ and $S(Y)$, and returns their largest overlap. If the two tables have no cells in common, the area of the largest overlap is equal to zero. It must take into account all possible mappings that could yield the largest overlap between the two tables. We refer to these mappings as *candidates*, since each of them represents a potential top mapping. To generate the set of candidates, the algorithm first evaluates all single attribute mappings, namely those where X_M consists of a single attribute x. For each such mapping, it computes the overlap area between $R(x)$ and $S(M(x))$. If the resulting overlap is greater than zero, the mapping is labeled as a *seed* and stored in a dedicated list.

Because every multi attribute mapping can be constructed as a combination of single attribute mappings, candidates can then be derived by combining seeds, and these combinations can naturally be represented as nodes of a lattice. As we move upward in the lattice, the width of the overlap grows, but its overall area does not necessarily increase, since the height may shrink when additional columns are introduced. In particular, the seed with the smallest overlap area within a given

combination determines an upper bound on the candidate's height, and consequently on its area. This bounding property can be leveraged in two ways: to prune the lattice by discarding unpromising candidates, and to prioritize candidates according to their maximum attainable area.

4.2 Detect Seeds

The first step consists in identifying the seeds, a task handled by the $findSeeds()$ function. This procedure examines all possible single attribute mappings between the two tables and computes their overlap area. Whenever the area is strictly greater than zero, the corresponding mapping is added to the set of seeds, together with its area value. We denote the total number of identified seeds as s, where $s = |Seeds|$. If an attribute from one table overlaps with cells from multiple attributes in the other table, it will give rise to multiple seeds. However, since a mapping is defined as a bijective function, a valid multi attribute mapping can include at most one seed from each group of seeds sharing a common attribute. Throughout the process, we implicitly consider only valid mappings, while invalid ones are automatically discarded.

Algorithm 1 findSeeds() function

Input: The two tables $R(X)$ and $S(Y)$
Output: List of the detected seeds

```
1  Seeds ← ∅
2  foreach x ∈ X do
3      foreach y ∈ Y do
4          seed.M ← M : x → y
5          if row_constraint == 'none' then
6              seed.L ← |(R[x] ∩* S[x]|                    // bag intersection
7          if row_constraint == 'ordered' then
8              seed.L ← LongestCommonSubsequence(R[x], S[x])
9          if row_constraint == 'contiguous' then
10             seed.L ← LongestCommonSubstring(R[x], S[x])
11         if seed.L > 0 then
12             Seeds.append(seed)
13 return Seeds
```

Figure 4.1

The $findSeeds()$ function, reported in Figure 4.1, has been adapted to handle different types of row constraints in the computation of seeds. In the original formulation, corresponding to the case where the row constraint is set to $'none'$, the function simply checks the bag intersection between the two columns and computes the overlap area as the cardinality of that intersection. In our implementation, however, two additional cases have been introduced to account for stricter constraints:

- When the constraint is *ordered*, the function applies the $LongestCommonSubsequence(LCS)$ algorithm between the two columns, thereby ensuring that the relative order of tuples is preserved.
- When the constraint is *contiguous*, the function relies on the $LongestCommonSubstring(LCStr)$ algorithm, which enforces the requirement that the overlap must consist of consecutive rows.

Through this extension, the $findSeeds()$ function preserves its original behavior in the unconstrained setting while also supporting more complex scenarios in which ordering and contiguity constraints must be satisfied.

4.3 Column Constraints in Overlap Detection

In order to correctly identify the largest overlap between two tables, it is crucial to regulate how column combinations are generated and validated. The procedure implemented in the $genCand()$ function illustrated in Figure 4.2 addresses this need by systematically producing candidate column sets and ensuring that they comply with the predefined column constraint. The function iteratively explores combinations of columns drawn from the detected seeds and constructs candidate mappings, for which properties such as width, height, and area are computed.

A central step in this process is the validation of candidates according to the column constraint parameter. Two distinct constraints are considered. When the constraint is set to *ordered*, the algorithm verifies that the column indices in both tables follow a strictly increasing order. This ensures that the structural consistency of the alignment is preserved, preventing cases in which the order of columns would be disrupted. Conversely, when the constraint is set to *contiguous*, the algorithm requires that column indices appear as consecutive sequences, thereby excluding candidates where gaps occur between matched columns.

Only those candidates that satisfy the chosen constraint and whose area exceeds a specified threshold are retained in the candidate queue. In this way, the algorithm not only enforces structural requirements on the alignment but also filters out configurations that would not meaningfully contribute to the identification of the largest overlap. The resulting set of candidates thus represents a refined search space that respects the column constraints while driving the computation toward the ultimate objective of maximizing the overlap between the two tables.

4.4 Row Constraints in Overlap Detection

In the second stage, once candidate column combinations have been generated, it becomes necessary to determine their actual overlap in terms of rows. The $verCand()$ function illustrated in Figure 4.3 is responsible for this task, as it evaluates each

Algorithm 1 genCand() function

Input: The priority queues for the lattice levels and for the candidates, the sorted list of the detected seeds
Output: The updated priority queues

1: $topL \leftarrow Levels.pop()$ ▷ top level
2: **for all** $comb \in combs(Seeds[:topL.i], topL.w)$ **do**
3: $valid = True$
4: $cand.M = comb$ ▷ mapping
5: $cand.w = topL.w$ ▷ width
6: $cand.h = Seeds[topL.i].A$ ▷ max height
7: $cand.A = cand.w \cdot cand.h$ ▷ max area
8: $cand.O \leftarrow \emptyset$ ▷ overlap
9: **if** $col\_constraint == \ 'ordered'$ **then**
10: $//\ valid = False\ if\ columns\ order\ is\ NOT\ increasing$
11: $first\_increasing \leftarrow$ **all** $(first[i] < first[i+1])$
12: $second\_increasing \leftarrow$ **all** $(second[i] < second[i+1])$
13: $valid \leftarrow first\_increasing\ \&\ second\_increasing$
14: **end if**
15: **if** $col\_constraint == \ 'contiguous'$ **then**
16: $//\ valid = False\ if\ columns\ order\ is\ NOT\ streactly\ increasing$
17: $first\_consecutive \leftarrow$ **all** $(first[i+1] - first[i]! = 1)$
18: $second\_consecutive \leftarrow$ **all** $(second[i+1] < second[i]! = 1)$
19: $valid \leftarrow first\_consecutive\ \&\ second\_consecutive$
20: **end if**
21: **if** $cand.A \geq \theta\ \&\ valid$ **then**
22: $Candidates.push(cand)$
23: **end if**
24: **end for**
25: **return** $Levels, Candidates$

Figure 4.2

candidate against the specified row constraint and computes the effective overlap area.

Depending on the row constraint, different strategies are adopted. If no constraint is enforced ($'none'$), the algorithm simply computes the bag intersection between the rows of the two tables, thereby capturing all common tuples without imposing any ordering. If the constraint is set to *ordered*, the overlap is determined by computing the longest common subsequence (LCS) between the selected row sets. This approach ensures that only row alignments that respect sequential ordering are retained. Finally, when the constraint is *contiguous*, the function relies on the longest common substring (LCStr) algorithm, which enforces that the overlap corresponds to a consecutive block of rows in both tables.

After the overlap set is computed, the candidate's actual height is given by the size of this set, while the actual area is obtained by multiplying the height by the previously determined width. Candidates whose area is greater than or equal to the current threshold are retained and reinserted into the candidate queue, whereas those falling below the threshold are pruned.

Algorithm 2 verCand() function

Input: The two tables $R(X)$ and $S(Y)$, the top candidate, the priority queues for the lattice levels and for the candidates
Output: The updated priority queues

1: **if** $row\_constraint == 'none'$ **then**
2: $topC.O \leftarrow$ ***compute_BagIntersection***$(R[X_{topC.M}], S[Y_{topC.M}])$
3: **end if**
4: **if** $row\_constraint == 'ordered'$ **then**
5: $topC.O \leftarrow$ ***compute_LCSubsequence***$(R[X_{topC.M}], S[Y_{topC.M}])$
6: **end if**
7: **if** $row\_constraint == 'contiguous'$ **then**
8: $topC.O \leftarrow$ ***compute_LCSubstring***$(R[X_{topC.M}], S[Y_{topC.M}])$
9: **end if**
10: $topC.h \leftarrow |topC.O|$ ▷ actual height
11: $topC.A \leftarrow topC.w \cdot topC.h$ ▷ actual area
12: **if** $topC.A \geq \theta$ **then**
13: $\theta \leftarrow topC.A$
14: $Candidates.push(topC)$ ▷ reinsert candidate
15: **for all** $cand \in Candidates$ **do**
16: **if** $cand.A < \theta$ **then**
17: $Candidates.delete(cand)$ ▷ prune candidate
18: **end if**
19: **end for**
20: **end if**
21: **return** $Candidates$

Figure 4.3

The final step consists in integrating the candidate generation and candidate verification procedures into a unified algorithm for largest overlap detection. The $genCand()$ function is responsible for expanding the search space by producing new candidate column combinations under the specified column constraints, while $verCand()$ validates these candidates by enforcing the row constraints and computing their effective overlap area. By combining these two complementary operations within a coordinated iterative framework, the algorithm progressively explores the space of possible alignments, prunes unpromising candidates, and retains only those that exceed the pruning threshold. This iterative refinement continues until no further candidates remain to be generated or verified. The result is the set of the largest overlaps between the two tables.

Chapter 5

Mapping-Based Largest Overlap Detection

This chapter introduces a variant of the method developed to address the problem of finding the largest overlap between two tables. Unlike the first approach, which relied on the classical Longest Common Subsequence (LCS) and Longest Common Substring (LCStr) algorithms, the second approach follows a different strategy. Specifically, it is based on the construction of an explicit mapping of the common elements between the two tables. The process will be illustrated step by step, followed by the presentation of two new algorithms designed to operate on the constructed mapping, rather than directly on the table columns.

5.1 Definition of Mapping

In contrast to methods that operate directly on table columns, the mapping approach explicitly captures the correspondences between the positions of common elements in the two tables. The resulting structure serves as a compact representation of all possible matches, enabling the detection of the largest overlap that adheres to the constraints introduced in Section 3.2, specifically ordering and contiguity.

Let $R(X)$ and $S(Y)$ be two tables defined as:

$$R(X) = \{r_{i,j} \mid 0 \leq i < m,\ 0 \leq j < n_R\}$$

$$S(Y) = \{s_{k,l} \mid 0 \leq k < p,\ 0 \leq l < n_S\}$$

where i, k denote row indices and j, l denote column indices.

Definition 1. A *Seed* is a pair of columns (j, l), where $j \in [0, n_R)$ refers to a column of table R and $l \in [0, n_S)$ refers to a column of table S. Each seed establishes a

possible comparison between the columns of the two tables.

Definition 2. Given a seed (j, l), the *Mapping* of that seed is defined as:

$$M_{j,l} = \{(i, k) \mid r_{i,j} = s_{k,l}\},$$

that is, the set of all row index pairs (i, k) such that the value in row i of column j in R coincides with the value in row k of column l in S.

Definition 3. The *Global Mapping* between R and S is the collection of all seeds together with their associated mappings:

$$M = \{\big((j, l), M_{j,l}\big) \mid j \in [0, n_R),\ l \in [0, n_S)\}.$$

The global mapping thus compactly encodes all possible overlaps between two tables at the attribute level, and it serves as the basis for the algorithms introduced in the following sections.

5.2 Example of Mapping

To illustrate the concept of mapping, consider the two tables $R(X)$ and $S(Y)$ reported in Figures 5.1a and 5.1b. Several values appear in both tables, such as *Apple*, *Samsung*, *Nike*, *China*, *USA*, *Electronics*, *Shoes*, *Black*, *White*, and *Red*.

	Brand	Category	Country	Rating	Color
1	Apple	Electronics	USA	4.8	White
2	Adidas	Shoes	China	4.2	Blue
3	Samsung	Electronics	South Korea	4.6	Black
4	Nike	Shoes	China	4.5	Black
5	Puma	Shoess	USA	4.1	Red
6	Sony	Electronics	Japan	4.7	Silver

(a) Table R(X)

	Supplier	Country	Type	Stars	Color
1	Samsung	South Korea	Electronic	4.6	Black
2	Nike	China	Shoes	4.	Black
3	Reebok	UK	Shoes	4.0	Green
4	Apple	USA	Electronics	4.8	White
5	H&M	Sweden	Clothing	3.5	Red

(b) Table S(Y)

Figure 5.1

The *Global Mapping* between these two tables can be expressed as follows:

$$\begin{aligned} M = [&((\mathbf{1,1}), [(1,4),(3,1),(4,2)]),\\ &((\mathbf{2,3}), [(1,4),(2,2),(2,3),(3,4),(4,2),(4,3),(6,4)]),\\ &((\mathbf{3,2}), [(1,4),(2,2),(3,1),(4,2),(5,4)]),\\ &((\mathbf{4,4}), [(1,4),(3,1)]),\\ &((\mathbf{5,5}), [(1,4),(3,1),(3,2),(4,1),(4,2),(5,5)])] \end{aligned}$$

In this representation:

- Each pair of columns (j, l) is a *Seed* (the index pairs in bold), where j refers to a column in $R(X)$ and l to a column in $S(Y)$.
- For each *Seed*, the associated list of index pairs (i, k) defines the mapping, i.e. all row-level correspondences where values in $R_{i,j}$ and $S_{k,l}$ are equal.

For example, the *Seed* **(1,1)**, comparing *Brand* in R with *Supplier* in S, has the mapping $[(1, 4), (3, 1), (4, 2)]$:

- $(1, 4)$: *Apple* appears at row 1 of R and row 4 of S.
- $(3, 1)$: *Samsung* appears at row 3 of R and row 1 of S.
- $(4, 2)$: *Nike* appears at row 4 of R and row 2 of S.

The Seed **(5,5)**, comparing the column *Color* in both tables, has the mapping

$$[(1, 4), (3, 1), (3, 2), (4, 1), (4, 2), (5, 5)]$$

which identifies matches such as *White*, *Black*, and *Red* across multiple rows. This example highlights how the mapping encodes, for each pair of comparable columns, the exact positions of the common values in the two tables.

The Figure 5.2 provides the pseudocode of the function *FindMatches()* to identify correspondences between two lists (or columns). The algorithm proceeds by first constructing two dictionaries, idx_map1 and idx_map2, which associate each value in the respective columns with the set of indices at which the value occurs. Once these dictionaries are built, the algorithm determines the set of common values shared by the two columns. For each shared value, it then generates all index pairs (i, j) such that $col1[i] = col2[j]$. The final output is a collection of these index pairs, which represent the matches between the two columns, i.e. the mapping.

This function constitutes a fundamental step in the mapping-based approach. Specifically, it is embedded within the *findSeeds()* function, discussed and reported in section 4.2, which systematically iterates over all possible combinations of columns between the two tables under comparison. For each pair of columns, *findSeeds()* calls the matching function to identify the shared values and their corresponding index mappings. These elementary matches (or seed matches) provide the basis for subsequent stages of the overlap detection process, as they represent the initial points of alignment from which larger structural correspondences can be derived.

Algorithm 9 FindMatches function()

Input: Two lists $col1$, $col2$
Output: List of matching index pairs (i, j) where $col1[i] = col2[j]$

```
idx_map1 ← {}
idx_map2 ← {}
for i, val in enumerate(col1) do
    idx_map1[val] ← i
end for
for i, val in enumerate(col2) do
    idx_map2[val] ← j
end for
common_values ← idx_map1.keys() ∩ idx_map2.keys()
results ← [ ]
for val in common_values do
    for i in idx_map1[val] do
        for j in idx_map2[val] do
            results.append(i, j)
        end for
    end for
end for
return results
```

Figure 5.2

5.3 Role of Mapping in Overlap Detection

Once the *Global Mapping* has been constructed, it becomes the foundation for finding the largest overlap between the two tables. In this framework, the seeds (j, l) identify the columns that are compared, while their associated mappings provide all row level correspondences. The problem of overlap detection can then be reformulated as the task of finding the longest consistent sequence of index pairs within the mapping, under specific constraints.

Ordering constraint.
To capture overlaps that preserve the relative order of rows across the two tables, the *Longest Increasing Pairs* algorithm, described in section 5.4, is applied to the mapping. The algorithm searches for the longest subsequence of index pairs (i, k) such that both i and k are strictly increasing. This ensures that the overlap respects the natural order of rows, in analogy with the Longest Common Subsequence but operating directly on indices of the shared values.

Contiguity constraint.
To capture overlaps that are not only ordered but also contiguous, the *Longest Consecutive Pairs* algorithm described in section 5.5, is applied. Here, the algorithm searches for the longest sequence of index pairs (i, k) such that both indices progress in consecutive steps, i.e. $(i + 1, k + 1)$. This guarantees that the overlap is formed by uninterrupted blocks of rows, corresponding to the notion of Longest Common

Substring, adapted to the mapping.

By decoupling the discovery of correspondences (mapping index construction) from the computation of the largest overlap, this approach achieves a modular design with several advantages:

1. **Flexibility:** the same mapping can be reused with different algorithms or constraints.
2. **Efficiency:** in cases where only a small number of values are shared between tables, the size of the mapping index is significantly smaller than the Cartesian product of rows, reducing computational complexity.
3. **Extensibility:** additional constraints (e.g. weighted similarities) can be incorporated into the mapping without altering the overall framework.

In other words, this approach enables a clear separation between identifying correspondences and applying algorithms to extract the largest overlap under ordering and contiguity constraints. The mapping thus serves as a flexible and reusable foundation for overlap detection. In the following sections, the two algorithms introduced will be described in detail, including their formal definitions, the implementation, and illustrative examples based on the constructed mappings.

5.4 Longest Increasing Pairs

The algorithm presented in Figure 5.3 addresses the problem of determining the length of the longest subsequence of pairs (f, s) in which both components are strictly increasing. The procedure begins with a trivial base case: if the input list of pairs is empty, the function immediately returns zero. Otherwise, the list is sorted in ascending order according to the first component f. In the case of ties, pairs are further ordered by the second component s in descending order. This ordering strategy ensures that pairs with identical values of f cannot be mistakenly included in the same subsequence, as they would necessarily violate the strictly increasing condition.

Once the pairs are properly ordered, the problem is reduced to computing the *Longest Increasing Subsequence* with respect to the second component s. To achieve this efficiently, the algorithm employs a method based on binary search. Specifically, a dynamic list *seq* is maintained to represent the best subsequence identified so far. For each element s, its appropriate position within *seq* is determined using binary search. If s is larger than all current elements in *seq*, it is appended to the end of the list; otherwise, it replaces the first element that is greater than or equal to it. This mechanism guarantees that *seq* always remains a valid candidate subsequence while enabling efficient updates.

The length of *seq* at the end of the procedure corresponds to the length of the longest subsequence in which both components f and s increase strictly. From a computational perspective, the sorting step dominates with a complexity of $O(n \log n)$.

Algorithm 7 LongestIncreasingSubsequencePairs function()

Input: *pairs*: list of matching index pairs
Output: Length of the longest subsequence where both f and s are strictly increasing

```
pairs.sort(lambda x : (x[0], -x[1]))
seq ← [ ]
for (f, s) in pairs do
    // position of s in seq found with binary search
    idx ← bisect.bisect_left(seq, s)
    if idx = len(seq) then
        seq.append(s)
    else
        seq[idx] = s
    end if
end for
return len(seq)
```

Figure 5.3

To better illustrate how the algorithm works, consider the following example. Suppose that the input is given as a list of index pairs, representing a particular mapping for a pair of columns:

$$\text{pairs} = [(5, 4), (6, 4), (6, 7), (2, 3)].$$

The algorithm first sorts the list by the first component in ascending order and, in the case of equality, by the second component in descending order. This yields the following:

$$\text{pairs} = [(2, 3), (5, 4), (6, 7), (6, 4)].$$

After sorting, the task reduces to identifying the longest increasing subsequence in terms of the second component s. This is achieved by processing each element iteratively:

1. The first pair $(2, 3)$ is considered, so $seq = [3]$.
2. Next, the pair $(5, 4)$ is processed. Since $4 > 3$, it extends the subsequence: $seq = [3, 4]$.
3. The pair $(6, 7)$ is encountered. As $7 > 4$, it again extends the subsequence: $seq = [3, 4, 7]$.
4. Finally, the pair $(6, 4)$ is processed. In this case, the value $s = 4$ does not extend the sequence but instead replaces the second element (since it is the smallest valid replacement found by binary search). The sequence becomes $seq = [3, 4, 7]$, which remains unchanged in length.

At the conclusion of this process, the sequence *seq* has length three. Therefore, the algorithm returns 3 as the length of the longest subsequence of pairs in which both components increase strictly.

5.5 Longest Consecutive Pairs

The algorithm presented in Figure 5.4 is designed to compute the length of the longest consecutive subsequence of pairs. In this context, a subsequence is considered consecutive if, for each pair (f, s), the next element in the subsequence is exactly $(f + 1, s + 1)$. This definition imposes a stricter condition than simple ordering, as it requires both components of successive pairs to increase simultaneously by one unit.

Algorithm 8 LongestConsecutiveSubsequencePairs function()

Input: *pairs*: list of matching index pairs
Output: Length of the longest subsequence where each next element is $(f + 1, s + 1)$

```
1:  dp ← {}
2:  max_len ← 0
3:  pair_set ← set(pairs)                    // for quick access to existing pairs
4:  for (f, s) in pairs do
5:      prev ← (f − 1, s − 1)
6:      if prev ∈ pair_set then
7:          dp[(f, s)] ← dp[prev] + 1
8:      else
9:          dp[(f, s)] ← 1
10:     end if
11:     max_len ← max(max_len, dp[(f, s)])
12: end for
13: return max_len
```

Figure 5.4

The procedure begins by initializing a dynamic programming table dp, implemented as a dictionary, and a variable max_len to keep track of the longest subsequence length found during execution. Additionally, the entire set of input pairs is stored in $pair\_set$ to enable constant time membership checks. The algorithm then iterates over each pair (f, s) in the input. For each element, it computes its immediate predecessor, defined as $(f-1, s-1)$. If this predecessor exists in the set of pairs, it means that the current pair can extend a previously identified consecutive subsequence. In this case, the value of $dp[(f, s)]$ is assigned as $dp[prev] + 1$. Otherwise, if the predecessor is not present, the pair itself initiates a new subsequence of length one, and $dp[(f, s)]$ is set to 1. After updating the entry for the current pair, the algorithm compares the new subsequence length with the global maximum and updates max_len accordingly. Once all pairs have been processed, the value of max_len represents the length of the longest consecutive subsequence satisfying the strict $(f + 1, s + 1)$ condition.

From a computational perspective, the algorithm achieves efficiency by leveraging a hash set for predecessor lookup, reducing the membership check to $O(1)$ time on average. As a result, the entire procedure runs in linear time relative to the number of pairs, i.e. $O(n)$.

To clarify the operation of the algorithm, let us consider an illustrative example. Suppose that the input is given as the following list of index pairs:

$$\text{pairs} = [(1,1),(2,2),(3,3),(5,5),(6,6),(7,7),(9,9)].$$

The goal is to identify the longest subsequence in which each element is the immediate successor of the previous one, i.e. $(f+1, s+1)$. The algorithm begins by constructing a hash set containing all input pairs. It also initializes an empty dictionary dp and a variable $max\_len = 0$.

The pairs are then examined one by one:

1. Consider the pair $(1,1)$. Its predecessor $(0,0)$ is not present in the set. Therefore, $dp[(1,1)] = 1$. The maximum length is updated: $max\_len = 1$.

2. Next, the pair $(2,2)$ is processed. Its predecessor $(1,1)$ exists, and $dp[(1,1)] = 1$. Thus, $dp[(2,2)] = dp[(1,1)] + 1 = 2$. The maximum length becomes $max\_len = 2$.

3. For $(3,3)$, the predecessor $(2,2)$ is found, with $dp[(2,2)] = 2$. Hence, $dp[(3,3)] = 3$, and the maximum length increases to $max\_len = 3$.

4. The pair $(5,5)$ is then considered. Its predecessor $(4,4)$ is not in the set, so it initiates a new chain with $dp[(5,5)] = 1$. The maximum length remains unchanged at 3.

5. For $(6,6)$, the predecessor $(5,5)$ is present, and $dp[(5,5)] = 1$. Therefore, $dp[(6,6)] = 2$. The maximum length remains 3.

6. Next, $(7,7)$ extends the chain further, as its predecessor $(6,6)$ is present with $dp[(6,6)] = 2$. Thus, $dp[(7,7)] = 3$. Again, the global maximum remains 3.

7. Finally, the pair $(9,9)$ is evaluated. Its predecessor $(8,8)$ is not present in the set, meaning it forms a new subsequence of length one: $dp[(9,9)] = 1$. The maximum length is not affected.

At the end of the process, the algorithm concludes that the longest consecutive subsequence of pairs has length three. This occurs in two distinct chains:
$(1,1) \to (2,2) \to (3,3)$ and $(5,5) \to (6,6) \to (7,7)$.

This example illustrates the fundamental mechanism of the algorithm: it dynamically builds the length of consecutive subsequences by referencing the presence of immediate predecessors, while simultaneously maintaining the global maximum length. The approach is efficient and ensures that all possible consecutive sequences are taken into account.

5.6 Relation to the Previous Approach

The Mapping-based strategy described in this chapter follows the same conceptual idea introduced in the LCS-based approach. The overall workflow, from the identification of seeds, through the generation of candidate mappings via column combinations, to the verification of their effective overlap, remains structurally identical. The main difference lies in the representation of correspondences and in the algorithms used to enforce row constraints. Whereas the first approach operated directly on the table columns and relied on the LCS and LCStr algorithms, the current formulation replaces them with the *Longest Increasing Pairs* and *Longest Consecutive Pairs* algorithms, respectively. These new algorithms operate on the mapping structure rather than on raw column values, yet they serve the same purpose: ensuring that overlaps respect ordering and contiguity constraints, respectively. As a result, once the global mapping is constructed, the remainder of the framework proceeds in the same way as before. To identify the largest overlap, multiple column mappings are combined and analyzed according to the same logical criteria defined in the initial approach.

From a practical standpoint, the *gen_cand*() function remains nearly identical to the one used in the LCS-based method (Figure 4.2 of Chapter 4), since column constraints are still enforced by simply checking the ordering of column indices. The only substantial changes occur in the *ver_cand*() function. In the original LCS-based formulation, row constraints were enforced by applying the LCS or LCStr algorithms directly to the column values. In the mapping-based strategy, by contrast, the overlap is first computed as the intersection of the tuple pair mappings, and the row constraints are then enforced through the Longest Increasing Pairs or Longest Consecutive Pairs algorithms. Consequently, while the overall workflow remains unchanged, the portion of *ver_cand*() responsible for verifying row constraints is slightly different, as shown in Figure 5.5.

Algorithm 3 verCand() function

Input: The two tables $R(X)$ and $S(Y)$, the top candidate $topC$, the priority queues for the candidates $Candidates$, the selected columns pairs col_pairs
Output: The updated priority queues

1: $map\_intersection \leftarrow \bigcap_{S \in col\_pairs} S$
2: **if** $row\_constraint == 'none'$ **then**
3: $\quad topC.O \leftarrow \textbf{\textit{compute\_BagIntersection}}(R[X_{topC.M}], S[Y_{topC.M}])$
4: **end if**
5: **if** $row\_constraint == 'ordered'$ **then**
6: $\quad topC.O \leftarrow \textbf{\textit{LongestIncreasingSubsequencePairs}}(map\_intersection)$
7: **end if**
8: **if** $row\_constraint == 'contiguous'$ **then**
9: $\quad topC.O \leftarrow \textbf{\textit{LongestConsecutiveSubsequencePairs}}(map\_intersection)$
10: **end if**
11: $topC.h \leftarrow |topC.O|$ ▷ actual height
12: $topC.A \leftarrow topC.w \cdot topC.h$ ▷ actual area
13: **if** $topC.A \geq \theta$ **then**
14: $\quad \theta \leftarrow topC.A$
15: $\quad Candidates.push(topC)$ ▷ reinsert candidate
16: $\quad$ **for all** $cand \in Candidates$ **do**
17: $\quad\quad$ **if** $cand.A < \theta$ **then**
18: $\quad\quad\quad Candidates.delete(cand)$ ▷ prune candidate
19: $\quad\quad$ **end if**
20: $\quad$ **end for**
21: **end if**
22: **return** $Candidates$

Figure 5.5

Chapter 6

Comparison of the Two Approaches

The two approaches to identify the largest overlap between two tables differ both in methodological assumptions and in practical performance. The **LCS/LCStr-based approach** operates directly on the columns of the two tables without the need to construct an intermediate mapping. This feature reduces preprocessing overhead and makes the method straightforward to apply. Moreover, it is inherently robust to the presence of duplicated elements: even if the two tables share a large number of repeated values, the algorithms can still process them without a significant increase in structural complexity. However, these same characteristics can also be seen as limitations. The absence of an explicit construction of a mapping restricts the flexibility of the framework and we lost its modularity, as it becomes difficult to incorporate additional constraints.

In fact, the **mapping-based approach** explicitly decouples the discovery of correspondences from the computation of the largest overlap. This modular design introduces several advantages. First, it improves **flexibility**, since the same mapping can support different algorithms and constraint sets. Second, it improves **efficiency** in cases where only a small number of values are shared between the tables, as the mapping is considerably smaller than the Cartesian product of rows. Third, it supports **extensibility**, allowing for the seamless integration of weighted similarities, attribute specific priorities, or other domain specific refinements. The main drawback of this approach arises in the presence of many shared duplicated values: in such cases, the mapping construction generates a large number of index pairs, which may increase preprocessing costs. This last aspect related to shared duplicate values will be discussed in detail in the next section to better understand what it is and how to manage it when it occurs.

6.1 Impact of Duplicated Values

The computational behavior of the two approaches is strongly influenced by the presence and distribution of duplicated values within the compared tables. In particular, when a large number of duplicates appear in both tables, the Mapping-based approach tends to suffer from a combinatorial explosion in the number of correspondences. Since each occurrence of a duplicated value in one table must be matched with all corresponding occurrences in the other, the mapping stage produces a dense set of index pairs. Formally, if a value appears k_1 times in the first table and k_2 times in the second, the mapping will contain $k_1 \times k_2$ entries for that value alone. When many of such values exist, this quadratic growth in mapping size substantially increases preprocessing and memory requirements, potentially overshadowing the benefits of modularity and flexibility. In contrast, in scenarios where the tables contain few duplicated elements, or where the overlap is dominated by unique or sparsely repeated values, the mapping remains compact. Under these conditions, the mapping-based approach becomes significantly more efficient than the LCS/LCStr-based alternative, as it avoids the need to consider all possible column values. The reduced mapping size translates into faster computation of the largest overlap. Hence, the efficiency of the mapping-based method is highly sensitive to data redundancy: it performs optimally when duplication is limited, but may degrade soon in the presence of extensive repetition across both tables.

6.2 Empirical Illustration

To illustrate the practical implications of duplicated values, we consider two contrasting scenarios. In both cases, the two tables share a comparable number of values; however, the distribution of these values, specifically their degree of duplication, differs substantially.

Case 1: High Duplication

Let the first table T_1 and the second table T_2 each contain 1,000 rows, with only 10 distinct values repeated many times. For example, both tables could include the values A and B, respectively 200 times and 150 times. When constructing the mapping, every occurrence of a value in T_1 must be paired with each matching occurrence in T_2. For the value A alone, this results in $200 \times 200 = 40,000$ index pairs. Extending this to all duplicated values quickly yields a mapping of hundreds of thousands of pairs, far exceeding the original table sizes. In this scenario, the mapping-based approach incurs a substantial computational cost during preprocessing. Both the time and memory required to store and process the mapping grow quadratically with the frequency of duplicated values. In contrast, the LCS/LCStr-based approach handles repeated values directly within its dynamic programming formulation, without enumerating all pairwise correspondences. Although the overall complexity of the LCS algorithm remains $O(nm)$, its computational complexity is considerably lower in this highly redundant case.

Case 2: Low Duplication, High Overlap

Now consider the opposite situation: both tables still contain 1,000 rows, but most values are unique or appear only a few times. Suppose 600 distinct values are shared between T_1 and T_2, each occurring once or twice at most. The resulting mapping now contains approximately 600 to 1,200 pairs, orders of magnitude smaller than in the previous case. Here, the mapping-based approach becomes clearly advantageous. Its compact mapping can be constructed rapidly, and the subsequent overlap computation operates only on relevant correspondences rather than on the full cross-product of table rows. Meanwhile, the LCS/LCStr-based approach must still process the entire column space, regardless of how sparse the overlap is, leading to unnecessary comparisons and slower execution. These contrasting cases demonstrate that the efficiency of the mapping-based method is critically dependent on the structure of the data. When many duplicated values are shared across the tables, the mapping construction becomes the dominant cost. However, when duplicates are rare and overlaps are concentrated among distinct elements, the mapping-based approach achieves superior computational performance, outperforming the LCS/LCStr-based method in both time and scalability.

6.3 Role of Shared Duplicates in Computational Growth

It is important to emphasize that the computational overhead described above does not depend entirely on the presence of duplicated values within each table, but rather on whether these duplicates are **shared** across both tables. The mapping-based approach generates a correspondence only for values that appear in both T_1 and T_2. Therefore, duplicates that occur exclusively within one table have no impact on mapping size.

Formally, let $D_1(v)$ and $D_2(v)$ denote the number of occurrences of a value v in T_1 and T_2, respectively. The number of mapping pairs contributed by v is given by $D_1(v) \times D_2(v)$. If v appears only in one table (e.g. $D_1(v) > 0$ but $D_2(v) = 0$), then no pairs are generated for that value. Thus, the total mapping size can be expressed as

$$|M| = \sum_{v \in T_1 \cap T_2} D_1(v) \times D_2(v)$$

This formulation highlights that the growth of the mapping is driven exclusively by the subset of values that are duplicated and shared between the two tables. Consequently, two datasets may each contain a large number of repeated values without causing any computational issue, provided that these repetitions do not overlap in content.

From a practical perspective, this distinction explains why the mapping-based ap-

proach may still perform efficiently in domains characterized by local redundancy (e.g. repeated categorical entries within each dataset) but limited cross-table duplication.

Chapter 7

Experimental Evaluation

This chapter presents the experimental assessment of the two proposed methods for detecting the largest overlap under constraints. The objective is to evaluate its accuracy, efficiency, and scalability. First, the datasets employed for testing are introduced, followed by the experimental setup and the evaluation metrics used. Finally, the obtained results are presented and discussed, highlighting the main findings.

7.1 Datasets

In order to evaluate the performance, experiments were conducted on two tabular datasets extracted from real world sources: Wikipedia and GitHub.

WikiTables is a collection of tables extracted from Wikipedia pages covering a wide range of domains such as sports, geography, and history. These tables are semi-structured and generally small to medium in size. They vary significantly in layout and content, with heterogeneous data types, irregular column structures, and occasional missing or duplicated values.

GitTables, on the other hand, consists of tables extracted from public GitHub repositories, often originating from CSV files. These tables tend to be more structured and domain-specific, typically featuring consistent column headers and predominantly numeric or categorical data. Unlike Wikipedia tables, GitTables often includes very large tables, with thousands of rows and dozens of columns.

Table 7.1 reports the statistics of the two datasets. WikiTables includes 19754 tables, while GitTables contains 47634 tables, more than twice. In terms of table size, there are substantial differences between the two datasets. Tables from Wikipedia have, on average, 10.21 rows and 8.07 columns, whereas tables from GitHub present much larger tables, with an average of 151.33 rows and 13.08 columns. The maximum table dimensions confirm this pattern: WikiTables tables reach up to 255 rows and 125

columns, while in GitTables they can have as many as 6013 rows and 592 columns. In this context, WikiTables provides smaller table, whereas GitTables poses a greater computational challenge due to its larger tables, making it suitable for evaluating the scalability of the framework.

Dataset		**# cols**	**# rows**	**# distinct**	**# missing**
WikiTables	mean	8.07	10.21	34.52	5.89
	min	1.00	1.00	1.00	0.00
	max	125	255	1765	1463
GitTables	mean	13.08	141.23	421.30	118.56
	min	1.00	1.00	1.00	0.00
	max	592	6 013	97 538	96 318

Table 7.1: Summary statistics for 19 754 WikiTables and 47 634 GitTables.

7.2 Experimental Setup

All experiments were conducted on a MacBook Air equipped with a *1.1 GHz Quad-Core Intel Core i5* processor, *8 GB of 3733 MHz LPDDR4X* memory, and running macOS *Sequoia 15.7.1*. The implementation was developed in *Python 3.11*, using *Pandas* and *NumPy* as libraries for data processing and numerical computation.

The implemented framework accepts two tables as input and returns their largest overlap that is the largest common subtable between them, computed according to the specified constraints on rows and columns. Consequently, the two datasets in their original form, each containing individual tables, are not directly suitable for the experimental phase. To evaluate the framework, it was necessary to generate pairs of tables from the available data. For this purpose, 60000 pairs of tables were randomly generated from WikiTables and 100000 pairs from GitTables. However, due to computational resource limitations, experiments were performed on a subset of these pairs: 10000 pairs for WikiTables and 5000 pairs for GitTables. This choice reflects the difference in table size across datasets. Wikipedia tables are generally smaller and less computationally demanding, while GitTables are typically larger and more expensive to process when computing the largest overlap.

The experiments were conducted under two constraint configurations, ordering constraint and contiguity constraint, each applied both to rows and to columns for the two approaches. For each dataset and configuration, both approaches were executed separately to assess their accuracy and computational efficiency. The results were collected in a structured *DataFrame* containing information such as the number of generated and verified candidates, the number of seeds, the overlap percentage $a\%$, and the execution time. Among these, execution time and overlap percentage, defined as the ratio between the detected overlap area and the size of the smaller table, were considered the primary evaluation metrics.

7.3 Results

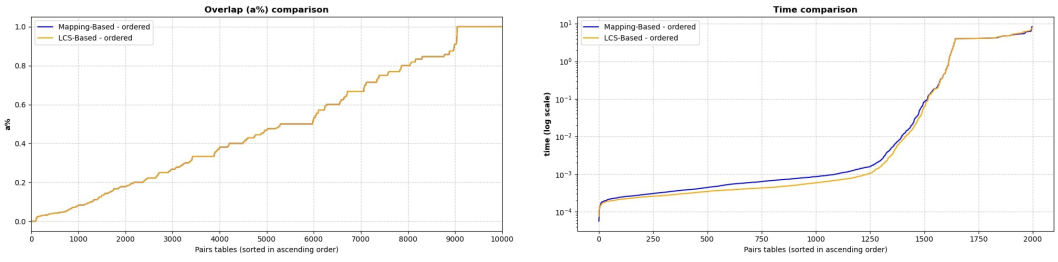
This section presents and discusses the results obtained from the tests executed. The analysis compares the LCS-based and mapping-based approaches under different combinations of row and column constraints, across the WikiTables and GitTables datasets. The discussion is structured by dataset and type of constraints, focusing on the overlap percentage $a\%$ and the execution time.

7.3.1 WikiTables

Ordering constraint

Figures 7.1a and 7.1b report the results obtained on the WikiTables dataset under the ordering constraint, where both rows and columns are required to preserve their relative order across the two tables. Figure 7.1a shows the comparison of the overlap percentage $a\%$ between the LCS-based and mapping-based methods. As can be observed, the two curves perfectly coincide, indicating that both methods consistently return the same overlap values for all table pairs. This outcome confirms that the two approaches solve the same underlying problem and produce equivalent results in terms of largest overlap found, despite relying on different computational strategies. In contrast, 7.1b compares the execution time of the two methods. The results reveal that the performances are very similar, with almost overlapping curves across all tested pairs. The small differences observed are within the expected margin of variation and confirm that, for this configuration, both implementations achieve comparable efficiency.

These observations are supported by the statistical summary reported in Table 7.2, which presents aggregate measures for both time and overlap. The mean execution times are nearly identical, (0.886 s) for the LCS-based method and (0.873 s) for the mapping-based one, while the overlap statistics are perfectly aligned.



(a) **(b)**

Figure 7.1

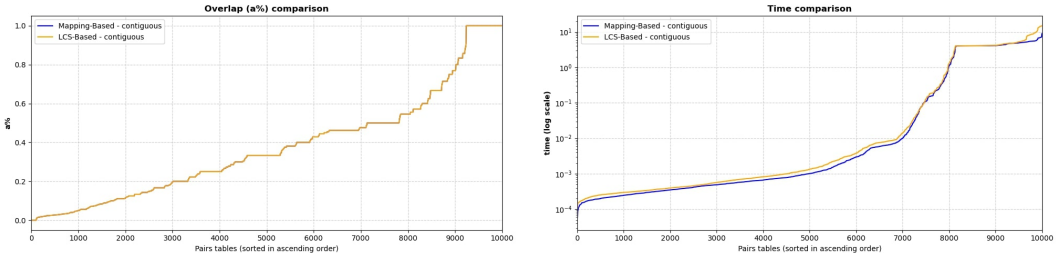
Statistics Time (s)	**LCS-Based**	**Mapping**
Min	0.000	0.000
Max	7.616	8.361
Mean	**0.886**	**0.873**
Std dev	1.835	1.799
Statistics Overlap (a%)	**LCS-Based**	**Mapping**
Mean	0.482	0.482
Std dev	0.302	0.302

Table 7.2

Contiguity constraint

Figures 7.2a and 7.2b illustrate the results obtained on the WikiTables dataset under the contiguity constraint, where both rows and columns are required to be aligned in consecutive positions. As expected, the overall overlap percentages decrease compared to the ordering case, since stricter alignment conditions reduce the number of matching regions between tables. In Figure 7.2a, it can be observed that the LCS-based and mapping-based approaches produce identical overlap values for all table pairs, confirming once again that the two methods compute the same largest overlap. This validates the correctness of the mapping-based formulation with respect to the original framework. Figure 7.2b reports the corresponding execution times, showing that the mapping-based approach generally achieves slightly lower values across most table pairs. The difference is more evident for the most computationally demanding cases, where the mapping representation helps to reduce redundant comparisons and improve overall scalability.

The statistical summary in Table 7.3 supports these observations. The mean execution time decreases from (1.077 s) for the LCS-based method to (0.905 s) for the mapping-based one, while the standard deviation also drops from (2.401 s) to (1.824 s), indicating more stable performance. The overlap statistics remain perfectly identical, with an average $a\%$ of 0.374 and a standard deviation of 0.272 for both methods.



(a) **(b)**

Figure 7.2

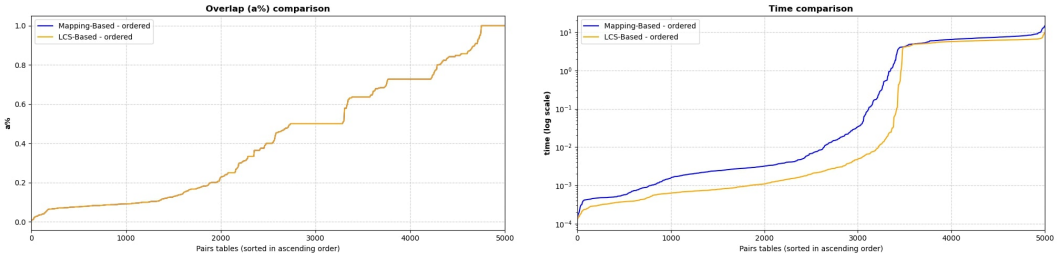
Statistics Time (s)	**LCS-Based**	**Mapping**
Min	0.000	0.000
Max	14.976	9.044
Mean	**1.077**	**0.905**
Std dev	2.401	1.824
Statistics Overlap (a%)	**LCS-Based**	**Mapping**
Mean	0.374	0.374
Std dev	0.272	0.272

Table 7.3

7.3.2 GitTables

Ordering constraint

Figures 7.3a and 7.3b present the results obtained on the GitTables dataset under the ordering constraint. As shown in Figure 7.3a, the overlap percentage $a\%$ produced by the LCS-based and mapping-based approaches is identical across all table pairs, confirming that both methods yield equivalent results in terms of overlap accuracy. However, Figure 7.3b reveals a noticeable difference in execution times. While the LCS-based approach tends to perform faster on smaller table pairs, the mapping-based method exhibits higher variability and slightly longer average times, particularly for larger and more complex tables. This behavior is also reflected in Table 7.4, where the mean execution time increases from (1.799 s) for the LCS-based to (2.223 s) for the mapping-based, with a corresponding rise in standard deviation. Despite this difference in runtime, both methods achieve the same average overlap (0.416), indicating complete consistency in the detected results. These findings suggest that, for larger datasets such as GitTables, the mapping-based formulation preserves accuracy but may introduce a modest computational overhead due to the increased complexity of mapping generation.



(a) **(b)**

Figure 7.3

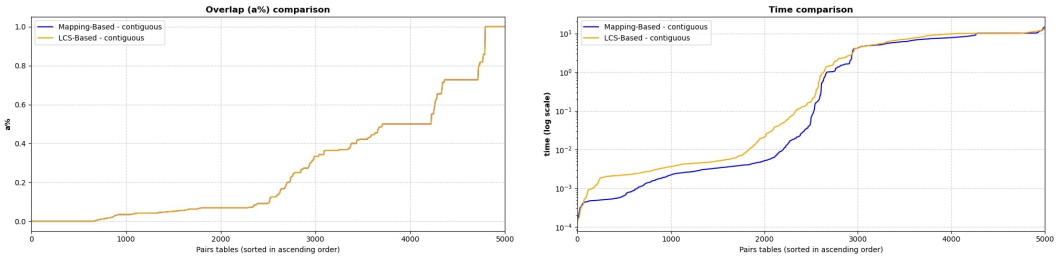
Statistics Time (s)	**LCS-Based**	**Mapping**
Min	0.000	0.000
Max	9.836	14.829
Mean	**1.799**	**2.223**
Std dev	2.728	3.317
Statistics Overlap (a%)	**LCS-Based**	**Mapping**
Mean	0.416	0.416
Std dev	0.305	0.305

Table 7.4

Contiguity constraint

Figures 7.4a and 7.4b display the results for the GitTables dataset under the contiguity constraint. As expected, the stricter alignment condition results in lower overlap percentages compared to the ordering configuration, since fewer matching regions can satisfy the contiguity requirement. In Figure 7.4a, the overlap ($a\%$) of the LCS-based and mapping-based approaches perfectly coincide, confirming that both methods identify the same overlapping regions and thus achieve equivalent accuracy. Figure 7.4b shows that the mapping-based approach achieves slightly lower execution times on average, particularly for larger and more complex pairs, demonstrating better scalability under this constraint.

The summary statistics in Table 7.5 confirm these findings: the mean execution time decreases from (3.673 s) for the LCS-based to (3.355 s) for the mapping-based, while the overlap percentage remains identical for both methods (0.272). Also in this case, the mapping-based approach preserves the same accuracy as the LCS-based, while providing a modest but consistent improvement in computational efficiency under the contiguity constraint.



(a) **(b)**

Figure 7.4

Statistics Time (s)	**LCS-Based**	**Mapping**
Min	0.000	0.000
Max	14.344	14.910
Mean	**3.673**	**3.355**
Std dev	4.352	4.094
Statistics Overlap (a%)	**LCS-Based**	**Mapping**
Mean	0.272	0.272
Std dev	0.286	0.286

Table 7.5

7.4 Summary

The experimental results show that the mapping-based and LCS-based approaches yield identical overlap values across all configurations, confirming the correctness and consistency of both methods. Performance differences arise with respect to computational efficiency. On smaller tables, the mapping-based approach performs similarly, or slightly better, than the LCS-based one, benefiting from reduced redundancy in the comparison process. However, on larger tables, especially those containing many duplicate values, the mapping representation becomes computationally demanding as the number of possible correspondences grows rapidly. These findings suggest that the advantages of the mapping-based method are most evident in scenarios with smaller, well-structured tables, while the LCS-based formulation remains more stable at scale.

Chapter 8

Final Framework

As mentioned in the previous chapters, the computational efficiency of the two strategies depends strongly on the content of the input tables. In particular, the mapping-based approach exhibits superior performance when the tables share a limited number of duplicated values, while the LCS-based method demonstrates more stable performance in the presence of duplicated values. However the first, on the other hand, can be applied effectively to larger tables, where its design offers better scalability. Instead, the LCS based requires that the computational overhead remains limited. These complementary behaviors suggest that no single approach is universally optimal. The most effective solution is to integrate both methods into a hybrid framework that automatically selects the most suitable strategy based on the characteristics of the data. In the next sections, we will describe how to analyze the distribution of duplicated values to determine which approach is more appropriate for a given pair of tables. Finally, we present the resulting framework and evaluate its performance against the two individual methods.

8.1 Preliminary Analysis for Method Selection

Before applying either the mapping-based or the LCS-based approach, it is possible to perform a lightweight statistical analysis of the two tables. The goal of this step is to determine whether the two tables exhibit a high degree of shared duplication, that is repeated values that occur in both tables, which would make the mapping-based approach computationally expensive. The procedure to follow is described below:

1. Compute Basic Frequency Distributions

For each input table T_1 and T_2, compute the frequency of each distinct value:

$$f_1(v) = \text{count of } v \text{ in } T_1, \quad f_2(v) = \text{count of } v \text{ in } T_2$$

2. Identify Shared Values

Determine the intersection of the two value sets:

$$V_{shared} = v \mid v \in T_1 \text{ and } v \in T_2$$

Only values in V_{shared} can contribute to the size of the mapping, so the analysis can focus on this subset.

3. Estimate Mapping Density

Estimate the potential size of the mapping as follows:

$$S_{est} = \sum_{v \in V_{shared}} f_1(v) \times f_2(v)$$

This quantity grows quadratically with the frequency of shared duplicates.

4. Define a Heuristic Threshold

To choose between approaches, define a **global duplication index** to predict the expected computational cost of the mapping-based approach.

$$R = \frac{S_{est}}{|T_1| \times |T_2|}$$

where S_{est} represents the estimated number of mapping pairs (i.e. potential correspondences) and $|T_1|, |T_2|$ denote the total number of values in the two tables, respectively. Once computed, R can be compared against a predefined heuristic threshold to determine the most suitable strategy:

- when R is **low**, the mapping-based approach is expected to be more efficient, since the number of shared duplicates, and therefore the mapping size, is limited
- when R is **high** the LCS-based approach becomes preferable, as the mapping construction would otherwise be computationally expensive.

Algorithm 1 estimate_mapping_density function()

Input: Two dataframes df_1, df_2; *threshold_dup*
Output: global duplication index R

```
/* Flatten all columns into single value lists */
T1 ← df1.values.flatten()
T2 ← df1.values.flatten()
/* Compute frequency distributions */
f1 ← Counter(T1)
f2 ← Counter(T2)
/* Determine shared values */
shared ← set(f1.keys()) & set(f2.keys())
/* Estimate mapping size */
S_est ← sum(f1[v] * f2[v] for v in shared)
total_pairs ← |T1| × |T2|
if total_pairs > 0 then
    R ← S_est/total_pairs
else
    R ← 0
end if
if R < threshold_dup then
    recommendation ← Mapping-based approach
else
    recommendation ← LCS-based approach
end if
return R
```

Figure 8.1

The *estimate_ mapping_ density* function has been implemented following the procedure described above and it is shown in Figure 8.1.

8.1.1 Approximation of the Mapping Density

It should be noted that the global duplication index R computed over the entire tables provides only an approximate estimation of the actual mapping density. In the mapping-based approach, correspondences are typically established between pairs of columns, each mapping being defined as a set of index pairs (i, j) such that the values in the selected columns coincide. The global analysis instead aggregates all values across the selected columns of each table before computing the frequency distributions $f_1(v)$ and $f_2(v)$. As a result, the measure R does not reflect the true per-column structure of the mapping, but rather provides a global estimate of the overall degree of shared duplication between the two datasets. This simplification has both advantages and limitations:

- **Advantages**: It allows a very fast estimate of the mapping density, without inspecting every possible columnn pair.

- **Limitations**: Because the mapping is actually constructed column-wise, the global index may *overestimate or underestimate* the true mapping density, depending on how duplicates are distributed across columns. For example, if duplicates are concentrated in a single column, the global estimate may exaggerate the mapping size; conversely, if they are dispersed across different columns, the true mapping density may be lower than suggested by the global measure.

For these reasons, the global duplication index should be interpreted as a heuristic indicator rather than an exact indicator. It effectively captures the expected computational trend but not the exact cost of constructing column-level mappings.

8.1.2 Correlation between Mapping Density Ratio and Execution Time

Figure 8.2 illustrates the relationship between the Global Duplication Index, also referred to as the Mapping Density Ratio (R), and the execution time of the mapping-based method. Each point represents a pair of tables used in the evaluation. The red regression line indicates the general trend in the data, while the shaded region represents the 95% confidence interval. As the figure illustrates, there is a positive correlation between R and execution time (Pearson $r = 0.45$), meaning that higher mapping density ratios are associated with longer computation times. This confirms the expected computational behavior: as the proportion of shared duplicated values between the two tables increases, the number of potential mappings grows quadratically, resulting in a substantial increase in processing cost. Performing this analysis on a dataset of 1000 table pairs collected from GitHub, a consistent relationship emerged between the Global Duplication Index (R) and the pairs for which the mapping-based method failed to complete. In these cases, the algorithm required excessive computation time to determine the largest overlap. Empirically, this critical value was found to be approximately $R \approx 0.1$. Hence, this empirical threshold can be used as a decision boundary for method selection:

- when $R < 0.1$, the mapping-based approach remains efficient, since the estimated mapping size is limited and the number of shared duplicate values is relatively small
- when $R \geq 0.1$, the computational cost becomes prohibitive, and the LCS-based method should be preferred.

This analysis therefore not only quantifies the correlation between duplication density and computational cost but also provides a practical heuristic criterion for automatically selecting the most suitable approach depending on the structural characteristics of the input tables.

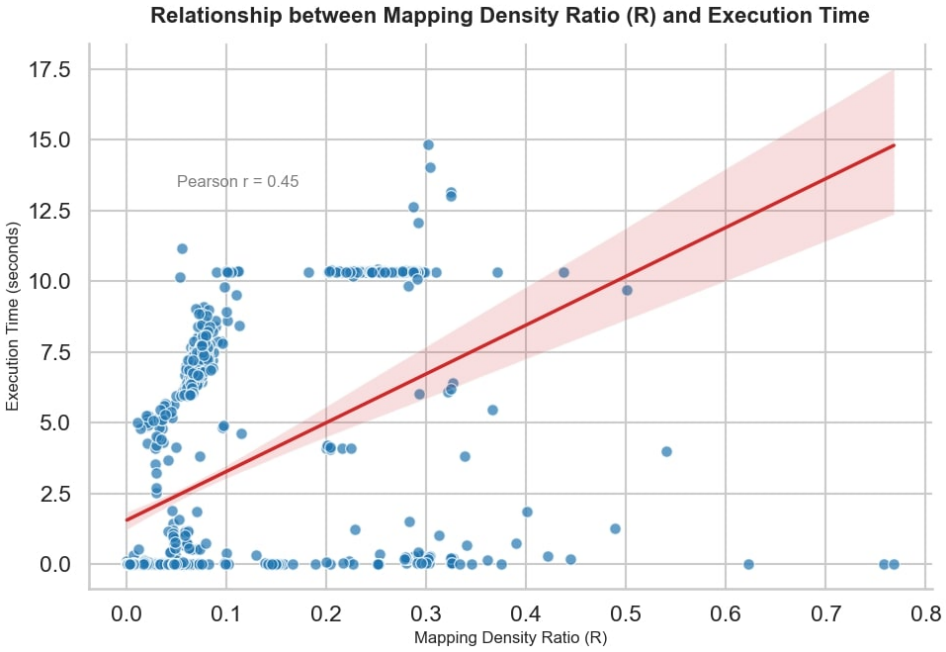


Figure 8.2

8.2 Framework Overview

The proposed **hybrid framework** integrates both approaches within a unified processing pipeline. The algorithm shown in Figure 8.3 summarizes how the proposed hybrid framework works. It first estimates the mapping density between the two input tables to quantify the degree of value duplication. Based on this ratio (R), the system dynamically selects the most appropriate processing strategy. When R exceeds the predefined duplication threshold, the mapping-based approach is adopted, as it offers higher efficiency on large and redundant datasets. Conversely, when duplication is limited, the framework switches to the LCS-based method. After the selected algorithm has completed, the framework retrieves the execution time and the maximum overlap value from the output metrics. The overlap percentage $a\%$ is then computed by normalizing the detected overlap with respect to the smallest of the two table dimensions. Subsequently, the function *reconstruct_largest_overlap()* is invoked to rebuild the actual largest overlapping region between the input tables. Finally, the system prints the reconstructed overlap, its relative area in percentage, and the corresponding execution time.

Figure 8.4 illustrates the pseudocode of the *reconstruct_largest_overlap* function, responsible for rebuilding the portion of data that corresponds to the largest overlap identified by the hybrid framework. The algorithm first extracts and sorts the column pairs involved in the best mapping result, then reconstructs a subtable using only the relevant columns from the input table *r_tab*. Depending on the row constraint, it applies either the *Longest Common Subsequence* or the *Longest Common Substring* procedure to identify the matching row indices. Finally, it assembles and returns the reconstructed overlap as a DataFrame, representing the largest overlap between the

two input tables.

Algorithm 2 final_framework main()

Input: Two tables *r_tab*, *s_tab*; *threshold_dup* = 0.1
Output: largest overlap

```
R = estimate_mapping_density(r_tab, s_tab)
if R < threshold_dup then
    results, metrics = Mapping_based(r_tab, s_tab, row_constraint, col_constraint)
else
    results, metrics = LCS_based(r_tab, s_tab, row_constraint, col_constraint)
end if
time = metrics[1]
max_overlap = metrics[12]
if max_overlap then
    a% = max_overlap / min(r_a, s_a)
else
    a_perc = None
end if
df = reconstruct_largest_overlap(r_tab, s_tab, seeds, results, row_constraint)
print('The largest overlap found out is :')
print(df)
print('The largest overlap area (%) is :', a_perc)
print('in seconds :', time)
```

Figure 8.3

Algorithm 1 reconstruct_largest_overlap function()

Input: Two tables *r_tab*, *s_tab*; list of *seeds*; list of *results*; *row_constraint*
Output: DataFrame representing the reconstructed largest overlap

```
col_pairs ← [seeds[s_id][0] for each s_id ∈ results[0]]
col_pairs.sort(key = lambda s : s[0], reverse = False)
valid_cols ← [ r_tab[ m[0] ]  for each  m ∈ col_pairs ]
data_dict ← {idx[0] : values | (idx, values) ∈ zip(col_pairs, valid_cols)}
if row_constraint = 'ordered' then
    idx_list ← compute_long_comm_subseq(r_tab, s_tab, results[0], seeds)
    df ← DataFrame(data_dict)[idx_list]
else
    h, start_idx ← compute_long_comm_substr(r_tab, s_tab, results[0], seeds)
    end_idx ← start_idx + h − 1
    df ← DataFrame(data_dict)[start_idx : end_idx]
end if
return df
```

Figure 8.4

8.3 Experimental Validation

To evaluate the performance of the proposed Final Framework, a comparative analysis was carried out against the two baseline algorithms and the integrated version combining both approaches. The assessment focused on computational time, measured on a dataset of 5000 pair tables from GitHub. As shown in Figure 8.5, the proposed Final Framework consistently demonstrates lower computation times compared to the Mapping-Based approach and performs comparably or better than the LCS-Based approach across the majority of test cases. The general trend indicates that the integration of both strategies within the final framework effectively mitigates the performance drawbacks observed when each algorithm is used in isolation.

The quantitative results in Table 8.1 further confirm this observation. The mean computation time for the Final Framework (2.117 s) is the lowest among the three approaches, improving upon both the LCS-Based (2.558 s) and Mapping-Based (3.310 s) methods. Similarly, the standard deviation (3.46 s) remains moderate, indicating stable and consistent performance. Although the maximum recorded time (13.066 s) is slightly higher than that of the LCS-Based approach, this value remains well below the Mapping-Based maximum (14.829 s), confirming the robustness of the framework under varying conditions. These results validate the efficiency of the proposed framework, which successfully combines the strengths of the LCS and Mapping-Based algorithms while minimizing their respective limitations.

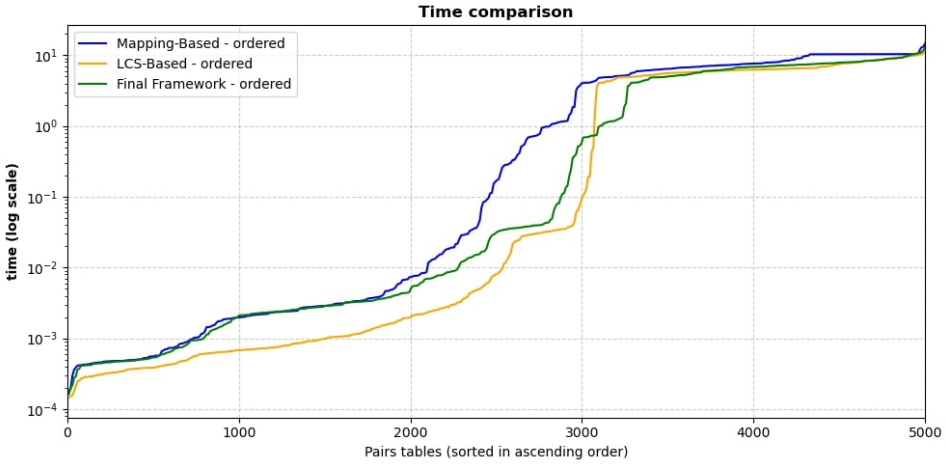


Figure 8.5

Statistics Time (s)	LCS-Based	Mapping-Based	Final Framework
Min	0.000	0.000	0.000
Max	11.155	14.829	13.066
Mean	**2.558**	**3.310**	**2.117**
Std dev	3.35	4.091	3.46

Table 8.1

Finally, we compare the performance of our final framework with the baseline method proposed in the literature, SLOTH [9], which computes the largest overlap between two tables without imposing any ordering constraints. In Figure 8.6, we present the results in terms of overlap percentage. As expected, SLOTH consistently achieves higher values, since it does not consider ordering, whereas our final framework incorporates ordering constraints that make the matching process more restrictive.

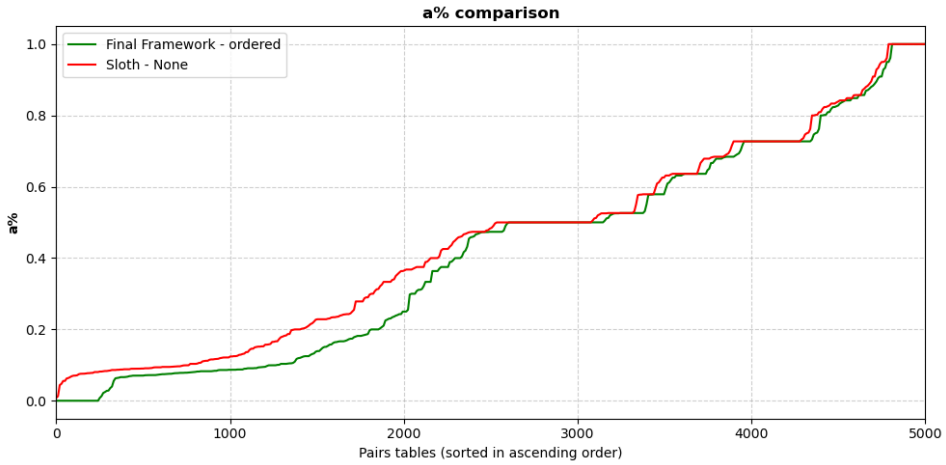


Figure 8.6

We also compare the execution time. As shown in Figure 8.7, the two curves exhibit very similar trends, indicating that the performance of our approach is essentially on the same level as SLOTH. This result is particularly significant because our framework incorporates ordering constraints that make the overlap computation more restrictive, yet it does not incur any noticeable loss in efficiency. A quantitative analysis further supports this observation, table 8.2. Our framework achieves a maximum of (13.066 s), with an average of (2.517 s) and a standard deviation of 3.46. SLOTH shows comparable maximum values, (12.647 s), but with a higher mean execution time of (3.435 s) and a larger variability (standard deviation of 4.487). These results demonstrate that the framework we developed is both efficient and robust.

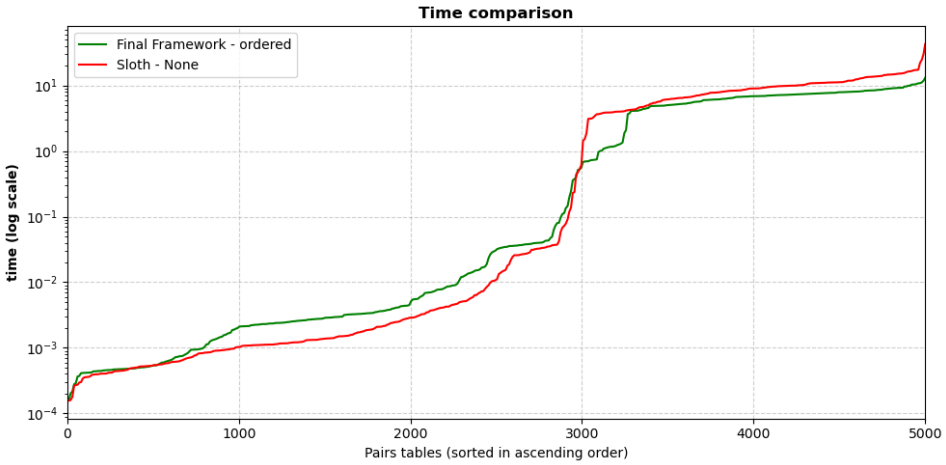


Figure 8.7

Statistics Time (s)	**SLOTH**	**Final Framework**
Min	0.000	0.000
Max	12.647	13.066
Mean	**3.435**	**2.117**
Std dev	4.48	3.46

Table 8.2

Chapter 9

Conclusion and Future Works

The exponential growth of structured data on the Web and within large scale repositories such as data lakes has led to the creation of millions of relational tables, often containing overlapping or duplicate information. Wikipedia provides a clear example: tables evolve over time, are frequently edited, and often get duplicated or modified through copy-and-paste, leading to multiple coexisting versions. In data lakes, a similar phenomenon occurs as new datasets are continuously ingested and updated. In these heterogeneous and dynamic environments, detecting similar or partially overlapping tables becomes crucial for tasks such as consistency checking, deduplication, and version tracking. This thesis addresses the problem of identifying the largest overlap between two tables under two structural constraints. Unlike existing methods that allow arbitrary reordering of rows and columns, the proposed framework enforces ordering and contiguity constraints, ensuring that structural and semantic relationships within tables are preserved. This enables the detection of structurally consistent duplicates, that is, tables that have evolved or expanded over time while maintaining their original organization. To tackle this problem, two complementary approaches were developed. The first, the LCS-based approach, adapts classical sequence algorithms to compute overlap directly over table rows and columns. The second, the Mapping-based approach, introduces an alternative formulation based on an explicit mapping of common elements between tables. This representation allows the overlap to be detected through the Longest Increasing Pairs and Longest Consecutive Pairs algorithms, which have been designed and implemented specifically for this work. They operate on index mappings rather than on the original data values.

The experimental evaluation on two real world datasets, WikiTables and GitTables, demonstrated that both approaches return identical overlap results, confirming the correctness and equivalence of their formulations. Performance analysis revealed that the Mapping-based approach can achieve comparable or slightly better efficiency on smaller and moderately sized tables, benefiting from its modular structure and reduced redundancy. However, results also showed that on larger tables with high duplication rates, the Mapping-based method tends to be computationally heavier,

as the number of pairwise correspondences grows quadratically with the number of duplicates. In general, this research provides a flexible framework for largest overlap detection that unifies multiple constraint settings (none, ordering, and contiguity) within a single approach. The framework adopts a hybrid strategy that automatically switches between the LCS-based and Mapping-based methods, depending on input characteristics such as duplication density. This design enables robust and efficient overlap detection across a wide range of table configurations.

The proposed framework has been designed with extensibility in mind, providing a solid foundation for further research in large-scale table analysis and data integration. Several directions can be explored to extend this work. A first line of improvement concerns algorithmic optimization. Reducing mapping redundancy and refining the search space for candidate correspondences could significantly improve efficiency, especially in scenarios involving high duplication or large scale datasets. The introduction of approximation techniques or index-based pruning strategies could further reduce computational costs without compromising accuracy. A second promising direction is parallelization and scalability. Implementing the algorithms within distributed computing environments or leveraging GPU computation could make the framework suitable for massive data lakes.

Dedication

Desidero infine rivolgere qualche parola a tutte le persone che mi hanno sostenuto o semplicemente sono rimaste al mio fianco durante questo percorso, iniziato ormai più di due anni fa.

Grazie di cuore ai miei genitori, che non hanno mai smesso di sostenermi, credere in me e incoraggiarmi, in ogni mia scelta, dandomi fiducia qualunque strada avessi scelto di intrapendere. Tutto il mio impegno e la mia determinazione è dovuta anche dal desiderio di renderli orgogliosi e di ripagare, almeno in parte, tutto ciò che hanno fatto per me. Li ringrazio soprattutto per i valori che mi hanno trasmesso e che mi hanno portato a diventare la persona che sono oggi.

Ringrazio anche mio fratello, per il sostegno reciproco che ci siamo dati, per quelle poche chiamate che comunque hanno fatto la differenza. So che posso contare su di lui anche se non ce lo diciamo spesso.

Ringrazio i miei zii, i miei cugini e la mia commare che in questi anni non hanno mai fatto mancare il loro affetto e il loro interesse per il mio percorso. Un pensiero particolare va a Sofia, per la sua presenza costante, per i messaggi con cui si è sincerata di come stessi, e per il bel rapporto che si è creato, fatto di confidenza, vicinanza e tanti piccoli gesti che hanno significato molto per me.

Ringrazio Vincenzo, Alessio, Nicola e Rosario, gli amici di sempre con cui ho condiviso tantissimi momenti di spensieratezza e divertimento. Per tutte quelle serate passate a scherzare e chiacchierare, e in cui, tra una bottiglia e l'altra, ci siamo ritrovati a fare le quattro.

Sono felicissimo di aver incontrato un amico come Gaetano. Ci siamo conosciuti in un periodo in cui entrambi avevamo bisogno l'uno dell'altro, e abbiamo trovato

insieme la forza per superare diversi ostacoli. Questo ci ha legato moltissimo. Da allora siamo rimasti sempre in contatto, anche quando la distanza non ci ha permesso di viverci di persona. Mi auguro che, tra molti anni, potremo guardarci indietro ed essere orgogliosi del percorso che abbiamo fatto per raggiungere i nostri obiettivi.

Una dedica speciale va infine a Francesca, la persona che più mi è stata vicina in questo anno. È arrivata all'improvviso, come un raggio di luce per poi splendere nella mia vita. Nei momenti più difficili, quando tutto sembrava crollare, c'è sempre stata. E nei momenti belli ha saputo gioire con me, orgogliosa dei miei traguardi. Sono profondamente grato per tutto ciò che è stata per me, per l'amore che mi ha dato, per le parole di conforto, per la stabilità e la serenità che ha portato nella mia vita. Per essere stata la persona di cui potermi fidare ciecamente, capace di ascoltare le mie insicurezze senza mai giudicarmi. Mi auguro di poter continuare a ridere, scherzare, vivere nuove esperienze, viaggiare e crescere insieme, maturando ancora l'uno accanto all'altro.

Davide Lupo
Dicembre 2025

Bibliography

[1] Shuo Zhang and Krisztian Balog. Web table extraction, retrieval, and augmentation: A survey. *ACM Trans.Intell.Syst. Technol.*, 2020.

[2] Michael J. Cafarella, Alon Y. Halevy, Daisy Zhe Wang, Eugene Wu, and Yang Zhang. Webtables: exploring the power of tables on the web. *Proc. VLDB Endow.*, 2008.

[3] Oliver Lehmberg, Dominique Ritze, Robert Meusel, and Christian Bizer. A large public corpus of web tables containing time and context metadata. *ACM*, 2016.

[4] Fatemeh Nargesian, Erkang Zhu, Renée J. Miller, Ken Q. Pu, and Patricia C. Arocena. Data lake management: Challenges and opportunities. *Proc. VLDB Endow.*, 2019.

[5] Tobias Bleifuß, Leon Bornemann, Dmitri V. Kalashnikov, Felix Naumann, and Divesh Srivastava. The secret life of wikipedia tables. *Proc. VLDB*, 2021.

[6] Erkang Zhu, Dong Deng, Fatemeh Nargesian, and Renée J. Miller. Josie: Overlap set similarity search for finding joinable tables in data lakes. *ACM*, 2019.

[7] Mahdi Esmailoghli, Jorge-Arnulfo Quiané-Ruiz, and Ziawasch Abedjan. Mate: Multi-attribute table extraction. *ACM*, 2022.

[8] Maximilian Koch, Mahdi Esmailoghli, Sören Auer, and Ziawasch Abedjan. Duplicate table discovery with xash. *Gesellschaft für Informatik e.V.*, 2023.

[9] Luca Zecchini, Tobias Bleifuß, Giovanni Simonini, Sonia Bergamaschi, and Felix Naumann. Determining the largest overlap between tables. *Proc. ACM Manag. Data*, 2024.

[10] Francesco Pugnaloni, Luca Zecchini, Matteo Paganelli, Giovanni Simonini, Matteo Lissandrini, and Felix Naumann. Table overlap estimation through graph embeddings. *Proc. ACM Manag. Data 3*, 2025.

[11] Wikipedia. Definition of what is a sequence. https://en.wikipedia.org/wiki/Sequence, 2025. Accessed: 2025-09-14.

[12] Wikipedia. Longest common subsequence definition. https://en.wikipedia.org/wiki/Longest_common_subsequence, 2025. Accessed: 2025-09-14.

[13] Wikipedia. Longest common substring definition. https://en.wikipedia.org/wiki/Longest_common_substring, 2025. Accessed: 2025-09-14.